

# The Supplemental Isabelle/HOL Library

March 13, 2025

## Contents

<b>1</b>	<b>Implementation of Association Lists</b>	<b>21</b>
1.1	<i>update</i> and <i>updates</i> . . . . .	21
1.2	<i>delete</i> . . . . .	23
1.3	<i>update-with-aux</i> and <i>delete-aux</i> . . . . .	24
1.4	<i>restrict</i> . . . . .	26
1.5	<i>clearjunk</i> . . . . .	27
1.6	<i>map-ran</i> . . . . .	28
1.7	<i>merge</i> . . . . .	29
1.8	<i>compose</i> . . . . .	30
1.9	<i>map-entry</i> . . . . .	32
1.10	<i>map-default</i> . . . . .	32
<b>2</b>	<b>Axiomatic Declaration of Bounded Natural Functors</b>	<b>33</b>
<b>3</b>	<b>Generalized Corecursor Sugar (<i>corec</i> and friends)</b>	<b>33</b>
3.1	Coinduction . . . . .	34
<b>4</b>	<b>A general “while” combinator</b>	<b>36</b>
4.1	Partial version . . . . .	37
4.2	Total version . . . . .	37
<b>5</b>	<b>The Bourbaki-Witt tower construction for transfinite iteration</b>	<b>41</b>
5.1	Connect with the while combinator for executability on chain-finite lattices. . . . .	44
<b>6</b>	<b>Division with modulus centered towards zero.</b>	<b>46</b>
<b>7</b>	<b>Order on characters</b>	<b>48</b>
<b>8</b>	<b>A generic phantom type</b>	<b>49</b>

<b>9</b>	<b>Cardinality of types</b>	<b>50</b>
9.1	Preliminary lemmas . . . . .	50
9.2	Cardinalities of types . . . . .	50
9.3	Classes with at least 1 and 2 . . . . .	51
9.4	A type class for deciding finiteness of types . . . . .	52
9.5	A type class for computing the cardinality of types . . . . .	52
9.6	Instantiations for <i>card-UNIV</i> . . . . .	52
<b>10</b>	<b>Code setup for sets with cardinality type information</b>	<b>56</b>
<b>11</b>	<b>Eliminating pattern matches</b>	<b>58</b>
<b>12</b>	<b>Lazy types in generated code</b>	<b>59</b>
12.1	The type <i>lazy</i> . . . . .	59
12.2	Implementation . . . . .	63
<b>13</b>	<b>Test infrastructure for the code generator</b>	<b>64</b>
13.1	YXML encoding for <i>term</i> . . . . .	64
13.2	Test engine and drivers . . . . .	66
<b>14</b>	<b>A combinator to build partial equivalence relations from a predicate and an equivalence relation</b>	<b>66</b>
<b>15</b>	<b>Formalisation of chain-complete partial orders, continuity and admissibility</b>	<b>67</b>
15.1	Continuity . . . . .	70
15.1.1	Theorem collection <i>cont-intro</i> . . . . .	71
15.2	Admissibility . . . . .	77
15.3	(=) as order . . . . .	81
15.4	ccpo for products . . . . .	81
15.5	Complete lattices as ccpo . . . . .	86
15.6	Parallel fixpoint induction . . . . .	89
<b>16</b>	<b>Confluence</b>	<b>93</b>
<b>17</b>	<b>Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums</b>	<b>96</b>
17.1	The datatype universe . . . . .	96
17.2	Freeness: Distinctness of Constructors . . . . .	98
17.3	Set Constructions . . . . .	101
<b>18</b>	<b>Bijections between natural numbers and other types</b>	<b>105</b>
18.1	Type <i>nat</i> $\times$ <i>nat</i> . . . . .	105
18.2	Type <i>nat</i> + <i>nat</i> . . . . .	107
18.3	Type <i>int</i> . . . . .	108

18.4	Type <i>nat list</i> . . . . .	108
18.5	Finite sets of naturals . . . . .	109
18.5.1	Preliminaries . . . . .	109
18.5.2	From sets to naturals . . . . .	110
18.5.3	From naturals to sets . . . . .	110
18.5.4	Proof of isomorphism . . . . .	111
<b>19</b>	<b>Encoding (almost) everything into natural numbers</b>	<b>111</b>
19.1	The class of countable types . . . . .	111
19.2	Conversion functions . . . . .	112
19.3	Finite types are countable . . . . .	112
19.4	Automatically proving countability of old-style datatypes . . . . .	112
19.5	Automatically proving countability of datatypes . . . . .	113
19.6	More Countable types . . . . .	113
19.7	The rationals are countably infinite . . . . .	114
<b>20</b>	<b>Infinite Sets and Related Concepts</b>	<b>115</b>
20.1	The set of natural numbers is infinite . . . . .	115
20.2	The set of integers is also infinite . . . . .	116
20.3	Infinitely Many and Almost All . . . . .	116
20.4	Enumeration of an Infinite Set . . . . .	118
20.5	Properties of <i>wellorder-class.enumerate</i> on finite sets . . . . .	120
<b>21</b>	<b>Countable sets</b>	<b>122</b>
21.1	Predicate for countable sets . . . . .	122
21.2	Enumerate a countable set . . . . .	123
21.3	Closure properties of countability . . . . .	125
21.4	Misc lemmas . . . . .	128
21.5	Uncountable . . . . .	129
<b>22</b>	<b>Countable Complete Lattices</b>	<b>130</b>
22.0.1	Instances of countable complete lattices . . . . .	135
<b>23</b>	<b>Type of (at Most) Countable Sets</b>	<b>136</b>
23.1	Cardinal stuff . . . . .	136
23.2	The type of countable sets . . . . .	137
23.3	Additional lemmas . . . . .	143
23.3.1	<i>cempty</i> . . . . .	143
23.3.2	<i>cinsert</i> . . . . .	143
23.3.3	<i>cimage</i> . . . . .	143
23.3.4	bounded quantification . . . . .	143
23.3.5	<i>cUnion</i> . . . . .	144
23.4	Setup for Lifting/Transfer . . . . .	144
23.4.1	Relator and predicator properties . . . . .	144

23.4.2	Transfer rules for the Transfer package . . . . .	145
23.5	Registration as BNF . . . . .	146
<b>24</b>	<b>Debugging facilities for code generated towards Isabelle/ML</b>	<b>147</b>
<b>25</b>	<b>Sequence of Properties on Subsequences</b>	<b>148</b>
<b>26</b>	<b>Common discrete functions</b>	<b>150</b>
26.1	Discrete logarithm . . . . .	150
26.2	Discrete square root . . . . .	151
<b>27</b>	<b>Pi and Function Sets</b>	<b>152</b>
27.1	Basic Properties of <i>Pi</i> . . . . .	153
27.2	Composition With a Restricted Domain: <i>compose</i> . . . . .	155
27.3	Bounded Abstraction: <i>restrict</i> . . . . .	155
27.4	Bijections Between Sets . . . . .	157
27.5	Extensionality . . . . .	157
27.6	Cardinality . . . . .	158
27.7	Extensional Function Spaces . . . . .	159
27.7.1	Injective Extensional Function Spaces . . . . .	162
27.7.2	Misc properties of functions, composition and restriction from HOL Light . . . . .	162
27.7.3	Cardinality . . . . .	163
27.8	The pigeonhole principle . . . . .	163
<b>28</b>	<b>Partitions and Disjoint Sets</b>	<b>163</b>
28.1	Set of Disjoint Sets . . . . .	163
28.1.1	Family of Disjoint Sets . . . . .	164
28.2	Construct Disjoint Sequences . . . . .	167
28.3	Partitions . . . . .	167
28.4	Constructions of partitions . . . . .	168
28.5	Finiteness of partitions . . . . .	168
28.6	Equivalence of partitions and equivalence classes . . . . .	169
28.7	Refinement of partitions . . . . .	169
28.8	The coarsest common refinement of a set of partitions . . . . .	169
<b>29</b>	<b>Type of finite sets defined as a subtype of sets</b>	<b>170</b>
29.1	Definition of the type . . . . .	171
29.2	Basic operations and type class instantiations . . . . .	171
29.3	Other operations . . . . .	173
29.4	Transferred lemmas from Set.thy . . . . .	176
29.5	Additional lemmas . . . . .	192
29.5.1	<i>ffUnion</i> . . . . .	192
29.5.2	<i>fbind</i> . . . . .	192
29.5.3	<i>fsingleton</i> . . . . .	192

29.5.4	<i>fempty</i>	192
29.5.5	<i>fset</i>	193
29.5.6	<i>ffilter</i>	193
29.5.7	<i>fset-of-list</i>	193
29.5.8	<i>finsert</i>	194
29.5.9	<i>fimage</i>	194
29.5.10	bounded quantification	194
29.5.11	<i>fcard</i>	195
29.5.12	<i>sorted-list-of-fset</i>	197
29.5.13	<i>ffold</i>	197
29.5.14	$( \subset )$	198
29.5.15	Group operations	198
29.5.16	Semilattice operations	198
29.6	Choice in fsets	200
29.7	Induction and Cases rules for fsets	201
29.8	Lemmas depending on induction	201
29.9	Setup for Lifting/Transfer	202
29.9.1	Relator and predicator properties	202
29.9.2	Transfer rules for the Transfer package	202
29.10	BNF setup	204
29.11	Size setup	205
29.12	Advanced relator customization	205
29.12.1	Countability	205
29.13	Quickcheck setup	206
29.14	Code Generation Setup	207
<b>30</b>	<b>Type of finite maps defined as a subtype of maps</b>	<b>208</b>
30.1	Auxiliary constants and lemmas over <i>map</i>	208
30.2	Abstract characterisation	209
30.3	Operations	210
30.4	BNF setup	223
30.5	<i>size</i> setup	227
30.6	Additional operations	227
30.7	Additional properties	229
30.8	Lifting/transfer setup	229
30.9	View as datatype	229
30.10	Code setup	230
30.11	Instances	231
30.12	Tests	232
<b>31</b>	<b>Disjoint FSets</b>	<b>232</b>

<b>32 Lists with elements distinct as canonical example for datatype invariants</b>	<b>233</b>
32.1 The type of distinct lists . . . . .	234
32.2 Executable version obeying invariant . . . . .	236
32.3 Induction principle and case distinction . . . . .	237
32.4 Functorial structure . . . . .	237
32.5 Quickcheck generators . . . . .	237
32.6 BNF instance . . . . .	237
<b>33 Type of dual ordered lattices</b>	<b>238</b>
33.1 Pointwise ordering . . . . .	240
33.2 Binary infimum and supremum . . . . .	241
33.3 Top and bottom elements . . . . .	242
33.4 Complement . . . . .	243
33.5 Complete lattice operations . . . . .	244
<b>34 Equipollence and Other Relations Connected with Cardinality</b>	<b>245</b>
34.1 Eqpoll . . . . .	245
34.2 The strict relation . . . . .	248
34.3 Mapping by an injection . . . . .	249
34.4 Inserting elements into sets . . . . .	249
34.5 Binary sums and unions . . . . .	250
34.6 Binary Cartesian products . . . . .	250
34.7 General Unions . . . . .	251
34.8 General Cartesian products (Pi) . . . . .	252
34.9 Misc other resultd . . . . .	253
<b>35 Continuity and iterations</b>	<b>257</b>
35.1 Continuity for complete lattices . . . . .	258
35.1.1 Least fixed points in countable complete lattices . . . . .	260
<b>36 Extended natural numbers (i.e. with infinity)</b>	<b>261</b>
36.1 Type definition . . . . .	261
36.2 Constructors and numbers . . . . .	262
36.3 Addition . . . . .	264
36.4 Multiplication . . . . .	264
36.5 Numerals . . . . .	265
36.6 Subtraction . . . . .	265
36.7 Ordering . . . . .	266
36.8 Cancellation simprocs . . . . .	269
36.9 Well-ordering . . . . .	270
36.10 Complete Lattice . . . . .	270
36.11 Traditional theorem names . . . . .	271

<b>37</b>	<b>Liminf and Limsup on conditionally complete lattices</b>	<b>271</b>
37.0.1	<i>Liminf</i> and <i>Limsup</i> . . . . .	273
37.1	More Limits . . . . .	277
<b>38</b>	<b>Extended real number line</b>	<b>278</b>
38.1	Definition and basic properties . . . . .	280
38.1.1	Addition . . . . .	282
38.1.2	Linear order on <i>ereal</i> . . . . .	284
38.1.3	Multiplication . . . . .	290
38.1.4	Power . . . . .	295
38.1.5	Subtraction . . . . .	296
38.1.6	Division . . . . .	299
38.2	Complete lattice . . . . .	304
38.3	Extended real intervals . . . . .	305
38.4	Topological space . . . . .	307
38.5	Relation to <i>enat</i> . . . . .	313
38.6	Limits on <i>ereal</i> . . . . .	315
38.6.1	Convergent sequences . . . . .	316
38.6.2	Sums . . . . .	321
38.6.3	Continuity . . . . .	326
38.6.4	liminf and limsup . . . . .	328
38.6.5	Tests for code generator . . . . .	331
<b>39</b>	<b>Indicator Function</b>	<b>331</b>
<b>40</b>	<b>The type of non-negative extended real numbers</b>	<b>335</b>
40.1	Defining the extended non-negative reals . . . . .	338
40.2	Cancellation simprocs . . . . .	341
40.3	Order with top . . . . .	341
40.4	Arithmetic . . . . .	344
40.5	Coercion from <i>real</i> to <i>ennreal</i> . . . . .	347
40.6	Coercion from <i>ennreal</i> to <i>real</i> . . . . .	352
40.7	Coercion from <i>enat</i> to <i>ennreal</i> . . . . .	353
40.8	Topology on <i>ennreal</i> . . . . .	354
40.9	Approximation lemmas . . . . .	360
40.10	<i>ennreal</i> theorems . . . . .	362
<b>41</b>	<b>Logarithm of Natural Numbers</b>	<b>366</b>
41.1	Preliminaries . . . . .	366
41.2	Floorlog . . . . .	366
41.3	. . . . .	367
41.4	Bitlen . . . . .	369

<b>42</b>	<b>Various algebraic structures combined with a lattice</b>	<b>370</b>
42.1	Positive Part, Negative Part, Absolute Value . . . . .	371
<b>43</b>	<b>Floating-Point Numbers</b>	<b>375</b>
43.1	Real operations preserving the representation as floating point number . . . . .	376
43.2	Arithmetic operations on floating point numbers . . . . .	378
43.3	Quickcheck . . . . .	380
43.4	Represent floats as unique mantissa and exponent . . . . .	381
43.5	Compute arithmetic operations . . . . .	382
43.6	Lemmas for types <i>real</i> , <i>nat</i> , <i>int</i> . . . . .	384
43.7	Rounding Real Numbers . . . . .	384
43.8	Rounding Floats . . . . .	385
43.9	Truncating Real Numbers . . . . .	387
43.10	Truncating Floats . . . . .	388
43.11	Approximation of positive rationals . . . . .	390
43.12	Division . . . . .	393
43.13	Approximate Addition . . . . .	393
43.14	Approximate Multiplication . . . . .	395
43.15	Approximate Power . . . . .	396
43.16	Lemmas needed by Approximate . . . . .	399
<b>44</b>	<b>Pointwise instantiation of functions to algebra type classes</b>	<b>403</b>
<b>45</b>	<b>Pointwise instantiation of functions to division</b>	<b>407</b>
45.1	Syntactic with division . . . . .	407
<b>46</b>	<b>Lexicographic order on functions</b>	<b>408</b>
<b>47</b>	<b>The <i>going-to</i> filter</b>	<b>409</b>
<b>48</b>	<b>Big sum and product over function bodies</b>	<b>411</b>
48.1	Abstract product . . . . .	411
48.2	Concrete sum . . . . .	413
48.3	Concrete product . . . . .	414
<b>49</b>	<b>Infinite Type Class</b>	<b>415</b>
<b>50</b>	<b>Algebraic operations on sets</b>	<b>416</b>
<b>51</b>	<b>Interval Type</b>	<b>422</b>
51.1	Membership . . . . .	426
51.2	Quickcheck . . . . .	433



<b>52</b>	<b>Approximate Operations on Intervals of Floating Point Numbers</b>	<b>435</b>
52.1	Intervals with Floating Point Bounds . . . . .	436
52.2	intros for <i>real-interval</i> . . . . .	437
52.3	bounds for lists . . . . .	438
52.4	constants for code generation . . . . .	441
<b>53</b>	<b>Immutable Arrays with Code Generation</b>	<b>441</b>
53.1	Fundamental operations . . . . .	441
53.2	Generic code equations . . . . .	442
53.3	Auxiliary operations for code generation . . . . .	443
53.4	Code Generation for SML . . . . .	444
53.5	Code Generation for Haskell . . . . .	444
<b>54</b>	<b>Definition of Landau symbols</b>	<b>445</b>
54.1	Definition of Landau symbols . . . . .	445
54.2	Landau symbols and limits . . . . .	459
54.3	Flatness of real functions . . . . .	465
54.4	Asymptotic Equivalence . . . . .	465
<b>55</b>	<b>Values extended by a bottom element</b>	<b>472</b>
55.1	Values extended by a top element . . . . .	474
55.2	Values extended by a top and a bottom element . . . . .	477
<b>56</b>	<b>Infinite Streams</b>	<b>479</b>
56.1	prepend list to stream . . . . .	480
56.2	set of streams with elements in some fixed set . . . . .	480
56.3	nth, take, drop for streams . . . . .	481
56.4	unary predicates lifted to streams . . . . .	484
56.5	recurring stream out of a list . . . . .	484
56.6	iterated application of a function . . . . .	485
56.7	stream repeating a single element . . . . .	486
56.8	stream of natural numbers . . . . .	486
56.9	flatten a stream of lists . . . . .	486
56.10	merge a stream of streams . . . . .	487
56.11	product of two streams . . . . .	487
56.12	interleave two streams . . . . .	487
56.13	zip . . . . .	488
56.14	zip via function . . . . .	488
<b>57</b>	<b>List prefixes, suffixes, and homeomorphic embedding</b>	<b>489</b>
57.1	Prefix order on lists . . . . .	489
57.2	Basic properties of prefixes . . . . .	490
57.3	Prefixes . . . . .	493

57.4	Longest Common Prefix . . . . .	494
57.5	Parallel lists . . . . .	495
57.6	Suffix order on lists . . . . .	496
57.7	Suffixes . . . . .	500
57.8	Homeomorphic embedding on lists . . . . .	501
57.9	Subsequences (special case of homeomorphic embedding) . . . . .	503
57.10	Appending elements . . . . .	504
57.11	Relation to standard list operations . . . . .	505
57.12	Contiguous sublists . . . . .	505
57.12.1	<i>sublist</i> . . . . .	505
57.12.2	<i>sublists</i> . . . . .	507
57.13	Parametricity . . . . .	508
<b>58</b>	<b>Linear Temporal Logic on Streams</b>	<b>509</b>
<b>59</b>	<b>Preliminaries</b>	<b>510</b>
<b>60</b>	<b>Linear temporal logic</b>	<b>510</b>
<b>61</b>	<b>Weak vs. strong until (contributed by Michael Foster, University of Sheffield)</b>	<b>520</b>
<b>62</b>	<b>Lists as vectors</b>	<b>521</b>
62.1	+ and - . . . . .	521
62.2	Inner product . . . . .	523
<b>63</b>	<b>Definitions of Least Upper Bounds and Greatest Lower Bounds</b>	<b>523</b>
63.1	Rules for the Relations $*\leq$ and $\leq*$ . . . . .	524
63.2	Rules about the Operators <i>leastP</i> , <i>ub</i> and <i>lub</i> . . . . .	524
63.3	Rules about the Operators <i>greatestP</i> , <i>isLb</i> and <i>isGlb</i> . . . . .	526
<b>64</b>	<b>An abstract view on maps for code generation.</b>	<b>528</b>
64.1	Parametricity transfer rules . . . . .	528
64.2	Type definition and primitive operations . . . . .	530
64.3	Functorial structure . . . . .	531
64.4	Derived operations . . . . .	531
64.5	Properties . . . . .	533
64.5.1	<i>entries</i> , <i>ordered-entries</i> , and <i>fold</i> . . . . .	541
64.6	Code generator setup . . . . .	544
<b>65</b>	<b>Monad notation for arbitrary types</b>	<b>544</b>
<b>66</b>	<b>Less common functions on lists</b>	<b>545</b>

<b>67 (Finite) Multisets</b>	<b>552</b>
67.1 The type of multisets	552
67.2 Representing multisets	553
67.3 Basic operations	554
67.3.1 Conversion to set and membership	554
67.3.2 Union	557
67.3.3 Difference	557
67.3.4 Min and Max	559
67.3.5 Equality of multisets	560
67.3.6 Pointwise ordering induced by count	561
67.3.7 Intersection and bounded union	565
67.3.8 Additional intersection facts	565
67.3.9 Additional bounded union facts	567
67.4 Replicate and repeat operations	568
67.4.1 Simprocs	569
67.4.2 Conditionally complete lattice	570
67.4.3 Filter (with comprehension syntax)	572
67.4.4 Size	574
67.5 Induction and case splits	576
67.5.1 Strong induction and subset induction for multisets	577
67.6 Least and greatest elements	578
67.7 The fold combinator	578
67.8 Image	579
67.9 Further conversions	582
67.10 More properties of the replicate, repeat, and image operations	587
67.11 Big operators	589
67.12 Multiset as order-ignorant lists	595
67.13 The multiset order	598
67.13.1 Well-foundedness	599
67.13.2 Closure-free presentation	599
67.13.3 Monotonicity	600
67.13.4 The multiset extension is cancellative for multiset union	601
67.13.5 Strict partial-order properties	601
67.13.6 Strict total-order properties	603
67.14 Quasi-executable version of the multiset extension	603
67.14.1 Monotonicity of multiset union	604
67.14.2 Termination proofs with multiset orders	604
67.15 Legacy theorem bindings	605
67.16 Naive implementation using lists	607
67.17 BNF setup	609
67.18 Size setup	612
67.19 Lemmas about Size	612

<b>68 More Theorems about the Multiset Order</b>	<b>613</b>
68.1 Alternative Characterizations . . . . .	613
68.1.1 The Dershowitz–Manna Ordering . . . . .	613
68.1.2 The Huet–Oppen Ordering . . . . .	613
68.1.3 Monotonicity . . . . .	614
68.1.4 Properties of Orders . . . . .	615
68.1.5 Simplifications . . . . .	619
68.2 Simprocs . . . . .	619
68.3 Additional facts and instantiations . . . . .	619
<b>69 Fixed Length Lists</b>	<b>621</b>
<b>70 Non-negative, non-positive integers and reals</b>	<b>623</b>
70.1 Non-positive integers . . . . .	623
70.2 Non-negative reals . . . . .	625
70.3 Non-positive reals . . . . .	626
<b>71 Numeral Syntax for Types</b>	<b>628</b>
71.1 Numeral Types . . . . .	628
71.2 <i>num1</i> . . . . .	629
71.3 Locales for modular arithmetic subtypes . . . . .	630
71.4 Ring class instances . . . . .	632
71.5 Order instances . . . . .	633
71.6 Code setup and type classes for code generation . . . . .	634
71.7 Syntax . . . . .	636
71.8 Examples . . . . .	636
<b>72 <math>\omega</math>-words</b>	<b>636</b>
72.1 Type declaration and elementary operations . . . . .	637
72.2 Subsequence, Prefix, and Suffix . . . . .	638
72.3 Prepending . . . . .	640
72.4 The limit set of an $\omega$ -word . . . . .	641
72.5 Index sequences and piecewise definitions . . . . .	643
<b>73 Combinator syntax for generic, open state monads (single-threaded monads)</b>	<b>645</b>
73.1 Motivation . . . . .	645
73.2 State transformations and combinators . . . . .	645
73.3 Monad laws . . . . .	646
73.4 Do-syntax . . . . .	647
<b>74 Canonical order on option type</b>	<b>647</b>

<b>75 Futures and parallel lists for code generated towards Isabelle/ML</b>	<b>651</b>
75.1 Futures . . . . .	652
75.2 Parallel lists . . . . .	652
<b>76 Input syntax for pattern aliases (or “as-patterns” in Haskell)</b>	<b>653</b>
76.1 Definition . . . . .	654
76.2 Usage . . . . .	654
<b>77 Periodic Functions</b>	<b>655</b>
<b>78 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view</b>	<b>657</b>
78.1 Preliminary: auxiliary operations for <i>almost everywhere zero</i> .	657
78.2 Type definition . . . . .	660
78.3 Additive structure . . . . .	662
78.4 Multiplicative structure . . . . .	664
78.5 Single-point mappings . . . . .	665
78.6 Integral domains . . . . .	667
78.7 Mapping order . . . . .	667
78.8 Fundamental mapping notions . . . . .	668
78.9 Degree . . . . .	670
78.10 Inductive structure . . . . .	671
78.11 Quasi-functorial structure . . . . .	671
78.12 Canonical dense representation of $\text{nat} \Rightarrow_0 'a$ . . . . .	673
78.13 Canonical sparse representation of $'a \Rightarrow_0 'b$ . . . . .	674
78.14 Size estimation . . . . .	675
78.15 Further mapping operations and properties . . . . .	676
78.16 Free Abelian Groups Over a Type . . . . .	676
<b>79 Exponentiation by Squaring</b>	<b>680</b>
<b>80 Preorders with explicit equivalence relation</b>	<b>681</b>
<b>81 Additive group operations on product types</b>	<b>682</b>
81.1 Operations . . . . .	682
81.2 Class instances . . . . .	683
<b>82 Roots of real quadratics</b>	<b>684</b>
<b>83 Pretty syntax for Quotient operations</b>	<b>686</b>
<b>84 Quotient infrastructure for the set type</b>	<b>686</b>
84.1 Contravariant set map (vimage) and set relator, rules for the Quotient package . . . . .	687

<b>85 Quotient infrastructure for the product type</b>	<b>688</b>
85.1 Rules for the Quotient package . . . . .	688
<b>86 Quotient infrastructure for the option type</b>	<b>690</b>
86.1 Rules for the Quotient package . . . . .	690
<b>87 Quotient infrastructure for the list type</b>	<b>692</b>
87.1 Rules for the Quotient package . . . . .	692
<b>88 Quotient infrastructure for the sum type</b>	<b>695</b>
88.1 Rules for the Quotient package . . . . .	695
<b>89 Quotient types</b>	<b>696</b>
89.1 Equivalence relations and quotient types . . . . .	697
89.2 Equality on quotients . . . . .	698
89.3 Picking representing elements . . . . .	698
<b>90 Ramsey’s Theorem</b>	<b>698</b>
90.1 Preliminary definitions . . . . .	699
90.1.1 The $n$ -element subsets of a set $A$ . . . . .	699
90.1.2 Further properties, involving equipollence . . . . .	701
90.1.3 Partition predicates . . . . .	702
90.2 Finite versions of Ramsey’s theorem . . . . .	702
90.2.1 The Erds–Szekeres theorem exhibits an upper bound for Ramsey numbers . . . . .	703
90.2.2 Trivial cases . . . . .	703
90.2.3 Ramsey’s theorem with TWO colours and arbitrary exponents (hypergraph version) . . . . .	703
90.2.4 Full Ramsey’s theorem with multiple colours and ar- bitrary exponents . . . . .	704
90.2.5 Simple graph version . . . . .	704
90.3 Preliminaries for the infinitary version . . . . .	704
90.3.1 “Axiom” of Dependent Choice . . . . .	704
90.3.2 Partition functions . . . . .	705
90.4 Ramsey’s Theorem: Infinitary Version . . . . .	705
90.5 Disjunctive Well-Foundedness . . . . .	706
<b>91 Modulo and congruence on the reals</b>	<b>707</b>
<b>92 Generic reflection and reification</b>	<b>710</b>
<b>93 Assigning lengths to types by type classes</b>	<b>711</b>
<b>94 Saturated arithmetic</b>	<b>713</b>
94.1 The type of saturated naturals . . . . .	713

<b>95 Set Idioms</b>	<b>717</b>
95.1 Idioms for being a suitable union/intersection of something . . .	717
95.2 The “Relative to” operator . . . . .	721
<b>96 Signed division: negative results rounded towards zero rather than minus infinity.</b>	<b>725</b>
<b>97 State monad</b>	<b>728</b>
<b>98 Comparators on linear quasi-orders</b>	<b>732</b>
98.1 Basic properties . . . . .	732
98.2 Fundamental comparator combinators . . . . .	736
98.3 Direct implementations for linear orders on selected types . .	737
<b>99 Stably sorted lists</b>	<b>737</b>
<b>100 Alternative sorting algorithms</b>	<b>741</b>
100.1 Quicksort . . . . .	741
100.2 Mergesort . . . . .	742
<b>101A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming</b>	<b>743</b>
<b>102A table-based implementation of the reflexive transitive closure</b>	<b>744</b>
<b>103 Binary Tree</b>	<b>745</b>
103.1 <i>map-tree</i> . . . . .	747
103.2 <i>size</i> . . . . .	747
103.3 <i>set-tree</i> . . . . .	748
103.4 <i>subtrees</i> . . . . .	748
103.5 <i>height</i> and <i>min-height</i> . . . . .	748
103.6 <i>complete</i> . . . . .	749
103.7 <i>acomplete</i> . . . . .	750
103.8 <i>wbalanced</i> . . . . .	750
103.9 <i>ipl</i> . . . . .	750
103.10 list of entries . . . . .	750
103.1 Binary Search Tree . . . . .	751
103.12 <i>heap</i> . . . . .	752
103.13 <i>mirror</i> . . . . .	752
<b>104 Multiset of Elements of Binary Tree</b>	<b>752</b>
<b>105 Unordered pairs</b>	<b>754</b>

<b>106A type of finite bit strings</b>	<b>758</b>
106.1 Preliminaries	758
106.2 Fundamentals	758
106.2.1 Type definition	758
106.2.2 Basic arithmetic	758
106.2.3 Basic tool setup	760
106.2.4 Basic code generation setup	760
106.2.5 Basic conversions	761
106.2.6 Basic ordering	768
106.3 Enumeration	769
106.4 Bit-wise operations	770
106.5 Conversions including casts	774
106.5.1 Generic unsigned conversion	774
106.5.2 Generic signed conversion	777
106.5.3 More	778
106.6 Arithmetic operations	782
106.7 Ordering	784
106.8 Bit-wise operations	786
106.9 More shift operations	788
106.10 Single-bit operations	788
106.11 Rotation	789
106.12 Split and cat operations	790
106.13 More on conversions	791
106.14 Testing bits	794
106.15 Word Arithmetic	797
106.16 Transferring goals from words to ints	801
106.17 Order on fixed-length words	803
106.18 Conditions for the addition (etc) of two words to overflow	804
106.19 Some proof tool support	806
106.20 More on overflows and monotonicity	807
106.21 Arithmetic type class instantiations	810
106.22 Word and nat	810
106.23 Cardinality, finiteness of set of words	814
106.24 Bitwise Operations on Words	815
106.24.1 Shift functions in terms of lists of bools	819
106.24.2 Mask	821
106.24.3 Slices	823
106.24.4 Recast	824
106.25 Split and cat	825
106.25.1 Split and slice	826
106.26 Rotation	826
106.26.1 Word rotation commutes with bit-wise operations	827
106.27 Maximum machine word	828
106.28 Recursion combinator for words	831



106.2	Tool support . . . . .	832
<b>107</b>	<b>The Field of Integers mod 2</b>	<b>832</b>
<b>108</b>	<b>Pointwise order on product types</b>	<b>836</b>
108.1	Pointwise ordering . . . . .	837
108.2	Binary infimum and supremum . . . . .	837
108.3	Top and bottom elements . . . . .	838
108.4	Complete lattice operations . . . . .	839
108.5	Complete distributive lattices . . . . .	841
108.6	Bekic's Theorem . . . . .	841
<b>109</b>	<b>Finite Lattices</b>	<b>841</b>
109.1	Finite Complete Lattices . . . . .	841
109.2	Finite Distributive Lattices . . . . .	844
109.3	Linear Orders . . . . .	844
109.4	Finite Linear Orders . . . . .	845
<b>110</b>	<b>Lexicographic order on lists</b>	<b>846</b>
<b>111</b>	<b>Lexicographic order on lists</b>	<b>847</b>
<b>112</b>	<b>Prefix order on lists as order class instance</b>	<b>849</b>
<b>113</b>	<b>Lexicographic order on product types</b>	<b>850</b>
<b>114</b>	<b>Subsequence Ordering</b>	<b>851</b>
114.1	Definitions and basic lemmas . . . . .	852
<b>115</b>	<b>Records based on BNF/datatype machinery</b>	<b>853</b>
<b>116</b>	<b>Implementation of mappings with Association Lists</b>	<b>855</b>
<b>117</b>	<b>Avoidance of pattern matching on natural numbers</b>	<b>859</b>
117.1	Case analysis . . . . .	860
117.2	Preprocessors . . . . .	860
117.3	Candidates which need special treatment . . . . .	860
<b>118</b>	<b>Implementation of natural numbers as binary numerals</b>	<b>860</b>
118.1	Representation . . . . .	861
118.2	Basic arithmetic . . . . .	861
118.3	Conversions . . . . .	863
<b>119</b>	<b>Code generation of prolog programs</b>	<b>863</b>
<b>120</b>	<b>Setup for Numerals</b>	<b>864</b>

<b>121</b>	<b>Implementation of integer numbers by target-language integers</b>	<b>864</b>
<b>122</b>	<b>Implementation of natural numbers by target-language integers</b>	<b>872</b>
122.1	Implementation for <i>nat</i> . . . . .	872
<b>123</b>	<b>Implementation of bit-shifts on target-language integers by built-in operations</b>	<b>875</b>
<b>124</b>	<b>Implementation of natural and integer numbers by target-language integers</b>	<b>879</b>
<b>125</b>	<b>Preprocessor setup for floats implemented by target language numerals</b>	<b>879</b>
<b>126</b>	<b>Abstract type of association lists with unique keys</b>	<b>880</b>
126.1	Preliminaries . . . . .	881
126.2	Type ( <i>'key, 'value</i> ) <i>alist</i> . . . . .	881
126.3	Primitive operations . . . . .	881
126.4	Abstract operation properties . . . . .	882
126.5	Further operations . . . . .	882
126.5.1	Equality . . . . .	882
126.5.2	Size . . . . .	882
126.6	Quickcheck generators . . . . .	883
<b>127</b>	<b>list is a BNF</b>	<b>885</b>
<b>128</b>	<b>Multisets partially implemented by association lists</b>	<b>885</b>
<b>129</b>	<b>Implementation of Red-Black Trees</b>	<b>890</b>
129.1	Datatype of RB trees . . . . .	890
129.2	Tree properties . . . . .	890
129.2.1	Content of a tree . . . . .	890
129.2.2	Search tree properties . . . . .	891
129.2.3	Tree lookup . . . . .	892
129.2.4	Red-black properties . . . . .	893
129.3	Insertion . . . . .	894
129.4	Deletion . . . . .	898
129.5	Modifying existing entries . . . . .	903
129.6	Mapping all entries . . . . .	904
129.7	Folding over entries . . . . .	904
129.8	Bulkloading a tree . . . . .	905
129.9	Building a RBT from a sorted list . . . . .	905
129.10	Union and intersection of sorted associative lists . . . . .	911

129.1	Code generator setup . . . . .	929
<b>130</b>	<b>Abstract type of RBT trees</b>	<b>930</b>
130.1	Type definition . . . . .	930
130.2	Primitive operations . . . . .	931
130.3	Derived operations . . . . .	932
130.4	Abstract lookup properties . . . . .	932
130.5	Quickcheck generators . . . . .	935
130.6	Hide implementation details . . . . .	935
<b>131</b>	<b>Implementation of mappings with Red-Black Trees</b>	<b>935</b>
131.1	Data type and invariant . . . . .	935
131.2	Operations . . . . .	936
131.3	Invariant preservation . . . . .	936
131.4	Map Semantics . . . . .	937
<b>132</b>	<b>Implementation of sets using RBT trees</b>	<b>937</b>
<b>133</b>	<b>Definition of code datatype constructors</b>	<b>937</b>
<b>134</b>	<b>Deletion of already existing code equations</b>	<b>937</b>
<b>135</b>	<b>Lemmas</b>	<b>938</b>
135.1	Auxiliary lemmas . . . . .	938
135.2	fold and filter . . . . .	938
135.3	foldi and Ball . . . . .	938
135.4	foldi and Bex . . . . .	939
135.5	folding over non empty trees and selecting the minimal and maximal element . . . . .	939
135.5.1	concrete . . . . .	939
135.5.2	abstract . . . . .	942
<b>136</b>	<b>Code equations</b>	<b>943</b>
<b>137</b>	<b>Common constants</b>	<b>948</b>
<b>138</b>	<b>Pairs</b>	<b>949</b>
<b>139</b>	<b>Filters</b>	<b>949</b>
<b>140</b>	<b>Bounded quantifiers</b>	<b>949</b>
<b>141</b>	<b>Operations on Predicates</b>	<b>949</b>
<b>142</b>	<b>Setup for Numerals</b>	<b>949</b>

<b>143</b>	<b>Arithmetic operations</b>	<b>949</b>
143.1	Arithmetic on naturals and integers . . . . .	949
143.2	Inductive definitions for ordering on naturals . . . . .	950
<b>144</b>	<b>Alternative list definitions</b>	<b>950</b>
144.1	Alternative rules for <i>length</i> . . . . .	950
144.2	Alternative rules for <i>list-all2</i> . . . . .	951
144.3	Alternative rules for membership in lists . . . . .	951
<b>145</b>	<b>Setup for <code>String.literal</code></b>	<b>951</b>
<b>146</b>	<b>Simplification rules for optimisation</b>	<b>952</b>
<b>147A</b>	<b>Prototype of Quickcheck based on the Predicate Compiler</b>	<b>952</b>
<b>148</b>	<b>TFL: recursive function definitions</b>	<b>952</b>
148.1	Lemmas for TFL . . . . .	952
148.2	Rule setup . . . . .	953
<b>149</b>	<b>Program extraction from proofs involving datatypes and inductive predicates</b>	<b>954</b>
<b>150</b>	<b>Refute</b>	<b>954</b>
<b>151</b>	<b>Misc lemmas on division, to be sorted out finally</b>	<b>956</b>

# 1 Implementation of Association Lists

```
theory AList
  imports Main
begin
```

```
context
begin
```

The operations preserve distinctness of keys and function *clearjunk* distributes over them. Since *clearjunk* enforces distinctness of keys it can be used to establish the invariant, e.g. for inductive proofs.

## 1.1 *update* and *updates*

```
qualified primrec update :: 'key  $\Rightarrow$  'val  $\Rightarrow$  ('key  $\times$  'val) list  $\Rightarrow$  ('key  $\times$  'val) list
  where
    update k v [] = [(k, v)]
    | update k v (p # ps) = (if fst p = k then (k, v) # ps else p # update k v ps)
```

```
lemma update-conv': map-of (update k v al) = (map-of al)(k $\mapsto$ v)
  <proof>
```

```
corollary update-conv: map-of (update k v al) k' = ((map-of al)(k $\mapsto$ v)) k'
  <proof>
```

```
lemma dom-update: fst ` set (update k v al) = {k}  $\cup$  fst ` set al
  <proof>
```

```
lemma update-keys:
  map fst (update k v al) =
    (if k  $\in$  set (map fst al) then map fst al else map fst al @ [k])
  <proof>
```

```
lemma distinct-update:
  assumes distinct (map fst al)
  shows distinct (map fst (update k v al))
  <proof>
```

```
lemma update-filter:
  a  $\neq$  k  $\implies$  update k v [q $\leftarrow$ ps. fst q  $\neq$  a] = [q $\leftarrow$ update k v ps. fst q  $\neq$  a]
  <proof>
```

```
lemma update-triv: map-of al k = Some v  $\implies$  update k v al = al
  <proof>
```

```
lemma update-nonempty [simp]: update k v al  $\neq$  []
  <proof>
```

**lemma** *update-eqD*:  $\text{update } k \ v \ al = \text{update } k \ v' \ al' \implies v = v'$   
 ⟨proof⟩

**lemma** *update-last* [simp]:  $\text{update } k \ v \ (\text{update } k \ v' \ al) = \text{update } k \ v \ al$   
 ⟨proof⟩

Note that the lists are not necessarily the same:  $\text{update } k \ v \ (\text{update } k' \ v' \ []) = [(k', v'), (k, v)]$  and  $\text{update } k' \ v' \ (\text{update } k \ v \ []) = [(k, v), (k', v')]$ .

**lemma** *update-swap*:  
 $k \neq k' \implies \text{map-of } (\text{update } k \ v \ (\text{update } k' \ v' \ al)) = \text{map-of } (\text{update } k' \ v' \ (\text{update } k \ v \ al))$   
 ⟨proof⟩

**lemma** *update-Some-unfold*:  
 $\text{map-of } (\text{update } k \ v \ al) \ x = \text{Some } y \iff$   
 $x = k \wedge v = y \vee x \neq k \wedge \text{map-of } al \ x = \text{Some } y$   
 ⟨proof⟩

**lemma** *image-update* [simp]:  $x \notin A \implies \text{map-of } (\text{update } x \ y \ al) \ `A = \text{map-of } al \ `A$

⟨proof⟩ **definition** *updates* ::  
 $'key \ list \Rightarrow 'val \ list \Rightarrow ('key \times 'val) \ list \Rightarrow ('key \times 'val) \ list$   
**where**  $\text{updates } ks \ vs = \text{fold } (\text{case-prod } \text{update}) \ (\text{zip } ks \ vs)$

**lemma** *updates-simps* [simp]:  
 $\text{updates } [] \ vs \ ps = ps$   
 $\text{updates } ks \ [] \ ps = ps$   
 $\text{updates } (k \# ks) \ (v \# vs) \ ps = \text{updates } ks \ vs \ (\text{update } k \ v \ ps)$   
 ⟨proof⟩

**lemma** *updates-key-simp* [simp]:  
 $\text{updates } (k \ # \ ks) \ vs \ ps =$   
 $(\text{case } vs \ \text{of } [] \Rightarrow ps \ | \ v \ # \ vs \Rightarrow \text{updates } ks \ vs \ (\text{update } k \ v \ ps))$   
 ⟨proof⟩

**lemma** *updates-conv'*:  $\text{map-of } (\text{updates } ks \ vs \ al) = (\text{map-of } al)(ks[\mapsto]vs)$   
 ⟨proof⟩

**lemma** *updates-conv*:  $\text{map-of } (\text{updates } ks \ vs \ al) \ k = ((\text{map-of } al)(ks[\mapsto]vs)) \ k$   
 ⟨proof⟩

**lemma** *distinct-updates*:  
**assumes**  $\text{distinct } (\text{map } \text{fst } al)$   
**shows**  $\text{distinct } (\text{map } \text{fst } (\text{updates } ks \ vs \ al))$   
 ⟨proof⟩

**lemma** *updates-append1* [simp]:  $\text{size } ks < \text{size } vs \implies$   
 $\text{updates } (ks@[k]) \ vs \ al = \text{update } k \ (vs[\text{size } ks]) \ (\text{updates } ks \ vs \ al)$   
 ⟨proof⟩

**lemma** *updates-list-update-drop* [simp]:  
 $size\ ks \leq i \implies i < size\ vs \implies$   
 $updates\ ks\ (vs[i:=v])\ al = updates\ ks\ vs\ al$   
 ⟨proof⟩

**lemma** *update-updates-conv-if*:  
 $map-of\ (updates\ xs\ ys\ (update\ x\ y\ al)) =$   
 $map-of$   
 $(if\ x \in set\ (take\ (length\ ys)\ xs)$   
 $then\ updates\ xs\ ys\ al$   
 $else\ (update\ x\ y\ (updates\ xs\ ys\ al)))$   
 ⟨proof⟩

**lemma** *updates-twist* [simp]:  
 $k \notin set\ ks \implies$   
 $map-of\ (updates\ ks\ vs\ (update\ k\ v\ al)) = map-of\ (update\ k\ v\ (updates\ ks\ vs\ al))$   
 ⟨proof⟩

**lemma** *updates-apply-notin* [simp]:  
 $k \notin set\ ks \implies map-of\ (updates\ ks\ vs\ al)\ k = map-of\ al\ k$   
 ⟨proof⟩

**lemma** *updates-append-drop* [simp]:  
 $size\ xs = size\ ys \implies updates\ (xs\ @\ zs)\ ys\ al = updates\ xs\ ys\ al$   
 ⟨proof⟩

**lemma** *updates-append2-drop* [simp]:  
 $size\ xs = size\ ys \implies updates\ xs\ (ys\ @\ zs)\ al = updates\ xs\ ys\ al$   
 ⟨proof⟩

## 1.2 delete

**qualified definition** *delete* :: 'key  $\Rightarrow$  ('key  $\times$  'val) list  $\Rightarrow$  ('key  $\times$  'val) list  
**where** *delete-eq*:  $delete\ k = filter\ (\lambda(k', -). k \neq k')$

**lemma** *delete-simps* [simp]:  
 $delete\ k\ [] = []$   
 $delete\ k\ (p \# ps) = (if\ fst\ p = k\ then\ delete\ k\ ps\ else\ p \# delete\ k\ ps)$   
 ⟨proof⟩

**lemma** *delete-conv'*:  $map-of\ (delete\ k\ al) = (map-of\ al)(k := None)$   
 ⟨proof⟩

**corollary** *delete-conv*:  $map-of\ (delete\ k\ al)\ k' = ((map-of\ al)(k := None))\ k'$   
 ⟨proof⟩

**lemma** *delete-keys*:  $map\ fst\ (delete\ k\ al) = removeAll\ k\ (map\ fst\ al)$   
 ⟨proof⟩

**lemma** *distinct-delete*:

**assumes** *distinct* (*map fst al*)  
**shows** *distinct* (*map fst (delete k al)*)  
 ⟨*proof*⟩

**lemma** *delete-id* [*simp*]:  $k \notin \text{fst } \text{'set } al \implies \text{delete } k \text{ } al = al$   
 ⟨*proof*⟩

**lemma** *delete-idem*:  $\text{delete } k \text{ } (\text{delete } k \text{ } al) = \text{delete } k \text{ } al$   
 ⟨*proof*⟩

**lemma** *map-of-delete* [*simp*]:  $k' \neq k \implies \text{map-of } (\text{delete } k \text{ } al) \text{ } k' = \text{map-of } al \text{ } k'$   
 ⟨*proof*⟩

**lemma** *delete-notin-dom*:  $k \notin \text{fst } \text{'set } (\text{delete } k \text{ } al)$   
 ⟨*proof*⟩

**lemma** *dom-delete-subset*:  $\text{fst } \text{'set } (\text{delete } k \text{ } al) \subseteq \text{fst } \text{'set } al$   
 ⟨*proof*⟩

**lemma** *delete-update-same*:  $\text{delete } k \text{ } (\text{update } k \text{ } v \text{ } al) = \text{delete } k \text{ } al$   
 ⟨*proof*⟩

**lemma** *delete-update*:  $k \neq l \implies \text{delete } l \text{ } (\text{update } k \text{ } v \text{ } al) = \text{update } k \text{ } v \text{ } (\text{delete } l \text{ } al)$   
 ⟨*proof*⟩

**lemma** *delete-twist*:  $\text{delete } x \text{ } (\text{delete } y \text{ } al) = \text{delete } y \text{ } (\text{delete } x \text{ } al)$   
 ⟨*proof*⟩

**lemma** *length-delete-le*:  $\text{length } (\text{delete } k \text{ } al) \leq \text{length } al$   
 ⟨*proof*⟩

### 1.3 *update-with-aux* and *delete-aux*

**qualified primrec** *update-with-aux* ::

$'val \Rightarrow 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$

**where**

$\text{update-with-aux } v \text{ } k \text{ } f \text{ } [] = [(k, f v)]$

$| \text{update-with-aux } v \text{ } k \text{ } f \text{ } (p \# ps) =$

$(\text{if } (\text{fst } p = k) \text{ then } (k, f (\text{snd } p)) \# ps \text{ else } p \# \text{update-with-aux } v \text{ } k \text{ } f \text{ } ps)$

The above *delete* traverses all the list even if it has found the key. This one does not have to keep going because it assumes the invariant that keys are distinct.

**qualified fun** *delete-aux* ::  $'key \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$

**where**

$\text{delete-aux } k \text{ } [] = []$

$| \text{delete-aux } k \text{ } ((k', v) \# xs) = (\text{if } k = k' \text{ then } xs \text{ else } (k', v) \# \text{delete-aux } k \text{ } xs)$



**lemma** *map-of-update-with-aux'*:

$$\begin{aligned} & \text{map-of } (\text{update-with-aux } v \ k \ f \ ps) \ k' = \\ & ((\text{map-of } ps)(k \mapsto (\text{case map-of } ps \ k \ \text{of } \text{None} \Rightarrow f \ v \mid \text{Some } v \Rightarrow f \ v))) \ k' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *map-of-update-with-aux*:

$$\begin{aligned} & \text{map-of } (\text{update-with-aux } v \ k \ f \ ps) = \\ & (\text{map-of } ps)(k \mapsto (\text{case map-of } ps \ k \ \text{of } \text{None} \Rightarrow f \ v \mid \text{Some } v \Rightarrow f \ v)) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *dom-update-with-aux*:  $\text{fst } \text{' set } (\text{update-with-aux } v \ k \ f \ ps) = \{k\} \cup \text{fst } \text{' set } ps$

$\langle \text{proof} \rangle$

**lemma** *distinct-update-with-aux [simp]*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } (\text{update-with-aux } v \ k \ f \ ps)) = \text{distinct } (\text{map } \text{fst } ps) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *set-update-with-aux*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \\ & \text{set } (\text{update-with-aux } v \ k \ f \ xs) = \\ & (\text{set } xs - \{k\} \times \text{UNIV} \cup \{(k, f (\text{case map-of } xs \ k \ \text{of } \text{None} \Rightarrow v \mid \text{Some } v \Rightarrow v))\}) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *set-delete-aux*:  $\text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \text{set } (\text{delete-aux } k \ xs) = \text{set } xs - \{k\} \times \text{UNIV}$

$\langle \text{proof} \rangle$

**lemma** *dom-delete-aux*:  $\text{distinct } (\text{map } \text{fst } ps) \Longrightarrow \text{fst } \text{' set } (\text{delete-aux } k \ ps) = \text{fst } \text{' set } ps - \{k\}$

$\langle \text{proof} \rangle$

**lemma** *distinct-delete-aux [simp]*:  $\text{distinct } (\text{map } \text{fst } ps) \Longrightarrow \text{distinct } (\text{map } \text{fst } (\text{delete-aux } k \ ps))$

$\langle \text{proof} \rangle$

**lemma** *map-of-delete-aux'*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \text{map-of } (\text{delete-aux } k \ xs) = (\text{map-of } xs)(k := \text{None}) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *map-of-delete-aux*:

$$\begin{aligned} & \text{distinct } (\text{map } \text{fst } xs) \Longrightarrow \text{map-of } (\text{delete-aux } k \ xs) \ k' = ((\text{map-of } xs)(k := \text{None})) \\ & \ k' \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *delete-aux-eq-Nil-conv*:  $\text{delete-aux } k \ ts = [] \longleftrightarrow ts = [] \vee (\exists v. ts = [(k, v)])$

⟨proof⟩

#### 1.4 restrict

**qualified definition**  $restrict :: 'key\ set \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$   
**where**  $restrict\text{-}eq: restrict\ A = filter\ (\lambda(k, v). k \in A)$

**lemma**  $restr\text{-}simps$  [simp]:

$restrict\ A\ [] = []$

$restrict\ A\ (p\#\ps) = (if\ fst\ p \in A\ then\ p\ \#\ restrict\ A\ ps\ else\ restrict\ A\ ps)$

⟨proof⟩

**lemma**  $restr\text{-}conv'$ :  $map\text{-}of\ (restrict\ A\ al) = ((map\text{-}of\ al)|'A)$

⟨proof⟩

**corollary**  $restr\text{-}conv$ :  $map\text{-}of\ (restrict\ A\ al)\ k = ((map\text{-}of\ al)|'A)\ k$

⟨proof⟩

**lemma**  $distinct\text{-}restr$ :  $distinct\ (map\ fst\ al) \Longrightarrow distinct\ (map\ fst\ (restrict\ A\ al))$

⟨proof⟩

**lemma**  $restr\text{-}empty$  [simp]:

$restrict\ \{\}\ al = []$

$restrict\ A\ [] = []$

⟨proof⟩

**lemma**  $restr\text{-}in$  [simp]:  $x \in A \Longrightarrow map\text{-}of\ (restrict\ A\ al)\ x = map\text{-}of\ al\ x$

⟨proof⟩

**lemma**  $restr\text{-}out$  [simp]:  $x \notin A \Longrightarrow map\text{-}of\ (restrict\ A\ al)\ x = None$

⟨proof⟩

**lemma**  $dom\text{-}restr$  [simp]:  $fst\ 'set\ (restrict\ A\ al) = fst\ 'set\ al \cap A$

⟨proof⟩

**lemma**  $restr\text{-}upd\text{-}same$  [simp]:  $restrict\ (-\{x\})\ (update\ x\ y\ al) = restrict\ (-\{x\})\ al$

⟨proof⟩

**lemma**  $restr\text{-}restr$  [simp]:  $restrict\ A\ (restrict\ B\ al) = restrict\ (A \cap B)\ al$

⟨proof⟩

**lemma**  $restr\text{-}update$ [simp]:

$map\text{-}of\ (restrict\ D\ (update\ x\ y\ al)) =$

$map\text{-}of\ ((if\ x \in D\ then\ (update\ x\ y\ (restrict\ (D - \{x\})\ al))\ else\ restrict\ D\ al))$

⟨proof⟩

**lemma**  $restr\text{-}delete$  [simp]:

$delete\ x\ (restrict\ D\ al) = (if\ x \in D\ then\ restrict\ (D - \{x\})\ al\ else\ restrict\ D\ al)$

⟨proof⟩

**lemma** *update-restr*:

$map\text{-of } (update\ x\ y\ (restrict\ D\ al)) = map\text{-of } (update\ x\ y\ (restrict\ (D - \{x\})\ al))$   
 ⟨proof⟩

**lemma** *update-restr-conv* [*simp*]:

$x \in D \implies$   
 $map\text{-of } (update\ x\ y\ (restrict\ D\ al)) = map\text{-of } (update\ x\ y\ (restrict\ (D - \{x\})\ al))$   
 ⟨proof⟩

**lemma** *restr-updates* [*simp*]:

$length\ xs = length\ ys \implies set\ xs \subseteq D \implies$   
 $map\text{-of } (restrict\ D\ (updates\ xs\ ys\ al)) =$   
 $map\text{-of } (updates\ xs\ ys\ (restrict\ (D - set\ xs)\ al))$   
 ⟨proof⟩

**lemma** *restr-delete-twist*:  $(restrict\ A\ (delete\ a\ ps)) = delete\ a\ (restrict\ A\ ps)$

⟨proof⟩

## 1.5 clearjunk

**qualified function** *clearjunk* ::  $('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$

**where**

$clearjunk\ [] = []$   
 $| clearjunk\ (p\#\ps) = p\ \#\ clearjunk\ (delete\ (fst\ p)\ ps)$   
 ⟨proof⟩

**termination**

⟨proof⟩

**lemma** *map-of-clearjunk*:  $map\text{-of } (clearjunk\ al) = map\text{-of } al$

⟨proof⟩

**lemma** *clearjunk-keys-set*:  $set\ (map\ fst\ (clearjunk\ al)) = set\ (map\ fst\ al)$

⟨proof⟩

**lemma** *dom-clearjunk*:  $fst\ ' set\ (clearjunk\ al) = fst\ ' set\ al$

⟨proof⟩

**lemma** *distinct-clearjunk* [*simp*]:  $distinct\ (map\ fst\ (clearjunk\ al))$

⟨proof⟩

**lemma** *ran-clearjunk*:  $ran\ (map\text{-of } (clearjunk\ al)) = ran\ (map\text{-of } al)$

⟨proof⟩

**lemma** *ran-map-of*:  $ran\ (map\text{-of } al) = snd\ ' set\ (clearjunk\ al)$

⟨proof⟩

**lemma** *graph-map-of*:  $Map.graph (map\text{-}of\ al) = set (clearjunk\ al)$   
 ⟨proof⟩

**lemma** *clearjunk-update*:  $clearjunk (update\ k\ v\ al) = update\ k\ v (clearjunk\ al)$   
 ⟨proof⟩

**lemma** *clearjunk-updates*:  $clearjunk (updates\ ks\ vs\ al) = updates\ ks\ vs (clearjunk\ al)$   
 ⟨proof⟩

**lemma** *clearjunk-delete*:  $clearjunk (delete\ x\ al) = delete\ x (clearjunk\ al)$   
 ⟨proof⟩

**lemma** *clearjunk-restrict*:  $clearjunk (restrict\ A\ al) = restrict\ A (clearjunk\ al)$   
 ⟨proof⟩

**lemma** *distinct-clearjunk-id* [simp]:  $distinct (map\ fst\ al) \implies clearjunk\ al = al$   
 ⟨proof⟩

**lemma** *clearjunk-idem*:  $clearjunk (clearjunk\ al) = clearjunk\ al$   
 ⟨proof⟩

**lemma** *length-clearjunk*:  $length (clearjunk\ al) \leq length\ al$   
 ⟨proof⟩

**lemma** *delete-map*:  
 assumes  $\bigwedge kv. fst (f\ kv) = fst\ kv$   
 shows  $delete\ k (map\ f\ ps) = map\ f (delete\ k\ ps)$   
 ⟨proof⟩

**lemma** *clearjunk-map*:  
 assumes  $\bigwedge kv. fst (f\ kv) = fst\ kv$   
 shows  $clearjunk (map\ f\ ps) = map\ f (clearjunk\ ps)$   
 ⟨proof⟩

## 1.6 map-ran

**definition** *map-ran* ::  $('key \Rightarrow 'val1 \Rightarrow 'val2) \Rightarrow ('key \times 'val1)\ list \Rightarrow ('key \times 'val2)\ list$   
 where  $map\text{-}ran\ f = map (\lambda(k, v). (k, f\ k\ v))$

**lemma** *map-ran-simps* [simp]:  
 $map\text{-}ran\ f\ [] = []$   
 $map\text{-}ran\ f ((k, v) \# ps) = (k, f\ k\ v) \# map\text{-}ran\ f\ ps$   
 ⟨proof⟩

**lemma** *map-ran-Cons-sel*:  $map\text{-}ran\ f (p \# ps) = (fst\ p, f (fst\ p) (snd\ p)) \# map\text{-}ran\ f\ ps$   
 ⟨proof⟩

**lemma** *length-map-ran[simp]*:  $\text{length } (\text{map-ran } f \text{ al}) = \text{length } \text{al}$   
 ⟨proof⟩

**lemma** *map-fst-map-ran[simp]*:  $\text{map } \text{fst } (\text{map-ran } f \text{ al}) = \text{map } \text{fst } \text{al}$   
 ⟨proof⟩

**lemma** *dom-map-ran*:  $\text{fst } \text{'set } (\text{map-ran } f \text{ al}) = \text{fst } \text{'set } \text{al}$   
 ⟨proof⟩

**lemma** *map-ran-conv*:  $\text{map-of } (\text{map-ran } f \text{ al}) \text{ k} = \text{map-option } (f \text{ k}) (\text{map-of } \text{al } \text{k})$   
 ⟨proof⟩

**lemma** *distinct-map-ran*:  $\text{distinct } (\text{map } \text{fst } \text{al}) \implies \text{distinct } (\text{map } \text{fst } (\text{map-ran } f \text{ al}))$   
 ⟨proof⟩

**lemma** *map-ran-filter*:  $\text{map-ran } f \text{ [p←ps. fst p ≠ a]} = \text{[p←map-ran } f \text{ ps. fst p ≠ a]}$   
 ⟨proof⟩

**lemma** *clearjunk-map-ran*:  $\text{clearjunk } (\text{map-ran } f \text{ al}) = \text{map-ran } f (\text{clearjunk } \text{al})$   
 ⟨proof⟩

## 1.7 merge

**qualified definition** *merge* ::  $(\text{'key} \times \text{'val}) \text{ list} \Rightarrow (\text{'key} \times \text{'val}) \text{ list} \Rightarrow (\text{'key} \times \text{'val}) \text{ list}$   
 where  $\text{merge } \text{qs } \text{ps} = \text{foldr } (\lambda(k, v). \text{update } k \text{ v}) \text{ ps } \text{qs}$

**lemma** *merge-simps [simp]*:  
 $\text{merge } \text{qs } [] = \text{qs}$   
 $\text{merge } \text{qs } (\text{p\#ps}) = \text{update } (\text{fst } \text{p}) (\text{snd } \text{p}) (\text{merge } \text{qs } \text{ps})$   
 ⟨proof⟩

**lemma** *merge-updates*:  $\text{merge } \text{qs } \text{ps} = \text{updates } (\text{rev } (\text{map } \text{fst } \text{ps})) (\text{rev } (\text{map } \text{snd } \text{ps})) \text{qs}$   
 ⟨proof⟩

**lemma** *dom-merge*:  $\text{fst } \text{'set } (\text{merge } \text{xs } \text{ys}) = \text{fst } \text{'set } \text{xs} \cup \text{fst } \text{'set } \text{ys}$   
 ⟨proof⟩

**lemma** *distinct-merge*:  $\text{distinct } (\text{map } \text{fst } \text{xs}) \implies \text{distinct } (\text{map } \text{fst } (\text{merge } \text{xs } \text{ys}))$   
 ⟨proof⟩

**lemma** *clearjunk-merge*:  $\text{clearjunk } (\text{merge } \text{xs } \text{ys}) = \text{merge } (\text{clearjunk } \text{xs}) \text{ys}$   
 ⟨proof⟩

**lemma** *merge-conv'*:  $\text{map-of } (\text{merge } \text{xs } \text{ys}) = \text{map-of } \text{xs} ++ \text{map-of } \text{ys}$

*<proof>*

**corollary** *merge-conv*:  $\text{map-of } (\text{merge } xs \ ys) \ k = (\text{map-of } xs \ ++ \ \text{map-of } ys) \ k$   
*<proof>*

**lemma** *merge-empty*:  $\text{map-of } (\text{merge } [] \ ys) = \text{map-of } ys$   
*<proof>*

**lemma** *merge-assoc* [*simp*]:  $\text{map-of } (\text{merge } m1 \ (\text{merge } m2 \ m3)) = \text{map-of } (\text{merge } (\text{merge } m1 \ m2) \ m3)$   
*<proof>*

**lemma** *merge-Some-iff*:  
 $\text{map-of } (\text{merge } m \ n) \ k = \text{Some } x \iff$   
 $\text{map-of } n \ k = \text{Some } x \vee \text{map-of } n \ k = \text{None} \wedge \text{map-of } m \ k = \text{Some } x$   
*<proof>*

**lemmas** *merge-SomeD* [*dest!*] = *merge-Some-iff* [*THEN iffD1*]

**lemma** *merge-find-right* [*simp*]:  $\text{map-of } n \ k = \text{Some } v \implies \text{map-of } (\text{merge } m \ n) \ k = \text{Some } v$   
*<proof>*

**lemma** *merge-None* [*iff*]:  $(\text{map-of } (\text{merge } m \ n) \ k = \text{None}) = (\text{map-of } n \ k = \text{None} \wedge \text{map-of } m \ k = \text{None})$   
*<proof>*

**lemma** *merge-upd* [*simp*]:  $\text{map-of } (\text{merge } m \ (\text{update } k \ v \ n)) = \text{map-of } (\text{update } k \ v \ (\text{merge } m \ n))$   
*<proof>*

**lemma** *merge-updatess* [*simp*]:  
 $\text{map-of } (\text{merge } m \ (\text{updates } xs \ ys \ n)) = \text{map-of } (\text{updates } xs \ ys \ (\text{merge } m \ n))$   
*<proof>*

**lemma** *merge-append*:  $\text{map-of } (xs \ @ \ ys) = \text{map-of } (\text{merge } ys \ xs)$   
*<proof>*

## 1.8 compose

**qualified function** *compose* ::  $('key \times 'a) \ \text{list} \Rightarrow ('a \times 'b) \ \text{list} \Rightarrow ('key \times 'b) \ \text{list}$   
**where**

$\text{compose } [] \ ys = []$   
 $|\ \text{compose } (x \ # \ xs) \ ys =$   
 $\quad (\text{case } \text{map-of } ys \ (\text{snd } x) \ \text{of}$   
 $\quad \quad \text{None} \Rightarrow \text{compose } (\text{delete } (\text{fst } x) \ xs) \ ys$   
 $\quad \quad \text{Some } v \Rightarrow (\text{fst } x, v) \ # \ \text{compose } xs \ ys)$   
*<proof>*

**termination**

⟨proof⟩

**lemma** *compose-first-None* [simp]:  $\text{map-of } xs \ k = \text{None} \implies \text{map-of } (\text{compose } xs \ ys) \ k = \text{None}$   
 ⟨proof⟩

**lemma** *compose-conv*:  $\text{map-of } (\text{compose } xs \ ys) \ k = (\text{map-of } ys \circ_m \text{map-of } xs) \ k$   
 ⟨proof⟩

**lemma** *compose-conv'*:  $\text{map-of } (\text{compose } xs \ ys) = (\text{map-of } ys \circ_m \text{map-of } xs)$   
 ⟨proof⟩

**lemma** *compose-first-Some* [simp]:  $\text{map-of } xs \ k = \text{Some } v \implies \text{map-of } (\text{compose } xs \ ys) \ k = \text{map-of } ys \ v$   
 ⟨proof⟩

**lemma** *dom-compose*:  $\text{fst } ' \text{ set } (\text{compose } xs \ ys) \subseteq \text{fst } ' \text{ set } xs$   
 ⟨proof⟩

**lemma** *distinct-compose*:  
**assumes** *distinct* (map fst xs)  
**shows** *distinct* (map fst (compose xs ys))  
 ⟨proof⟩

**lemma** *compose-delete-twist*:  $\text{compose } (\text{delete } k \ xs) \ ys = \text{delete } k \ (\text{compose } xs \ ys)$   
 ⟨proof⟩

**lemma** *compose-clearjunk*:  $\text{compose } xs \ (\text{clearjunk } ys) = \text{compose } xs \ ys$   
 ⟨proof⟩

**lemma** *clearjunk-compose*:  $\text{clearjunk } (\text{compose } xs \ ys) = \text{compose } (\text{clearjunk } xs) \ ys$   
 ⟨proof⟩

**lemma** *compose-empty* [simp]:  $\text{compose } xs \ [] = []$   
 ⟨proof⟩

**lemma** *compose-Some-iff*:  
 $(\text{map-of } (\text{compose } xs \ ys) \ k = \text{Some } v) \longleftrightarrow$   
 $(\exists k'. \text{map-of } xs \ k = \text{Some } k' \wedge \text{map-of } ys \ k' = \text{Some } v)$   
 ⟨proof⟩

**lemma** *map-comp-None-iff*:  
 $\text{map-of } (\text{compose } xs \ ys) \ k = \text{None} \longleftrightarrow$   
 $(\text{map-of } xs \ k = \text{None} \vee (\exists k'. \text{map-of } xs \ k = \text{Some } k' \wedge \text{map-of } ys \ k' = \text{None}))$   
 ⟨proof⟩

**1.9** *map-entry*

**qualified fun** *map-entry* :: 'key  $\Rightarrow$  ('val  $\Rightarrow$  'val)  $\Rightarrow$  ('key  $\times$  'val) list  $\Rightarrow$  ('key  $\times$  'val) list

**where**

*map-entry* k f [] = []  
 | *map-entry* k f (p # ps) =  
 (if fst p = k then (k, f (snd p)) # ps else p # *map-entry* k f ps)

**lemma** *map-of-map-entry*:

*map-of* (*map-entry* k f xs) =  
 (*map-of* xs)(k := case *map-of* xs k of None  $\Rightarrow$  None | Some v'  $\Rightarrow$  Some (f v'))  
 <proof>

**lemma** *dom-map-entry*: fst ' set (*map-entry* k f xs) = fst ' set xs

<proof>

**lemma** *distinct-map-entry*:

**assumes** *distinct* (map fst xs)  
**shows** *distinct* (map fst (*map-entry* k f xs))  
 <proof>

**1.10** *map-default*

**fun** *map-default* :: 'key  $\Rightarrow$  'val  $\Rightarrow$  ('val  $\Rightarrow$  'val)  $\Rightarrow$  ('key  $\times$  'val) list  $\Rightarrow$  ('key  $\times$  'val) list

**where**

*map-default* k v f [] = [(k, v)]  
 | *map-default* k v f (p # ps) =  
 (if fst p = k then (k, f (snd p)) # ps else p # *map-default* k v f ps)

**lemma** *map-of-map-default*:

*map-of* (*map-default* k v f xs) =  
 (*map-of* xs)(k := case *map-of* xs k of None  $\Rightarrow$  Some v | Some v'  $\Rightarrow$  Some (f v'))  
 <proof>

**lemma** *dom-map-default*: fst ' set (*map-default* k v f xs) = insert k (fst ' set xs)

<proof>

**lemma** *distinct-map-default*:

**assumes** *distinct* (map fst xs)  
**shows** *distinct* (map fst (*map-default* k v f xs))  
 <proof>

**end**

**end**



## 2 Axiomatic Declaration of Bounded Natural Functors

```

theory BNF-Axiomatization
imports Main
keywords
  bnf-axiomatization :: thy-decl
begin

⟨ML⟩

end

```

## 3 Generalized Corecursor Sugar (corec and friends)

```

theory BNF-Corec
imports Main
keywords
  corec :: thy-defn and
  corecursive :: thy-goal-defn and
  friend-of-corec :: thy-goal-defn and
  coinduction-upto :: thy-decl
begin

lemma obj-distinct-prems:  $P \longrightarrow P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$ 
  ⟨proof⟩

lemma inject-refine:  $g (f x) = x \Longrightarrow g (f y) = y \Longrightarrow f x = f y \longleftrightarrow x = y$ 
  ⟨proof⟩

lemma convol-apply: BNF-Def.convolf g x = (f x, g x)
  ⟨proof⟩

lemma Grp-UNIV-id: BNF-Def.Grp UNIV id = (=)
  ⟨proof⟩

lemma sum-comp-cases:
  assumes  $f \circ \text{Inl} = g \circ \text{Inl}$  and  $f \circ \text{Inr} = g \circ \text{Inr}$ 
  shows  $f = g$ 
  ⟨proof⟩

lemma case-sum-Inl-Inr-L: case-sum (f ∘ Inl) (f ∘ Inr) = f
  ⟨proof⟩

lemma eq-o-InrI:  $\llbracket g \circ \text{Inl} = h; \text{case-sum } h f = g \rrbracket \Longrightarrow f = g \circ \text{Inr}$ 
  ⟨proof⟩

lemma id-bnf-o: BNF-Composition.id-bnf ∘ f = f

```

*<proof>*

**lemma** *o-id-bnf*:  $f \circ \text{BNF-Composition.id-bnf} = f$   
*<proof>*

**lemma** *if-True-False*:

$(\text{if } P \text{ then True else } Q) \longleftrightarrow P \vee Q$   
 $(\text{if } P \text{ then False else } Q) \longleftrightarrow \neg P \wedge Q$   
 $(\text{if } P \text{ then } Q \text{ else True}) \longleftrightarrow \neg P \vee Q$   
 $(\text{if } P \text{ then } Q \text{ else False}) \longleftrightarrow P \wedge Q$   
*<proof>*

**lemma** *if-distrib-fun*:  $(\text{if } c \text{ then } f \text{ else } g) x = (\text{if } c \text{ then } f x \text{ else } g x)$   
*<proof>*

### 3.1 Coinduction

**lemma** *eq-comp-compI*:  $a \circ b = f \circ x \implies x \circ c = \text{id} \implies f = a \circ (b \circ c)$   
*<proof>*

**lemma** *self-bounded-weaken-left*:  $(a :: 'a :: \text{semilattice-inf}) \leq \text{inf } a b \implies a \leq b$   
*<proof>*

**lemma** *self-bounded-weaken-right*:  $(a :: 'a :: \text{semilattice-inf}) \leq \text{inf } b a \implies a \leq b$   
*<proof>*

**lemma** *symp-iff*:  $\text{symp } R \longleftrightarrow R = R^{-1-1}$   
*<proof>*

**lemma** *equivp-inf*:  $\llbracket \text{equivp } R; \text{equivp } S \rrbracket \implies \text{equivp } (\text{inf } R S)$   
*<proof>*

**lemma** *vimage2p-rel-prod*:

$(\lambda x y. \text{rel-prod } R S (\text{BNF-Def.convol } f1 \ g1 \ x) (\text{BNF-Def.convol } f2 \ g2 \ y)) =$   
 $(\text{inf } (\text{BNF-Def.vimage2p } f1 \ f2 \ R) (\text{BNF-Def.vimage2p } g1 \ g2 \ S))$   
*<proof>*

**lemma** *predicate2I-obj*:  $(\forall x y. P x y \longrightarrow Q x y) \implies P \leq Q$   
*<proof>*

**lemma** *predicate2D-obj*:  $P \leq Q \implies P x y \longrightarrow Q x y$   
*<proof>*

**locale** *cong* =

**fixes** *rel* ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'b \Rightarrow \text{bool})$   
**and** *eval* ::  $'b \Rightarrow 'a$   
**and** *retr* ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool})$   
**assumes** *rel-mono*:  $\bigwedge R S. R \leq S \implies \text{rel } R \leq \text{rel } S$   
**and** *equivp-retr*:  $\bigwedge R. \text{equivp } R \implies \text{equivp } (\text{retr } R)$

**and** *retr-eval*:  $\bigwedge R x y. \llbracket (\text{rel-fun } (\text{rel } R) R) \text{ eval eval}; \text{rel } (\text{inf } R (\text{retr } R)) x y \rrbracket$   
 $\implies$   
 $\text{retr } R (\text{eval } x) (\text{eval } y)$

**begin**

**definition** *cong* ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$  **where**  
 $\text{cong } R \equiv \text{equivp } R \wedge (\text{rel-fun } (\text{rel } R) R) \text{ eval eval}$

**lemma** *cong-retr*:  $\text{cong } R \implies \text{cong } (\text{inf } R (\text{retr } R))$   
 $\langle \text{proof} \rangle$

**lemma** *cong-equivp*:  $\text{cong } R \implies \text{equivp } R$   
 $\langle \text{proof} \rangle$

**definition** *gen-cong* ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$  **where**  
 $\text{gen-cong } R j1 j2 \equiv \forall R'. R \leq R' \wedge \text{cong } R' \longrightarrow R' j1 j2$

**lemma** *gen-cong-reflp*[*intro, simp*]:  $x = y \implies \text{gen-cong } R x y$   
 $\langle \text{proof} \rangle$

**lemma** *gen-cong-symp*[*intro*]:  $\text{gen-cong } R x y \implies \text{gen-cong } R y x$   
 $\langle \text{proof} \rangle$

**lemma** *gen-cong-transp*[*intro*]:  $\text{gen-cong } R x y \implies \text{gen-cong } R y z \implies \text{gen-cong } R x z$   
 $\langle \text{proof} \rangle$

**lemma** *equivp-gen-cong*:  $\text{equivp } (\text{gen-cong } R)$   
 $\langle \text{proof} \rangle$

**lemma** *leq-gen-cong*:  $R \leq \text{gen-cong } R$   
 $\langle \text{proof} \rangle$

**lemmas** *imp-gen-cong*[*intro*] = *predicate2D*[*OF leq-gen-cong*]

**lemma** *gen-cong-minimal*:  $\llbracket R \leq R'; \text{cong } R' \rrbracket \implies \text{gen-cong } R \leq R'$   
 $\langle \text{proof} \rangle$

**lemma** *congdd-base-gen-congdd-base-aux*:  
 $\text{rel } (\text{gen-cong } R) x y \implies R \leq R' \implies \text{cong } R' \implies R' (\text{eval } x) (\text{eval } y)$   
 $\langle \text{proof} \rangle$

**lemma** *cong-gen-cong*:  $\text{cong } (\text{gen-cong } R)$   
 $\langle \text{proof} \rangle$

**lemma** *gen-cong-eval-rel-fun*:  
 $(\text{rel-fun } (\text{rel } (\text{gen-cong } R)) (\text{gen-cong } R)) \text{ eval eval}$   
 $\langle \text{proof} \rangle$

**lemma** *gen-cong-eval*:

$rel (gen-cong R) x y \implies gen-cong R (eval x) (eval y)$   
 $\langle proof \rangle$

**lemma** *gen-cong-idem*:  $gen-cong (gen-cong R) = gen-cong R$

$\langle proof \rangle$

**lemma** *gen-cong-rho*:

$\varrho = eval \circ f \implies rel (gen-cong R) (f x) (f y) \implies gen-cong R (\varrho x) (\varrho y)$   
 $\langle proof \rangle$

**lemma** *coinduction*:

**assumes** *coind*:  $\forall R. R \leq retr R \longrightarrow R \leq (=)$

**assumes** *cih*:  $R \leq retr (gen-cong R)$

**shows**  $R \leq (=)$

$\langle proof \rangle$

**end**

**lemma** *rel-sum-case-sum*:

$rel-fun (rel-sum R S) T (case-sum f1 g1) (case-sum f2 g2) = (rel-fun R T f1 f2)$   
 $\wedge rel-fun S T g1 g2)$   
 $\langle proof \rangle$

**context**

**fixes** *rel eval rel' eval' retr emb*

**assumes** *base*:  $cong rel eval retr$

**and** *step*:  $cong rel' eval' retr$

**and** *emb*:  $eval' \circ emb = eval$

**and** *emb-transfer*:  $rel-fun (rel R) (rel' R) emb emb$

**begin**

**interpretation** *base*:  $cong rel eval retr \langle proof \rangle$

**interpretation** *step*:  $cong rel' eval' retr \langle proof \rangle$

**lemma** *gen-cong-emb*:  $base.gen-cong R \leq step.gen-cong R$

$\langle proof \rangle$

**end**

**named-theorems** *friend-of-corec-simps*

$\langle ML \rangle$

**end**

## 4 A general “while” combinator

**theory** *While-Combinator*

**imports** *Main*

begin

#### 4.1 Partial version

**definition** *while-option* :: ( $'a \Rightarrow \text{bool}$ )  $\Rightarrow$  ( $'a \Rightarrow 'a$ )  $\Rightarrow$   $'a \Rightarrow 'a$  option **where**  
*while-option*  $b\ c\ s = (\text{if } (\exists k. \neg b ((c \rightsquigarrow k)\ s))$   
     *then*  $\text{Some } ((c \rightsquigarrow (\text{LEAST } k. \neg b ((c \rightsquigarrow k)\ s)))\ s)$   
     *else*  $\text{None}$ )

**theorem** *while-option-unfold*[code]:  
*while-option*  $b\ c\ s = (\text{if } b\ s\ \text{then } \text{while-option } b\ c\ (c\ s)\ \text{else } \text{Some } s)$   
 ⟨proof⟩

**lemma** *while-option-stop2*:  
*while-option*  $b\ c\ s = \text{Some } t \Longrightarrow \exists k. t = (c \rightsquigarrow k)\ s \wedge \neg b\ t$   
 ⟨proof⟩

**lemma** *while-option-stop*: *while-option*  $b\ c\ s = \text{Some } t \Longrightarrow \neg b\ t$   
 ⟨proof⟩

**theorem** *while-option-rule*:  
**assumes** *step*:  $\bigwedge s. P\ s \Longrightarrow b\ s \Longrightarrow P\ (c\ s)$   
**and result**: *while-option*  $b\ c\ s = \text{Some } t$   
**and init**:  $P\ s$   
**shows**  $P\ t$   
 ⟨proof⟩

**lemma** *funpow-commute*:  
 $\llbracket \forall k' < k. f\ (c\ ((c \rightsquigarrow k')\ s)) = c'\ (f\ ((c \rightsquigarrow k')\ s)) \rrbracket \Longrightarrow f\ ((c \rightsquigarrow k)\ s) = (c' \rightsquigarrow k)\ (f\ s)$   
 ⟨proof⟩

**lemma** *while-option-commute-invariant*:  
**assumes** *Invariant*:  $\bigwedge s. P\ s \Longrightarrow b\ s \Longrightarrow P\ (c\ s)$   
**assumes** *TestCommute*:  $\bigwedge s. P\ s \Longrightarrow b\ s = b'\ (f\ s)$   
**assumes** *BodyCommute*:  $\bigwedge s. P\ s \Longrightarrow b\ s \Longrightarrow f\ (c\ s) = c'\ (f\ s)$   
**assumes** *Initial*:  $P\ s$   
**shows** *map-option*  $f\ (\text{while-option } b\ c\ s) = \text{while-option } b'\ c'\ (f\ s)$   
 ⟨proof⟩

**lemma** *while-option-commute*:  
**assumes**  $\bigwedge s. b\ s = b'\ (f\ s) \wedge s. \llbracket b\ s \rrbracket \Longrightarrow f\ (c\ s) = c'\ (f\ s)$   
**shows** *map-option*  $f\ (\text{while-option } b\ c\ s) = \text{while-option } b'\ c'\ (f\ s)$   
 ⟨proof⟩

#### 4.2 Total version

**definition** *while* :: ( $'a \Rightarrow \text{bool}$ )  $\Rightarrow$  ( $'a \Rightarrow 'a$ )  $\Rightarrow$   $'a \Rightarrow 'a$   
**where** *while*  $b\ c\ s = \text{the } (\text{while-option } b\ c\ s)$

**lemma** *while-unfold* [code]:

$while\ b\ c\ s = (if\ b\ s\ then\ while\ b\ c\ (c\ s)\ else\ s)$   
 ⟨proof⟩

**lemma** *def-while-unfold*:

**assumes** *fdef*:  $f == while\ test\ do$   
**shows**  $f\ x = (if\ test\ x\ then\ f(do\ x)\ else\ x)$   
 ⟨proof⟩

The proof rule for *while*, where  $P$  is the invariant.

**theorem** *while-rule-lemma*:

**assumes** *invariant*:  $\bigwedge s. P\ s \implies b\ s \implies P\ (c\ s)$   
**and** *terminate*:  $\bigwedge s. P\ s \implies \neg b\ s \implies Q\ s$   
**and** *wf*:  $wf\ \{(t, s). P\ s \wedge b\ s \wedge t = c\ s\}$   
**shows**  $P\ s \implies Q\ (while\ b\ c\ s)$   
 ⟨proof⟩

**theorem** *while-rule*:

$\llbracket P\ s;$   
 $\bigwedge s. \llbracket P\ s; b\ s \rrbracket \implies P\ (c\ s);$   
 $\bigwedge s. \llbracket P\ s; \neg b\ s \rrbracket \implies Q\ s;$   
 $wf\ r;$   
 $\bigwedge s. \llbracket P\ s; b\ s \rrbracket \implies (c\ s, s) \in r \rrbracket \implies$   
 $Q\ (while\ b\ c\ s)$   
 ⟨proof⟩

Combine invariant preservation and variant decrease in one goal:

**theorem** *while-rule2*:

$\llbracket P\ s;$   
 $\bigwedge s. \llbracket P\ s; b\ s \rrbracket \implies P\ (c\ s) \wedge (c\ s, s) \in r;$   
 $\bigwedge s. \llbracket P\ s; \neg b\ s \rrbracket \implies Q\ s;$   
 $wf\ r \rrbracket \implies$   
 $Q\ (while\ b\ c\ s)$   
 ⟨proof⟩

Proving termination:

**theorem** *wf-while-option-Some*:

**assumes** *wf*  $\{(t, s). (P\ s \wedge b\ s) \wedge t = c\ s\}$   
**and**  $\bigwedge s. P\ s \implies b\ s \implies P(c\ s)$  **and**  $P\ s$   
**shows**  $\exists t. while\ option\ b\ c\ s = Some\ t$   
 ⟨proof⟩

**lemma** *wf-rel-while-option-Some*:

**assumes** *wf*:  $wf\ R$   
**assumes** *smaller*:  $\bigwedge s. P\ s \wedge b\ s \implies (c\ s, s) \in R$   
**assumes** *inv*:  $\bigwedge s. P\ s \wedge b\ s \implies P(c\ s)$   
**assumes** *init*:  $P\ s$   
**shows**  $\exists t. while\ option\ b\ c\ s = Some\ t$   
 ⟨proof⟩

**theorem** *measure-while-option-Some*: **fixes**  $f :: 's \Rightarrow \text{nat}$   
**shows**  $(\bigwedge s. P s \Longrightarrow b s \Longrightarrow P(c s) \wedge f(c s) < f s)$   
 $\Longrightarrow P s \Longrightarrow \exists t. \text{while-option } b c s = \text{Some } t$   
 $\langle \text{proof} \rangle$

Kleene iteration starting from the empty set and assuming some finite bounding set:

**lemma** *while-option-finite-subset-Some*: **fixes**  $C :: 'a \text{ set}$   
**assumes** *mono f* **and**  $\bigwedge X. X \subseteq C \Longrightarrow f X \subseteq C$  **and** *finite C*  
**shows**  $\exists P. \text{while-option } (\lambda A. f A \neq A) f \{\} = \text{Some } P$   
 $\langle \text{proof} \rangle$

**lemma** *lfp-the-while-option*:  
**assumes** *mono f* **and**  $\bigwedge X. X \subseteq C \Longrightarrow f X \subseteq C$  **and** *finite C*  
**shows**  $\text{lfp } f = \text{the}(\text{while-option } (\lambda A. f A \neq A) f \{\})$   
 $\langle \text{proof} \rangle$

**lemma** *lfp-while*:  
**assumes** *mono f* **and**  $\bigwedge X. X \subseteq C \Longrightarrow f X \subseteq C$  **and** *finite C*  
**shows**  $\text{lfp } f = \text{while } (\lambda A. f A \neq A) f \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *wf-finite-less*:  
**assumes** *finite (C :: 'a::order set)*  
**shows**  $\text{wf } \{(x, y). \{x, y\} \subseteq C \wedge x < y\}$   
 $\langle \text{proof} \rangle$

**lemma** *wf-finite-greater*:  
**assumes** *finite (C :: 'a::order set)*  
**shows**  $\text{wf } \{(x, y). \{x, y\} \subseteq C \wedge y < x\}$   
 $\langle \text{proof} \rangle$

**lemma** *while-option-finite-increasing-Some*:  
**fixes**  $f :: 'a::\text{order} \Rightarrow 'a$   
**assumes** *mono f* **and** *finite (UNIV :: 'a set)* **and**  $s \leq f s$   
**shows**  $\exists P. \text{while-option } (\lambda A. f A \neq A) f s = \text{Some } P$   
 $\langle \text{proof} \rangle$

**lemma** *lfp-the-while-option-lattice*:  
**fixes**  $f :: 'a::\text{complete-lattice} \Rightarrow 'a$   
**assumes** *mono f* **and** *finite (UNIV :: 'a set)*  
**shows**  $\text{lfp } f = \text{the}(\text{while-option } (\lambda A. f A \neq A) f \text{bot})$   
 $\langle \text{proof} \rangle$

**lemma** *lfp-while-lattice*:  
**fixes**  $f :: 'a::\text{complete-lattice} \Rightarrow 'a$   
**assumes** *mono f* **and** *finite (UNIV :: 'a set)*  
**shows**  $\text{lfp } f = \text{while } (\lambda A. f A \neq A) f \text{bot}$   
 $\langle \text{proof} \rangle$

**lemma** *while-option-finite-decreasing-Some*:

**fixes**  $f :: 'a::order \Rightarrow 'a$   
**assumes** *mono f and finite (UNIV :: 'a set) and  $f s \leq s$*   
**shows**  $\exists P. \text{while-option } (\lambda A. f A \neq A) f s = \text{Some } P$   
*<proof>*

**lemma** *gfp-the-while-option-lattice*:

**fixes**  $f :: 'a::complete-lattice \Rightarrow 'a$   
**assumes** *mono f and finite (UNIV :: 'a set)*  
**shows**  $\text{gfp } f = \text{the}(\text{while-option } (\lambda A. f A \neq A) f \text{ top})$   
*<proof>*

**lemma** *gfp-while-lattice*:

**fixes**  $f :: 'a::complete-lattice \Rightarrow 'a$   
**assumes** *mono f and finite (UNIV :: 'a set)*  
**shows**  $\text{gfp } f = \text{while } (\lambda A. f A \neq A) f \text{ top}$   
*<proof>*

Computing the reflexive, transitive closure by iterating a successor function. Stops when an element is found that does not satisfy the test.

More refined (and hence more efficient) versions can be found in ITP 2011 paper by Nipkow (the theories are in the AFP entry *Flyspeck* by Nipkow) and the AFP article *Executable Transitive Closures* by René Thiemann.

**context**

**fixes**  $p :: 'a \Rightarrow \text{bool}$   
**and**  $f :: 'a \Rightarrow 'a \text{ list}$   
**and**  $x :: 'a$   
**begin**

**qualified fun** *rtrancl-while-test* ::  $'a \text{ list} \times 'a \text{ set} \Rightarrow \text{bool}$   
**where** *rtrancl-while-test*  $(ws, -) = (ws \neq [] \wedge p(\text{hd } ws))$

**qualified fun** *rtrancl-while-step* ::  $'a \text{ list} \times 'a \text{ set} \Rightarrow 'a \text{ list} \times 'a \text{ set}$   
**where** *rtrancl-while-step*  $(ws, Z) =$   
 $(\text{let } x = \text{hd } ws; \text{new} = \text{remdups } (\text{filter } (\lambda y. y \notin Z) (f x))$   
 $\text{in } (\text{new} @ \text{tl } ws, \text{set new} \cup Z))$

**definition** *rtrancl-while* ::  $('a \text{ list} * 'a \text{ set}) \text{ option}$

**where** *rtrancl-while* = *while-option rtrancl-while-test rtrancl-while-step*  $([x], \{x\})$

**qualified fun** *rtrancl-while-invariant* ::  $'a \text{ list} \times 'a \text{ set} \Rightarrow \text{bool}$

**where** *rtrancl-while-invariant*  $(ws, Z) =$   
 $(x \in Z \wedge \text{set } ws \subseteq Z \wedge \text{distinct } ws \wedge \{(x, y). y \in \text{set}(f x)\} \text{ “ } (Z - \text{set } ws) \subseteq Z$   
 $\wedge$   
 $Z \subseteq \{(x, y). y \in \text{set}(f x)\}^* \text{ “ } \{x\} \wedge (\forall z \in Z - \text{set } ws. p z))$

**qualified lemma** *rtrancl-while-invariant*:

**assumes** *inv: rtrancl-while-invariant st and test: rtrancl-while-test st*



**shows** *rtrancl-while-invariant* (*rtrancl-while-step st*)  
 ⟨*proof*⟩

**lemma** *rtrancl-while-Some*:

**assumes** *rtrancl-while* = *Some(ws,Z)*

**shows** if *ws* = []

then  $Z = \{(x,y). y \in \text{set}(f\ x)\}^* \text{ “ } \{x\} \wedge (\forall z \in Z. p\ z)$

else  $\neg p(\text{hd}\ ws) \wedge \text{hd}\ ws \in \{(x,y). y \in \text{set}(f\ x)\}^* \text{ “ } \{x\}$

⟨*proof*⟩

**lemma** *rtrancl-while-finite-Some*:

**assumes** *finite* ( $\{(x,y). y \in \text{set}(f\ x)\}^* \text{ “ } \{x\}$ ) (**is finite ?Cl**)

**shows**  $\exists y. \text{rtrancl-while} = \text{Some}\ y$

⟨*proof*⟩

**end**

**end**

## 5 The Bourbaki-Witt tower construction for trans-finite iteration

**theory** *Bourbaki-Witt-Fixpoint*

**imports** *While-Combinator*

**begin**

**lemma** *ChainsI* [*intro?*]:

$(\bigwedge a\ b. \llbracket a \in Y; b \in Y \rrbracket \implies (a, b) \in r \vee (b, a) \in r) \implies Y \in \text{Chains}\ r$

⟨*proof*⟩

**lemma** *in-Chains-subset*:  $\llbracket M \in \text{Chains}\ r; M' \subseteq M \rrbracket \implies M' \in \text{Chains}\ r$

⟨*proof*⟩

**lemma** *in-ChainsD*:  $\llbracket M \in \text{Chains}\ r; x \in M; y \in M \rrbracket \implies (x, y) \in r \vee (y, x) \in r$

⟨*proof*⟩

**lemma** *Chains-FieldD*:  $\llbracket M \in \text{Chains}\ r; x \in M \rrbracket \implies x \in \text{Field}\ r$

⟨*proof*⟩

**lemma** *in-Chains-conv-chain*:  $M \in \text{Chains}\ r \longleftrightarrow \text{Complete-Partial-Order.chain}$

$(\lambda x\ y. (x, y) \in r)\ M$

⟨*proof*⟩

**lemma** *partial-order-on-trans*:

$\llbracket \text{partial-order-on}\ A\ r; (x, y) \in r; (y, z) \in r \rrbracket \implies (x, z) \in r$

⟨*proof*⟩

**locale** *bourbaki-witt-fixpoint* =

**fixes**  $lub :: 'a \text{ set} \Rightarrow 'a$   
**and**  $leq :: ('a \times 'a) \text{ set}$   
**and**  $f :: 'a \Rightarrow 'a$   
**assumes**  $po$ : *Partial-order leq*  
**and**  $lub\text{-least}$ :  $\llbracket M \in \text{Chains } leq; M \neq \{\}; \bigwedge x. x \in M \Longrightarrow (x, z) \in leq \rrbracket \Longrightarrow (lub\ M, z) \in leq$   
**and**  $lub\text{-upper}$ :  $\llbracket M \in \text{Chains } leq; x \in M \rrbracket \Longrightarrow (x, lub\ M) \in leq$   
**and**  $lub\text{-in-Field}$ :  $\llbracket M \in \text{Chains } leq; M \neq \{\} \rrbracket \Longrightarrow lub\ M \in \text{Field } leq$   
**and**  $increasing$ :  $\bigwedge x. x \in \text{Field } leq \Longrightarrow (x, f\ x) \in leq$   
**begin**

**lemma**  $leq\text{-trans}$ :  $\llbracket (x, y) \in leq; (y, z) \in leq \rrbracket \Longrightarrow (x, z) \in leq$   
 $\langle proof \rangle$

**lemma**  $leq\text{-refl}$ :  $x \in \text{Field } leq \Longrightarrow (x, x) \in leq$   
 $\langle proof \rangle$

**lemma**  $leq\text{-antisym}$ :  $\llbracket (x, y) \in leq; (y, x) \in leq \rrbracket \Longrightarrow x = y$   
 $\langle proof \rangle$

**inductive-set**  $iterates\text{-above} :: 'a \Rightarrow 'a \text{ set}$

**for**  $a$

**where**

$base$ :  $a \in iterates\text{-above } a$

$| step$ :  $x \in iterates\text{-above } a \Longrightarrow f\ x \in iterates\text{-above } a$

$| Sup$ :  $\llbracket M \in \text{Chains } leq; M \neq \{\}; \bigwedge x. x \in M \Longrightarrow x \in iterates\text{-above } a \rrbracket \Longrightarrow lub\ M \in iterates\text{-above } a$

**definition**  $fixp\text{-above} :: 'a \Rightarrow 'a$

**where**  $fixp\text{-above } a = (if\ a \in \text{Field } leq\ \text{then } lub\ (iterates\text{-above } a)\ \text{else } a)$

**lemma**  $fixp\text{-above-outside}$ :  $a \notin \text{Field } leq \Longrightarrow fixp\text{-above } a = a$   
 $\langle proof \rangle$

**lemma**  $fixp\text{-above-inside}$ :  $a \in \text{Field } leq \Longrightarrow fixp\text{-above } a = lub\ (iterates\text{-above } a)$   
 $\langle proof \rangle$

**context**

**notes**  $leq\text{-refl}$  [*intro!*, *simp*]

**and**  $base$  [*intro*]

**and**  $step$  [*intro*]

**and**  $Sup$  [*intro*]

**and**  $leq\text{-trans}$  [*trans*]

**begin**

**lemma**  $iterates\text{-above-le-f}$ :  $\llbracket x \in iterates\text{-above } a; a \in \text{Field } leq \rrbracket \Longrightarrow (x, f\ x) \in leq$   
 $\langle proof \rangle$

**lemma**  $iterates\text{-above-Field}$ :  $\llbracket x \in iterates\text{-above } a; a \in \text{Field } leq \rrbracket \Longrightarrow x \in \text{Field } leq$

*leq*  
 ⟨*proof*⟩

**lemma** *iterates-above-ge*:  
 assumes *y*:  $y \in \text{iterates-above } a$   
 and *a*:  $a \in \text{Field } leq$   
 shows  $(a, y) \in leq$   
 ⟨*proof*⟩

**lemma** *iterates-above-lub*:  
 assumes *M*:  $M \in \text{Chains } leq$   
 and *nempty*:  $M \neq \{\}$   
 and *upper*:  $\bigwedge y. y \in M \implies \exists z \in M. (y, z) \in leq \wedge z \in \text{iterates-above } a$   
 shows  $\text{lub } M \in \text{iterates-above } a$   
 ⟨*proof*⟩

**lemma** *iterates-above-successor*:  
 assumes *y*:  $y \in \text{iterates-above } a$   
 and *a*:  $a \in \text{Field } leq$   
 shows  $y = a \vee y \in \text{iterates-above } (f a)$   
 ⟨*proof*⟩

**lemma** *iterates-above-Sup-aux*:  
 assumes *M*:  $M \in \text{Chains } leq$   $M \neq \{\}$   
 and *M'*:  $M' \in \text{Chains } leq$   $M' \neq \{\}$   
 and *comp*:  $\bigwedge x. x \in M \implies x \in \text{iterates-above } (\text{lub } M') \vee \text{lub } M' \in \text{iterates-above } x$   
 shows  $(\text{lub } M, \text{lub } M') \in leq \vee \text{lub } M \in \text{iterates-above } (\text{lub } M')$   
 ⟨*proof*⟩

**lemma** *iterates-above-triangle*:  
 assumes *x*:  $x \in \text{iterates-above } a$   
 and *y*:  $y \in \text{iterates-above } a$   
 and *a*:  $a \in \text{Field } leq$   
 shows  $x \in \text{iterates-above } (y \vee a) \vee y \in \text{iterates-above } x$   
 ⟨*proof*⟩

**lemma** *chain-iterates-above*:  
 assumes *a*:  $a \in \text{Field } leq$   
 shows  $\text{iterates-above } a \in \text{Chains } leq$  (is ?C ∈ -)  
 ⟨*proof*⟩

**lemma** *fixp-iterates-above*:  $\text{fixp-above } a \in \text{iterates-above } a$   
 ⟨*proof*⟩

**lemma** *fixp-above-Field*:  $a \in \text{Field } leq \implies \text{fixp-above } a \in \text{Field } leq$   
 ⟨*proof*⟩

**lemma** *fixp-above-unfold*:

**assumes**  $a: a \in \text{Field } \text{leq}$   
**shows**  $\text{fixp-above } a = f (\text{fixp-above } a)$  (**is**  $?a = f ?a$ )  
 $\langle \text{proof} \rangle$

**end**

**lemma**  $\text{fixp-above-induct}$  [*case-names adm base step*]:  
**assumes**  $\text{adm}: \text{ccpo.admissible } \text{lub } (\lambda x y. (x, y) \in \text{leq}) P$   
**and**  $\text{base}: P a$   
**and**  $\text{step}: \bigwedge x. P x \implies P (f x)$   
**shows**  $P (\text{fixp-above } a)$   
 $\langle \text{proof} \rangle$

**end**

## 5.1 Connect with the while combinator for executability on chain-finite lattices.

**context**  $\text{bourbaki-witt-fixpoint}$  **begin**

**lemma**  $\text{in-Chains-finite}$ : — Translation from  $\llbracket \text{Complete-Partial-Order.chain } (\leq) ?A; \text{finite } ?A; ?A \neq \{\} \rrbracket \implies \text{Sup } ?A \in ?A$ .  
**assumes**  $M \in \text{Chains } \text{leq}$   
**and**  $M \neq \{\}$   
**and**  $\text{finite } M$   
**shows**  $\text{lub } M \in M$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{fun-pow-iterates-above}$ :  $(f \rightsquigarrow k) a \in \text{iterates-above } a$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{chfin-iterates-above-fun-pow}$ :  
**assumes**  $x \in \text{iterates-above } a$   
**assumes**  $\forall M \in \text{Chains } \text{leq}. \text{finite } M$   
**shows**  $\exists j. x = (f \rightsquigarrow j) a$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{Chain-finite-iterates-above-fun-pow-iff}$ :  
**assumes**  $\forall M \in \text{Chains } \text{leq}. \text{finite } M$   
**shows**  $x \in \text{iterates-above } a \iff (\exists j. x = (f \rightsquigarrow j) a)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{fixp-above-Kleene-iter-ex}$ :  
**assumes**  $(\forall M \in \text{Chains } \text{leq}. \text{finite } M)$   
**obtains**  $k$  **where**  $\text{fixp-above } a = (f \rightsquigarrow k) a$   
 $\langle \text{proof} \rangle$

**context**  $\text{fixes } a$  **assumes**  $a: a \in \text{Field } \text{leq}$  **begin**

**lemma** *funpow-Field-leq*:  $(f \text{ ^^ } k) a \in \text{Field leq}$   
 $\langle \text{proof} \rangle$

**lemma** *funpow-prefix*:  $j < k \implies ((f \text{ ^^ } j) a, (f \text{ ^^ } k) a) \in \text{leq}$   
 $\langle \text{proof} \rangle$

**lemma** *funpow-suffix*:  $(f \text{ ^^ } \text{Suc } k) a = (f \text{ ^^ } k) a \implies ((f \text{ ^^ } (j + k)) a, (f \text{ ^^ } k) a) \in \text{leq}$   
 $\langle \text{proof} \rangle$

**lemma** *funpow-stability*:  $(f \text{ ^^ } \text{Suc } k) a = (f \text{ ^^ } k) a \implies ((f \text{ ^^ } j) a, (f \text{ ^^ } k) a) \in \text{leq}$   
 $\langle \text{proof} \rangle$

**lemma** *funpow-in-Chains*:  $\{(f \text{ ^^ } k) a \mid k. \text{True}\} \in \text{Chains leq}$   
 $\langle \text{proof} \rangle$

**lemma** *fixp-above-Kleene-iter*:

**assumes**  $\forall M \in \text{Chains leq. finite } M$  — convenient but surely not necessary

**assumes**  $(f \text{ ^^ } \text{Suc } k) a = (f \text{ ^^ } k) a$

**shows**  $\text{fixp-above } a = (f \text{ ^^ } k) a$

$\langle \text{proof} \rangle$

**context** **assumes** *chfn*:  $\forall M \in \text{Chains leq. finite } M$  **begin**

**lemma** *Chain-finite-wf*:  $\text{wf } \{(f \text{ ^^ } k) a, (f \text{ ^^ } k) a \mid k. f \text{ ^^ } k) a \neq (f \text{ ^^ } k) a\}$   
 $\langle \text{proof} \rangle$

**lemma** *while-option-finite-increasing*:  $\exists P. \text{while-option } (\lambda A. f A \neq A) f a = \text{Some } P$   
 $\langle \text{proof} \rangle$

**lemma** *fixp-above-the-while-option*:  $\text{fixp-above } a = \text{the } (\text{while-option } (\lambda A. f A \neq A) f a)$   
 $\langle \text{proof} \rangle$

**lemma** *fixp-above-conv-while*:  $\text{fixp-above } a = \text{while } (\lambda A. f A \neq A) f a$   
 $\langle \text{proof} \rangle$

**end**

**end**

**end**

**lemma** *bourbaki-witt-fixpoint-complete-latticeI*:

**fixes**  $f :: 'a :: \text{complete-lattice} \Rightarrow 'a$

**assumes**  $\bigwedge x. x \leq f x$

**shows** *bourbaki-witt-fixpoint*  $\text{Sup } \{(x, y). x \leq y\} f$   
 ⟨proof⟩

**end**

## 6 Division with modulus centered towards zero.

**theory** *Centered-Division*

**imports** *Main*

**begin**

**lemma** *off-iff-abs-mod-2-eq-one*:

⟨ $\text{odd } l \iff |l| \bmod 2 = 1$ ⟩ **for**  $l :: \text{int}$

⟨proof⟩

The following specification of division on integers centers the modulus around zero. This is useful e.g. to define division on Gauss numbers. N.b.: This is not mentioned [2].

**definition** *centered-divide* ::  $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$  (**infixl**  $\langle \text{cdiv} \rangle$  70)

**where**  $\langle k \text{ cdiv } l = \text{sgn } l * ((k + |l| \text{ div } 2) \text{ div } |l|) \rangle$

**definition** *centered-modulo* ::  $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$  (**infixl**  $\langle \text{cmod} \rangle$  70)

**where**  $\langle k \text{ cmod } l = (k + |l| \text{ div } 2) \bmod |l| - |l| \text{ div } 2 \rangle$

Example:  $k \text{ cmod } 5 \in \{-2, -1, 0, 1, 2\}$

**lemma** *signed-take-bit-eq-cmod*:

⟨ $\text{signed-take-bit } n \ k = k \text{ cmod } (2 \wedge \text{Suc } n)$ ⟩

⟨proof⟩

Property  $\text{signed-take-bit } n \ k = k \text{ cmod } 2^{\text{Suc } n}$  is the key to generalize centered division to arbitrary structures satisfying *ring-bit-operations*, but so far it is not clear what practical relevance that would have.

**lemma** *cdiv-mult-cmod-eq*:

⟨ $k \text{ cdiv } l * l + k \text{ cmod } l = k$ ⟩

⟨proof⟩

**lemma** *mult-cdiv-cmod-eq*:

⟨ $l * (k \text{ cdiv } l) + k \text{ cmod } l = k$ ⟩

⟨proof⟩

**lemma** *cmod-cdiv-mult-eq*:

⟨ $k \text{ cmod } l + k \text{ cdiv } l * l = k$ ⟩

⟨proof⟩

**lemma** *cmod-mult-cdiv-eq*:

⟨ $k \text{ cmod } l + l * (k \text{ cdiv } l) = k$ ⟩

⟨proof⟩

**lemma** *minus-cdiv-mult-eq-cmod*:

$$\langle k - k \text{ cdiv } l * l = k \text{ cmod } l \rangle$$

*<proof>*

**lemma** *minus-mult-cdiv-eq-cmod*:

$$\langle k - l * (k \text{ cdiv } l) = k \text{ cmod } l \rangle$$

*<proof>*

**lemma** *minus-cmod-eq-cdiv-mult*:

$$\langle k - k \text{ cmod } l = k \text{ cdiv } l * l \rangle$$

*<proof>*

**lemma** *minus-cmod-eq-mult-cdiv*:

$$\langle k - k \text{ cmod } l = l * (k \text{ cdiv } l) \rangle$$

*<proof>*

**lemma** *cdiv-0-eq [simp]*:

$$\langle k \text{ cdiv } 0 = 0 \rangle$$

*<proof>*

**lemma** *cmod-0-eq [simp]*:

$$\langle k \text{ cmod } 0 = k \rangle$$

*<proof>*

**lemma** *cdiv-1-eq [simp]*:

$$\langle k \text{ cdiv } 1 = k \rangle$$

*<proof>*

**lemma** *cmod-1-eq [simp]*:

$$\langle k \text{ cmod } 1 = 0 \rangle$$

*<proof>*

**lemma** *zero-cdiv-eq [simp]*:

$$\langle 0 \text{ cdiv } k = 0 \rangle$$

*<proof>*

**lemma** *zero-cmod-eq [simp]*:

$$\langle 0 \text{ cmod } k = 0 \rangle$$

*<proof>*

**lemma** *cdiv-minus-eq*:

$$\langle k \text{ cdiv } - l = - (k \text{ cdiv } l) \rangle$$

*<proof>*

**lemma** *cmod-minus-eq [simp]*:

$$\langle k \text{ cmod } - l = k \text{ cmod } l \rangle$$

*<proof>*

**lemma** *cdiv-abs-eq*:

$\langle k \text{ cdiv } |l| = \text{sgn } l * (k \text{ cdiv } l) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *cmod-abs-eq* [*simp*]:  
 $\langle k \text{ cmod } |l| = k \text{ cmod } l \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *nonzero-mult-cdiv-cancel-right*:  
 $\langle k * l \text{ cdiv } l = k \rangle$  **if**  $\langle l \neq 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *cdiv-self-eq* [*simp*]:  
 $\langle k \text{ cdiv } k = 1 \rangle$  **if**  $\langle k \neq 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *cmod-self-eq* [*simp*]:  
 $\langle k \text{ cmod } k = 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *cmod-less-divisor*:  
 $\langle k \text{ cmod } l < |l| - |l| \text{ div } 2 \rangle$  **if**  $\langle l \neq 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *cmod-less-equal-divisor*:  
 $\langle k \text{ cmod } l \leq |l| \text{ div } 2 \rangle$  **if**  $\langle l \neq 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *divisor-less-equal-cmod'*:  
 $\langle |l| \text{ div } 2 - |l| \leq k \text{ cmod } l \rangle$  **if**  $\langle l \neq 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *divisor-less-equal-cmod*:  
 $\langle -(|l| \text{ div } 2) \leq k \text{ cmod } l \rangle$  **if**  $\langle l \neq 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *abs-cmod-less-equal*:  
 $\langle |k \text{ cmod } l| \leq |l| \text{ div } 2 \rangle$  **if**  $\langle l \neq 0 \rangle$   
 $\langle \text{proof} \rangle$

**end**

## 7 Order on characters

**theory** *Char-ord*  
**imports** *Main*  
**begin**

**instantiation** *char* :: *linorder*  
**begin**



**definition** *less-eq-char* ::  $\langle \text{char} \Rightarrow \text{char} \Rightarrow \text{bool} \rangle$   
**where**  $\langle c1 \leq c2 \iff \text{of-char } c1 \leq (\text{of-char } c2 :: \text{nat}) \rangle$

**definition** *less-char* ::  $\langle \text{char} \Rightarrow \text{char} \Rightarrow \text{bool} \rangle$   
**where**  $\langle c1 < c2 \iff \text{of-char } c1 < (\text{of-char } c2 :: \text{nat}) \rangle$

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

**end**

**lemma** *less-eq-char-simp* [*simp*, *code*]:  
 $\langle \text{Char } b0\ b1\ b2\ b3\ b4\ b5\ b6\ b7 \leq \text{Char } c0\ c1\ c2\ c3\ c4\ c5\ c6\ c7$   
 $\iff \text{lexordp-eq } [b7, b6, b5, b4, b3, b2, b1, b0] [c7, c6, c5, c4, c3, c2, c1, c0] \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *less-char-simp* [*simp*, *code*]:  
 $\langle \text{Char } b0\ b1\ b2\ b3\ b4\ b5\ b6\ b7 < \text{Char } c0\ c1\ c2\ c3\ c4\ c5\ c6\ c7$   
 $\iff \text{ord-class.lexordp } [b7, b6, b5, b4, b3, b2, b1, b0] [c7, c6, c5, c4, c3, c2,$   
 $c1, c0] \rangle$   
 $\langle \text{proof} \rangle$

**instantiation** *char* :: *distrib-lattice*  
**begin**

**definition**  $\langle (\text{inf} :: \text{char} \Rightarrow -) = \text{min} \rangle$

**definition**  $\langle (\text{sup} :: \text{char} \Rightarrow -) = \text{max} \rangle$

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

**end**

**code-identifier**

**code-module** *Char-ord*  $\rightarrow$   
 $(\text{SML})\ \text{Str}\ \mathbf{and}\ (\text{OCaml})\ \text{Str}\ \mathbf{and}\ (\text{Haskell})\ \text{Str}\ \mathbf{and}\ (\text{Scala})\ \text{Str}$

**end**

## 8 A generic phantom type

**theory** *Phantom-Type*

**imports** *Main*

**begin**

**datatype**  $\langle 'a, 'b \rangle\ \text{phantom} = \text{phantom}\ (\text{of-phantom}: 'b)$

**lemma** *type-definition-phantom'*: *type-definition of-phantom phantom UNIV*  
 ⟨*proof*⟩

**lemma** *phantom-comp-of-phantom* [*simp*]: *phantom* ∘ *of-phantom* = *id*  
**and** *of-phantom-comp-phantom* [*simp*]: *of-phantom* ∘ *phantom* = *id*  
 ⟨*proof*⟩

**syntax** *-Phantom* :: *type* ⇒ *logic* (⟨⟨*indent=1 notation=⟨mixfix Phantom⟩Phantom/(1'(-))⟩⟩)*

**syntax-consts** *-Phantom* == *phantom*

**translations**

*Phantom*(*'t*) => *CONST phantom* :: - ⇒ (*'t*, -) *phantom*

⟨*ML*⟩

**lemma** *of-phantom-inject* [*simp*]:  
*of-phantom* *x* = *of-phantom* *y* ⟷ *x* = *y*  
 ⟨*proof*⟩

**end**

## 9 Cardinality of types

**theory** *Cardinality*  
**imports** *Phantom-Type*  
**begin**

### 9.1 Preliminary lemmas

**lemma** (in *type-definition*) *univ*:  
 $UNIV = Abs \text{ ' } A$   
 ⟨*proof*⟩

**lemma** (in *type-definition*) *card*:  $card (UNIV :: \text{' } b \text{ set}) = card A$   
 ⟨*proof*⟩

### 9.2 Cardinalities of types

**syntax** *-type-card* :: *type* => *nat* (⟨⟨*indent=1 notation=⟨mixfix CARD⟩CARD/(1'(-))⟩⟩)*

**syntax-consts** *-type-card* == *card*

**translations** *CARD*(*'t*) => *CONST card* (*CONST UNIV* :: *'t set*)

⟨*ML*⟩

**lemma** *card-prod* [*simp*]:  $CARD(\text{' } a \times \text{' } b) = CARD(\text{' } a) * CARD(\text{' } b)$   
 ⟨*proof*⟩

**lemma** *card-UNIV-sum*:  $CARD('a + 'b) = (if\ CARD('a) \neq 0 \wedge CARD('b) \neq 0$   
 $then\ CARD('a) + CARD('b)\ else\ 0)$

*<proof>*

**lemma** *card-sum* [*simp*]:  $CARD('a + 'b) = CARD('a::finite) + CARD('b::finite)$

*<proof>*

**lemma** *card-UNIV-option*:  $CARD('a\ option) = (if\ CARD('a) = 0\ then\ 0\ else$   
 $CARD('a) + 1)$

*<proof>*

**lemma** *card-option* [*simp*]:  $CARD('a\ option) = Suc\ CARD('a::finite)$

*<proof>*

**lemma** *card-UNIV-set*:  $CARD('a\ set) = (if\ CARD('a) = 0\ then\ 0\ else\ 2 \wedge CARD('a))$

*<proof>*

**lemma** *card-set* [*simp*]:  $CARD('a\ set) = 2 \wedge CARD('a::finite)$

*<proof>*

**lemma** *card-nat* [*simp*]:  $CARD(nat) = 0$

*<proof>*

**lemma** *card-fun*:  $CARD('a \Rightarrow 'b) = (if\ CARD('a) \neq 0 \wedge CARD('b) \neq 0 \vee$   
 $CARD('b) = 1\ then\ CARD('b) \wedge CARD('a)\ else\ 0)$

*<proof>*

**corollary** *finite-UNIV-fun*:

$finite\ (UNIV :: ('a \Rightarrow 'b)\ set) \longleftrightarrow$

$finite\ (UNIV :: 'a\ set) \wedge finite\ (UNIV :: 'b\ set) \vee CARD('b) = 1$

(**is** ?lhs  $\longleftrightarrow$  ?rhs)

*<proof>*

**lemma** *card-literal*:  $CARD(String.literal) = 0$

*<proof>*

### 9.3 Classes with at least 1 and 2

Class *finite* already captures "at least 1"

**lemma** *zero-less-card-finite* [*simp*]:  $0 < CARD('a::finite)$

*<proof>*

**lemma** *one-le-card-finite* [*simp*]:  $Suc\ 0 \leq CARD('a::finite)$

*<proof>*

**class** *CARD-1* =

**assumes** *CARD-1*:  $CARD ('a) = 1$

**begin**

```
subclass finite
```

```
⟨proof⟩
```

```
end
```

Class for cardinality "at least 2"

```
class card2 = finite +
```

```
  assumes two-le-card:  $2 \leq \text{CARD}('a)$ 
```

```
lemma one-less-card:  $\text{Suc } 0 < \text{CARD}('a::\text{card2})$ 
```

```
⟨proof⟩
```

```
lemma one-less-int-card:  $1 < \text{int } \text{CARD}('a::\text{card2})$ 
```

```
⟨proof⟩
```

#### 9.4 A type class for deciding finiteness of types

```
type-synonym 'a finite-UNIV = ('a, bool) phantom
```

```
class finite-UNIV =
```

```
  fixes finite-UNIV :: ('a, bool) phantom
```

```
  assumes finite-UNIV: finite-UNIV = Phantom('a) (finite (UNIV :: 'a set))
```

```
lemma finite-UNIV-code [code-unfold]:
```

```
  finite (UNIV :: 'a :: finite-UNIV set)
```

```
   $\longleftrightarrow$  of-phantom (finite-UNIV :: 'a finite-UNIV)
```

```
⟨proof⟩
```

#### 9.5 A type class for computing the cardinality of types

```
definition is-list-UNIV :: 'a list  $\Rightarrow$  bool
```

```
where is-list-UNIV xs = (let c =  $\text{CARD}('a)$  in if c = 0 then False else size  
(remdups xs) = c)
```

```
lemma is-list-UNIV-iff: is-list-UNIV xs  $\longleftrightarrow$  set xs = UNIV
```

```
⟨proof⟩
```

```
type-synonym 'a card-UNIV = ('a, nat) phantom
```

```
class card-UNIV = finite-UNIV +
```

```
  fixes card-UNIV :: 'a card-UNIV
```

```
  assumes card-UNIV: card-UNIV = Phantom('a)  $\text{CARD}('a)$ 
```

#### 9.6 Instantiations for *card-UNIV*

```
instantiation nat :: card-UNIV begin
```

```
definition finite-UNIV = Phantom(nat) False
```

```
definition card-UNIV = Phantom(nat) 0
```

```
instance ⟨proof⟩
```

**end**

**instantiation** *int* :: *card-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom(int) False*  
**definition** *card-UNIV* = *Phantom(int) 0*  
**instance** *<proof>*  
**end**

**instantiation** *natural* :: *card-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom(natural) False*  
**definition** *card-UNIV* = *Phantom(natural) 0*  
**instance**  
*<proof>*  
**end**

**instantiation** *integer* :: *card-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom(integer) False*  
**definition** *card-UNIV* = *Phantom(integer) 0*  
**instance**  
*<proof>*  
**end**

**instantiation** *list* :: (*type*) *card-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom('a list) False*  
**definition** *card-UNIV* = *Phantom('a list) 0*  
**instance** *<proof>*  
**end**

**instantiation** *unit* :: *card-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom(unit) True*  
**definition** *card-UNIV* = *Phantom(unit) 1*  
**instance** *<proof>*  
**end**

**instantiation** *bool* :: *card-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom(bool) True*  
**definition** *card-UNIV* = *Phantom(bool) 2*  
**instance** *<proof>*  
**end**

**instantiation** *char* :: *card-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom(char) True*  
**definition** *card-UNIV* = *Phantom(char) 256*  
**instance** *<proof>*  
**end**

**instantiation** *prod* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom('a × 'b)*  
*(of-phantom (finite-UNIV :: 'a finite-UNIV) ∧ of-phantom (finite-UNIV :: 'b*

```

finite-UNIV)
instance <proof>
end

instantiation prod :: (card-UNIV, card-UNIV) card-UNIV begin
definition card-UNIV = Phantom('a × 'b)
  (of-phantom (card-UNIV :: 'a card-UNIV) * of-phantom (card-UNIV :: 'b card-UNIV))
instance <proof>
end

instantiation sum :: (finite-UNIV, finite-UNIV) finite-UNIV begin
definition finite-UNIV = Phantom('a + 'b)
  (of-phantom (finite-UNIV :: 'a finite-UNIV) ∧ of-phantom (finite-UNIV :: 'b
finite-UNIV))
instance
  <proof>
end

instantiation sum :: (card-UNIV, card-UNIV) card-UNIV begin
definition card-UNIV = Phantom('a + 'b)
  (let ca = of-phantom (card-UNIV :: 'a card-UNIV);
      cb = of-phantom (card-UNIV :: 'b card-UNIV)
      in if ca ≠ 0 ∧ cb ≠ 0 then ca + cb else 0)
instance <proof>
end

instantiation fun :: (finite-UNIV, card-UNIV) finite-UNIV begin
definition finite-UNIV = Phantom('a ⇒ 'b)
  (let cb = of-phantom (card-UNIV :: 'b card-UNIV)
      in cb = 1 ∨ of-phantom (finite-UNIV :: 'a finite-UNIV) ∧ cb ≠ 0)
instance
  <proof>
end

instantiation fun :: (card-UNIV, card-UNIV) card-UNIV begin
definition card-UNIV = Phantom('a ⇒ 'b)
  (let ca = of-phantom (card-UNIV :: 'a card-UNIV);
      cb = of-phantom (card-UNIV :: 'b card-UNIV)
      in if ca ≠ 0 ∧ cb ≠ 0 ∨ cb = 1 then cb ^ ca else 0)
instance <proof>
end

instantiation option :: (finite-UNIV) finite-UNIV begin
definition finite-UNIV = Phantom('a option) (of-phantom (finite-UNIV :: 'a fi-
nite-UNIV))
instance <proof>
end

instantiation option :: (card-UNIV) card-UNIV begin

```

**definition** *card-UNIV* = *Phantom*('a option)  
 (let *c* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*) in if *c* ≠ 0 then *Suc c* else 0)  
**instance** ⟨*proof*⟩  
**end**

**instantiation** *String.literal* :: *card-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom*(*String.literal*) *False*  
**definition** *card-UNIV* = *Phantom*(*String.literal*) 0  
**instance**  
 ⟨*proof*⟩  
**end**

**instantiation** *set* :: (*finite-UNIV*) *finite-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom*('a *set*) (*of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*))  
**instance** ⟨*proof*⟩  
**end**

**instantiation** *set* :: (*card-UNIV*) *card-UNIV* **begin**  
**definition** *card-UNIV* = *Phantom*('a *set*)  
 (let *c* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*) in if *c* = 0 then 0 else 2 ^ *c*)  
**instance** ⟨*proof*⟩  
**end**

**lemma** *UNIV-finite-1*: *UNIV* = *set* [*finite-1.a1*]  
 ⟨*proof*⟩

**lemma** *UNIV-finite-2*: *UNIV* = *set* [*finite-2.a1*, *finite-2.a2*]  
 ⟨*proof*⟩

**lemma** *UNIV-finite-3*: *UNIV* = *set* [*finite-3.a1*, *finite-3.a2*, *finite-3.a3*]  
 ⟨*proof*⟩

**lemma** *UNIV-finite-4*: *UNIV* = *set* [*finite-4.a1*, *finite-4.a2*, *finite-4.a3*, *finite-4.a4*]  
 ⟨*proof*⟩

**lemma** *UNIV-finite-5*:  
*UNIV* = *set* [*finite-5.a1*, *finite-5.a2*, *finite-5.a3*, *finite-5.a4*, *finite-5.a5*]  
 ⟨*proof*⟩

**instantiation** *Enum.finite-1* :: *card-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom*(*Enum.finite-1*) *True*  
**definition** *card-UNIV* = *Phantom*(*Enum.finite-1*) 1  
**instance**  
 ⟨*proof*⟩  
**end**

**instantiation** *Enum.finite-2* :: *card-UNIV* **begin**  
**definition** *finite-UNIV* = *Phantom*(*Enum.finite-2*) *True*

```

definition card-UNIV = Phantom(Enum.finite-2) 2
instance
  ⟨proof⟩
end

```

```

instantiation Enum.finite-3 :: card-UNIV begin
definition finite-UNIV = Phantom(Enum.finite-3) True
definition card-UNIV = Phantom(Enum.finite-3) 3
instance
  ⟨proof⟩
end

```

```

instantiation Enum.finite-4 :: card-UNIV begin
definition finite-UNIV = Phantom(Enum.finite-4) True
definition card-UNIV = Phantom(Enum.finite-4) 4
instance
  ⟨proof⟩
end

```

```

instantiation Enum.finite-5 :: card-UNIV begin
definition finite-UNIV = Phantom(Enum.finite-5) True
definition card-UNIV = Phantom(Enum.finite-5) 5
instance
  ⟨proof⟩
end

```

```

end

```

## 10 Code setup for sets with cardinality type information

```

theory Code-Cardinality imports Cardinality begin

```

Implement  $CARD('a)$  via *card-UNIV-class.card-UNIV* and provide implementations for *finite*, *card*,  $(\subseteq)$ , and  $(=)$  if the calling context already provides *finite-UNIV* and *card-UNIV* instances. If we implemented the latter always via *card-UNIV-class.card-UNIV*, we would require instances of essentially all element types, i.e., a lot of instantiation proofs and – at run time – possibly slow dictionary constructions.

```

context
begin

```

```

qualified definition card-UNIV' :: 'a card-UNIV
where [code del]: card-UNIV' = Phantom('a) CARD('a)

```

```

lemma CARD-code [code-unfold]:
  CARD('a) = of-phantom (card-UNIV' :: 'a card-UNIV)
  ⟨proof⟩

```



**lemma** *card-UNIV'-code* [code]:

*card-UNIV' = card-UNIV*

*<proof>*

**end**

**lemma** *card-Compl*:

*finite A  $\implies$  card (- A) = card (UNIV :: 'a set) - card (A :: 'a set)*

*<proof>*

**context** *fixes* *xs :: 'a :: finite-UNIV list*

**begin**

**qualified definition** *finite' :: 'a set  $\Rightarrow$  bool*

**where** [*simp, code del, code-abbrev*]: *finite' = finite*

**lemma** *finite'-code* [code]:

*finite' (set xs)  $\longleftrightarrow$  True*

*finite' (List.coset xs)  $\longleftrightarrow$  of-phantom (finite-UNIV :: 'a finite-UNIV)*

*<proof>*

**end**

**context** *fixes* *xs :: 'a :: card-UNIV list*

**begin**

**qualified definition** *card' :: 'a set  $\Rightarrow$  nat*

**where** [*simp, code del, code-abbrev*]: *card' = card*

**lemma** *card'-code* [code]:

*card' (set xs) = length (remdups xs)*

*card' (List.coset xs) = of-phantom (card-UNIV :: 'a card-UNIV) - length (remdups xs)*

*<proof>* **definition** *subset' :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool*

**where** [*simp, code del, code-abbrev*]: *subset' = ( $\subseteq$ )*

**lemma** *subset'-code* [code]:

*subset' A (List.coset ys)  $\longleftrightarrow$  ( $\forall y \in$  set ys.  $y \notin A$ )*

*subset' (set ys) B  $\longleftrightarrow$  ( $\forall y \in$  set ys.  $y \in B$ )*

*subset' (List.coset xs) (set ys)  $\longleftrightarrow$  (let  $n = \text{CARD}('a)$  in  $n > 0 \wedge \text{card}(\text{set } xs @ ys) = n$ )*

*<proof>* **definition** *eq-set :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool*

**where** [*simp, code del, code-abbrev*]: *eq-set = (=)*

**lemma** *eq-set-code* [code]:

**fixes** *ys*

**defines** *rhs  $\equiv$*

*let  $n = \text{CARD}('a)$*

```

in if n = 0 then False else
  let xs' = remdups xs; ys' = remdups ys
  in length xs' + length ys' = n ∧ (∀ x ∈ set xs'. x ∉ set ys') ∧ (∀ y ∈ set ys'.
y ∉ set xs')
  shows eq-set (List.coset xs) (set ys) ⟷ rhs
  and eq-set (set ys) (List.coset xs) ⟷ rhs
  and eq-set (set xs) (set ys) ⟷ (∀ x ∈ set xs. x ∈ set ys) ∧ (∀ y ∈ set ys. y ∈
set xs)
  and eq-set (List.coset xs) (List.coset ys) ⟷ (∀ x ∈ set xs. x ∈ set ys) ∧ (∀ y ∈
set ys. y ∈ set xs)
⟨proof⟩

end

```

Provide more informative exceptions than Match for non-rewritten cases. If generated code raises one these exceptions, then a code equation calls the mentioned operator for an element type that is not an instance of *card-UNIV* and is therefore not implemented via *card-UNIV-class.card-UNIV*. Constrain the element type with sort *card-UNIV* to change this.

**lemma** *card-coset-error* [code]:

```

card (List.coset xs) =
  Code.abort (STR "card (List.coset -) requires type class instance card-UNIV")
  (λ-. card (List.coset xs))
⟨proof⟩

```

**lemma** *coset-subseteq-set-code* [code]:

```

List.coset xs ⊆ set ys ⟷
  (if xs = [] ∧ ys = [] then False
  else Code.abort
    (STR "subset-eq (List.coset -) (List.set -) requires type class instance card-UNIV")
    (λ-. List.coset xs ⊆ set ys))
⟨proof⟩

```

**notepad begin** — test code setup

⟨proof⟩

**end**

**end**

## 11 Eliminating pattern matches

**theory** *Case-Converter*

**imports** *Main*

**begin**

**definition** *missing-pattern-match* :: *String.literal* ⇒ (*unit* ⇒ 'a) ⇒ 'a **where**  
[*code del*]: *missing-pattern-match* m f = f ()

**lemma** *missing-pattern-match-cong* [*cong*]:  
 $m = m' \implies \text{missing-pattern-match } m \ f = \text{missing-pattern-match } m' \ f$   
 ⟨*proof*⟩

**lemma** *missing-pattern-match-code* [*code-unfold*]:  
 $\text{missing-pattern-match} = \text{Code.abort}$   
 ⟨*proof*⟩

⟨*ML*⟩

**end**

## 12 Lazy types in generated code

**theory** *Code-Lazy*  
**imports** *Case-Converter*  
**keywords**  
*code-lazy-type*  
*activate-lazy-type*  
*deactivate-lazy-type*  
*activate-lazy-types*  
*deactivate-lazy-types*  
*print-lazy-types* :: *thy-decl*  
**begin**

This theory and the CodeLazy tool described in [3].

It hooks into Isabelle’s code generator such that the generated code evaluates a user-specified set of type constructors lazily, even in target languages with eager evaluation. The lazy type must be algebraic, i.e., values must be built from constructors and a corresponding case operator decomposes them. Every datatype and codatatype is algebraic and thus eligible for lazification.

### 12.1 The type *lazy*

**typedef** *'a lazy* = *UNIV* :: *'a set* ⟨*proof*⟩  
**setup-lifting** *type-definition-lazy*  
**lift-definition** *delay* :: (*unit*  $\Rightarrow$  *'a*)  $\Rightarrow$  *'a lazy* **is**  $\lambda f. f \ ()$  ⟨*proof*⟩  
**lift-definition** *force* :: *'a lazy*  $\Rightarrow$  *'a* **is**  $\lambda x. x$  ⟨*proof*⟩

**code-datatype** *delay*  
**lemma** *force-delay* [*code*]:  $\text{force } (\text{delay } f) = f \ ()$  ⟨*proof*⟩  
**lemma** *delay-force*:  $\text{delay } (\lambda -. \text{force } s) = s$  ⟨*proof*⟩

**definition** *termify-lazy2* :: *'a* :: *typerep lazy*  $\Rightarrow$  *term*  
**where** *termify-lazy2* *x* =  
 $\text{Code-Evaluation.App } (\text{Code-Evaluation.Const } (\text{STR } \text{"Code-Lazy.delay"})) (\text{TYPEREPEP}((\text{unit} \Rightarrow \text{'a}) \Rightarrow \text{'a lazy}))$

```
(Code-Evaluation.Const (STR "Pure.dummy-pattern") (TYPEREP((unit =>
'a))))
```

**definition** *termify-lazy* ::

```
(String.literal => 'typerep => 'term) =>
('term => 'term => 'term) =>
(String.literal => 'typerep => 'term => 'term) =>
'typerep => ('typerep => 'typerep => 'typerep) => ('typerep => 'typerep) =>
('a => 'term) => 'typerep => 'a :: typerep lazy => 'term => term
where termify-lazy - - - - - x = termify-lazy2 x
```

**declare** [[code drop: Code-Evaluation.term-of :: - lazy => -]]

**lemma** *term-of-lazy-code* [code]:

```
Code-Evaluation.term-of x ≡
termify-lazy
Code-Evaluation.Const Code-Evaluation.App Code-Evaluation.Abs
TYPEREP(unit) (λT U. typerep.Typeprep (STR "fun") [T, U]) (λT. type-
rep.Typeprep (STR "Code-Lazy.lazy") [T])
Code-Evaluation.term-of TYPEREP('a) x (Code-Evaluation.Const (STR ""))
(TYPEREP(unit))
for x :: 'a :: {typerep, term-of} lazy
⟨proof⟩
```

The implementations of `- lazy` using language primitives cache forced values.

Term reconstruction for lazy looks into the lazy value and reconstructs it to the depth it has been evaluated. This is not done for Haskell as we do not know of any portable way to inspect whether a lazy value has been evaluated to or not.

**code-printing code-module** *Lazy* → (SML)

```
⟨signature LAZY =
```

```
sig
```

```
type 'a lazy;
val lazy : (unit -> 'a) -> 'a lazy;
val force : 'a lazy -> 'a;
val peek : 'a lazy -> 'a option
val termify-lazy :
(string -> 'typerep -> 'term) ->
('term -> 'term -> 'term) ->
(string -> 'typerep -> 'term -> 'term) ->
'typerep -> ('typerep -> 'typerep -> 'typerep) -> ('typerep -> 'typerep) ->
('a -> 'term) -> 'typerep -> 'a lazy -> 'term -> 'term;
end;
```

```
structure Lazy : LAZY =
```

```
struct
```

```
datatype 'a content =
```

```

    Delay of unit -> 'a
  | Value of 'a
  | Exn of exn;

datatype 'a lazy = Lazy of 'a content ref;

fun lazy f = Lazy (ref (Delay f));

fun force (Lazy x) = case !x of
  Delay f => (
    let val res = f (); val - = x := Value res; in res end
    handle exn => (x := Exn exn; raise exn))
  | Value x => x
  | Exn exn => raise exn;

fun peek (Lazy x) = case !x of
  Value x => SOME x
  | - => NONE;

fun termify-lazy const app abs unitT funT lazyT term-of T x - =
  app (const Code-Lazy.delay (funT (funT unitT T) (lazyT T)))
    (case peek x of SOME y => abs - unitT (term-of y)
     | - => const Pure.dummy-pattern (funT unitT T));

end;> for type-constructor lazy constant delay force termify-lazy
| type-constructor lazy -> (SML) - Lazy.lazy
| constant delay -> (SML) Lazy.lazy
| constant force -> (SML) Lazy.force
| constant termify-lazy -> (SML) Lazy.termify'-lazy

code-reserved (SML) Lazy

code-printing — For code generation within the Isabelle environment, we reuse
the thread-safe implementation of lazy from ~/src/Pure/Concurrent/lazy.ML
code-module Lazy -> (Eval) <> for constant undefined
| type-constructor lazy -> (Eval) - Lazy.lazy
| constant delay -> (Eval) Lazy.lazy
| constant force -> (Eval) Lazy.force
| code-module Termify-Lazy -> (Eval)
<structure Termify-Lazy = struct
fun termify-lazy
  (-: string -> typ -> term) (-: term -> term -> term) (-: string -> typ ->
term -> term)
  (-: typ) (-: typ -> typ -> typ) (-: typ -> typ)
  (term-of: 'a -> term) (T: typ) (x: 'a Lazy.lazy) (-: term) =
  Const (Code-Lazy.delay, (HOLogic.unitT --> T) --> Type (Code-Lazy.lazy,
[T])) $
  (case Lazy.peek x of
    SOME (Exn.Res x) => absdummy HOLogic.unitT (term-of x)

```

```
| - => Const (Pure.dummy-pattern, HOLogic.unitT --> T));
end;› for constant termify-lazy
| constant termify-lazy → (Eval) Termify'-Lazy.termify'-lazy
```

**code-reserved** (Eval) Termify-Lazy

### code-printing

```
type-constructor lazy → (OCaml) - Lazy.t
| constant delay → (OCaml) Lazy.from'-fun
| constant force → (OCaml) Lazy.force
| code-module Termify-Lazy → (OCaml)
⟨module Termify-Lazy : sig
  val termify-lazy :
    (string -> 'typerep -> 'term) ->
    ('term -> 'term -> 'term) ->
    (string -> 'typerep -> 'term -> 'term) ->
    'typerep -> ('typerep -> 'typerep -> 'typerep) -> ('typerep -> 'typerep) ->
    ('a -> 'term) -> 'typerep -> 'a Lazy.t -> 'term -> 'term
end = struct
```

```
let termify-lazy const app abs unitT funT lazyT term-of ty x =
  app (const Code-Lazy.delay (funT (funT unitT ty) (lazyT ty)))
    (if Lazy.is-val x then abs - unitT (term-of (Lazy.force x))
     else const Pure.dummy-pattern (funT unitT ty));;
```

```
end;› for constant termify-lazy
| constant termify-lazy → (OCaml) Termify'-Lazy.termify'-lazy
```

**code-reserved** (OCaml) Lazy Termify-Lazy

### code-printing

```
code-module Lazy → (Haskell) ⟨
module Lazy(Lazy, delay, force) where
```

```
newtype Lazy a = Lazy a
delay f = Lazy (f ())
force (Lazy x) = x› for type-constructor lazy constant delay force
| type-constructor lazy → (Haskell) Lazy.Lazy -
| constant delay → (Haskell) Lazy.delay
| constant force → (Haskell) Lazy.force
```

**code-reserved** (Haskell) Lazy

### code-printing

```
code-module Lazy → (Scala)
⟨object Lazy {
  final class Lazy[A] (f: Unit => A) {
    var evaluated = false;
```

```

    lazy val x: A = f(())

    def get() : A = {
      evaluated = true;
      return x
    }
  }

  def force[A] (x: Lazy[A]) : A = {
    return x.get()
  }

  def delay[A] (f: Unit => A) : Lazy[A] = {
  }

  def termify-lazy[Typerep, Term, A] (
    const: String => Typerep => Term,
    app: Term => Term => Term,
    abs: String => Typerep => Term => Term,
    unitT: Typerep,
    funT: Typerep => Typerep => Typerep,
    lazyT: Typerep => Typerep,
    term-of: A => Term,
    ty: Typerep,
    x: Lazy[A],
    dummy: Term) : Term = {
    x.evaluated match {
      case true => app(const(Code-Lazy.delay)(funT(funT(unitT)(ty))(lazyT(ty))))(abs(-)(unitT)(term-of(x.get)))
      case false => app(const(Code-Lazy.delay)(funT(funT(unitT)(ty))(lazyT(ty))))(const(Pure.dummy-pattern))
    }
  }
}
} for type-constructor lazy constant delay force termify-lazy
| type-constructor lazy → (Scala) Lazy.Lazy[-]
| constant delay → (Scala) Lazy.delay
| constant force → (Scala) Lazy.force
| constant termify-lazy → (Scala) Lazy.termify'-lazy

```

**code-reserved** (Scala) Lazy

Make evaluation with the simplifier respect *delays*.

**lemma** *delay-lazy-cong*: *delay f = delay f* ⟨proof⟩  
 ⟨ML⟩

## 12.2 Implementation

⟨ML⟩

end

## 13 Test infrastructure for the code generator

```
theory Code-Test
imports Main
keywords test-code :: diag
begin
```

### 13.1 YXML encoding for *term*

```
datatype (plugins del: code size quickcheck) yxml-of-term = YXML
```

```
lemma yot-anything: x = (y :: yxml-of-term)
⟨proof⟩
```

```
definition yot-empty :: yxml-of-term where [code del]: yot-empty = YXML
```

```
definition yot-literal :: String.literal ⇒ yxml-of-term
```

```
  where [code del]: yot-literal - = YXML
```

```
definition yot-append :: yxml-of-term ⇒ yxml-of-term ⇒ yxml-of-term
```

```
  where [code del]: yot-append - - = YXML
```

```
definition yot-concat :: yxml-of-term list ⇒ yxml-of-term
```

```
  where [code del]: yot-concat - = YXML
```

Serialise *yxml-of-term* to native string of target language

```
code-printing type-constructor yxml-of-term
```

```
  → (SML) string
```

```
  and (OCaml) string
```

```
  and (Haskell) String
```

```
  and (Scala) String
```

```
| constant yot-empty
```

```
  → (SML)
```

```
  and (OCaml)
```

```
  and (Haskell)
```

```
  and (Scala)
```

```
| constant yot-literal
```

```
  → (SML) -
```

```
  and (OCaml) -
```

```
  and (Haskell) -
```

```
  and (Scala) -
```

```
| constant yot-append
```

```
  → (SML) String.concat [(-), (-)]
```

```
  and (OCaml) String.concat [(-); (-)]
```

```
  and (Haskell) infixr 5 ++
```

```
  and (Scala) infixl 5 +
```

```
| constant yot-concat
```

```
  → (SML) String.concat
```

```
  and (OCaml) String.concat
```

```
  and (Haskell) Prelude.concat
```

```
  and (Scala) -.mkString()
```

Stripped-down implementations of Isabelle’s XML tree with YXML en-



coding as defined in `~/src/Pure/PIDE/xml.ML`, `~/src/Pure/PIDE/yxml.ML`  
sufficient to encode *term* as in `~/src/Pure/term_xml.ML`.

**datatype** (*plugins del: code size quickcheck*) *xml-tree* = *XML-Tree*

**lemma** *xml-tree-anything*:  $x = (y :: \textit{xml-tree})$   
(*proof*)

**context begin**  
(*ML*)

**type-synonym** *attributes* = (*String.literal* × *String.literal*) *list*  
**type-synonym** *body* = *xml-tree list*

**definition** *Elem* :: *String.literal* ⇒ *attributes* ⇒ *xml-tree list* ⇒ *xml-tree*  
**where** [*code del*]: *Elem* - - = *XML-Tree*

**definition** *Text* :: *String.literal* ⇒ *xml-tree*  
**where** [*code del*]: *Text* - = *XML-Tree*

**definition** *node* :: *xml-tree list* ⇒ *xml-tree*  
**where** *node* *ts* = *Elem* (*STR* "'") [] *ts*

**definition** *tagged* :: *String.literal* ⇒ *String.literal option* ⇒ *xml-tree list* ⇒ *xml-tree*  
**where** *tagged* *tag* *x* *ts* = *Elem* *tag* (*case* *x* of *None* ⇒ [] | *Some* *x'* ⇒ [(*STR* "'0'", *x'*)] *ts*)

**definition** *list* **where** *list* *f* *xs* = *map* (*node* ∘ *f*) *xs*

**definition** *X* :: *yxml-of-term* **where** *X* = *yot-literal* (*STR* 0x05)

**definition** *Y* :: *yxml-of-term* **where** *Y* = *yot-literal* (*STR* 0x06)

**definition** *XY* :: *yxml-of-term* **where** *XY* = *yot-append* *X* *Y*

**definition** *XYX* :: *yxml-of-term* **where** *XYX* = *yot-append* *XY* *X*

**end**

**code-datatype** *xml.Elem* *xml.Text*

**definition** *yxml-string-of-xml-tree* :: *xml-tree* ⇒ *yxml-of-term* ⇒ *yxml-of-term*  
**where** [*code del*]: *yxml-string-of-xml-tree* - - = *YXML*

**lemma** *yxml-string-of-xml-tree-code* [*code*]:  
*yxml-string-of-xml-tree* (*xml.Elem* *name* *atts* *ts*) *rest* =  
  *yot-append* *xml.XY* (  
  *yot-append* (*yot-literal* *name*) (  
  *foldr* ( $\lambda(a, x)$  *rest*.  
    *yot-append* *xml.Y* (  
      *yot-append* (*yot-literal* *a*) (  
      *yot-append* (*yot-literal* (*STR* "'='')) (  
      *yot-append* (*yot-literal* *x*) *rest*)))))) *atts* (  
  *rest*))

```

  foldr yxml-string-of-xml-tree ts (
    yot-append xml.XYX rest))))
  yxml-string-of-xml-tree (xml.Text s) rest = yot-append (yot-literal s) rest
⟨proof⟩

```

**definition** *yxml-string-of-body* :: *xml.body* ⇒ *yxml-of-term*  
**where** *yxml-string-of-body* ts = foldr *yxml-string-of-xml-tree* ts *yot-empty*

Encoding *term* into XML trees as defined in `~/src/Pure/term_xml.ML`.

**definition** *xml-of-ty* :: *Typerep.typerep* ⇒ *xml.body*  
**where** [`del`]: *xml-of-ty* - = [*XML-Tree*]

**definition** *xml-of-term* :: *Code-Evaluation.term* ⇒ *xml.body*  
**where** [`del`]: *xml-of-term* - = [*XML-Tree*]

**lemma** *xml-of-ty-code* [`code`]:  
*xml-of-ty* (*typerep.Typerep t args*) = [*xml.tagged* (*STR "0"*) (*Some t*) (*xml.list xml-of-ty args*)]

⟨proof⟩

**lemma** *xml-of-term-code* [`code`]:  
*xml-of-term* (*Code-Evaluation.Const x ty*) = [*xml.tagged* (*STR "0"*) (*Some x*) (*xml-of-ty ty*)]  
*xml-of-term* (*Code-Evaluation.App t1 t2*) = [*xml.tagged* (*STR "5"*) *None* [*xml.node xml-of-term t1*, *xml.node xml-of-term t2*]]  
*xml-of-term* (*Code-Evaluation.Abs x ty t*) = [*xml.tagged* (*STR "4"*) (*Some x*) [*xml.node xml-of-ty ty*, *xml.node xml-of-term t*]]

— **FIXME**: *Code-Evaluation.Free* is used only in *HOL.Quickcheck-Narrowing* to represent uninstantiated parameters in constructors. Here, we always translate them to **Free** variables.

*xml-of-term* (*Code-Evaluation.Free x ty*) = [*xml.tagged* (*STR "1"*) (*Some x*) (*xml-of-ty ty*)]

⟨proof⟩

**definition** *yxml-string-of-term* :: *Code-Evaluation.term* ⇒ *yxml-of-term*  
**where** *yxml-string-of-term* = *yxml-string-of-body* ∘ *xml-of-term*

## 13.2 Test engine and drivers

⟨ML⟩

end

## 14 A combinator to build partial equivalence relations from a predicate and an equivalence relation

**theory** *Combine-PER*

```

imports Main
begin

```

```

unbundle lattice-syntax

```

```

definition combine-per :: ('a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool
  where combine-per P R = (λx y. P x ∧ P y) □ R

```

```

lemma combine-per-simp [simp]:
  combine-per P R x y ⟷ P x ∧ P y ∧ x ≈ y for R (infixl ⟨≈⟩ 50)
  ⟨proof⟩

```

```

lemma combine-per-top [simp]: combine-per ⊤ R = R
  ⟨proof⟩

```

```

lemma combine-per-eq [simp]: combine-per P HOL.eq = HOL.eq □ (λx y. P x)
  ⟨proof⟩

```

```

lemma symp-combine-per: symp R ⇒ symp (combine-per P R)
  ⟨proof⟩

```

```

lemma transp-combine-per: transp R ⇒ transp (combine-per P R)
  ⟨proof⟩

```

```

lemma combine-perI: P x ⇒ P y ⇒ x ≈ y ⇒ combine-per P R x y for R
  (infixl ⟨≈⟩ 50)
  ⟨proof⟩

```

```

lemma symp-combine-per-symp: symp R ⇒ symp (combine-per P R)
  ⟨proof⟩

```

```

lemma transp-combine-per-transp: transp R ⇒ transp (combine-per P R)
  ⟨proof⟩

```

```

lemma equivp-combine-per-part-equivp [intro?]:
  fixes R (infixl ⟨≈⟩ 50)
  assumes ∃x. P x and equivp R
  shows part-equivp (combine-per P R)
  ⟨proof⟩

```

```

end

```

## 15 Formalisation of chain-complete partial orders, continuity and admissibility

```

theory Complete-Partial-Order2
  imports Main
begin

```

**unbundle** *lattice-syntax*

**lemma** *chain-transfer* [*transfer-rule*]:

**includes** *lifting-syntax*

**shows**  $((A \text{====>} A \text{====>} (=)) \text{====>} \text{rel-set } A \text{====>} (=)) \text{ Complete-Partial-Order.chain}$   
*Complete-Partial-Order.chain*

*<proof>*

**lemma** *linorder-chain* [*simp, intro!*]:

**fixes**  $Y :: - :: \text{linorder set}$

**shows** *Complete-Partial-Order.chain*  $(\leq) Y$

*<proof>*

**lemma** *fun-lub-apply*:  $\bigwedge \text{Sup. fun-lub Sup } Y x = \text{Sup } ((\lambda f. f x) \text{ ‘ } Y)$

*<proof>*

**lemma** *fun-lub-empty* [*simp*]: *fun-lub lub*  $\{\} = (\lambda -. \text{lub } \{\})$

*<proof>*

**lemma** *chain-fun-ordD*:

**assumes** *Complete-Partial-Order.chain*  $(\text{fun-ord } le) Y$

**shows** *Complete-Partial-Order.chain*  $le ((\lambda f. f x) \text{ ‘ } Y)$

*<proof>*

**lemma** *chain-Diff*:

*Complete-Partial-Order.chain ord A*

$\implies \text{Complete-Partial-Order.chain ord } (A - B)$

*<proof>*

**lemma** *chain-rel-prodD1*:

*Complete-Partial-Order.chain*  $(\text{rel-prod } \text{orda } \text{ordb}) Y$

$\implies \text{Complete-Partial-Order.chain } \text{orda } (\text{fst } \text{ ‘ } Y)$

*<proof>*

**lemma** *chain-rel-prodD2*:

*Complete-Partial-Order.chain*  $(\text{rel-prod } \text{orda } \text{ordb}) Y$

$\implies \text{Complete-Partial-Order.chain } \text{ordb } (\text{snd } \text{ ‘ } Y)$

*<proof>*

**context** *ccpo begin*

**lemma** *ccpo-fun*: *class.ccpo*  $(\text{fun-lub Sup}) (\text{fun-ord } (\leq)) (\text{mk-less } (\text{fun-ord } (\leq)))$

*<proof>*

**lemma** *ccpo-Sup-below-iff*: *Complete-Partial-Order.chain*  $(\leq) Y \implies \text{Sup } Y \leq x$

$\longleftrightarrow (\forall y \in Y. y \leq x)$

*<proof>*

**lemma** *Sup-minus-bot*:

**assumes** *chain*: *Complete-Partial-Order.chain* ( $\leq$ )  $A$   
**shows**  $\sqcup(A - \{\sqcup\}) = \sqcup A$   
 (**is** *?lhs = ?rhs*)

*<proof>*

**lemma** *mono-lub*:

**fixes** *le-b* (**infix**  $\langle \sqsubseteq \rangle$  *l0*)  
**assumes** *chain*: *Complete-Partial-Order.chain* (*fun-ord* ( $\leq$ ))  $Y$   
**and** *mono*:  $\bigwedge f. f \in Y \implies \text{monotone } \text{le-b } (\leq) f$   
**shows** *monotone* ( $\sqsubseteq$ ) ( $\leq$ ) (*fun-lub* *Sup*  $Y$ )

*<proof>*

**context**

**fixes** *le-b* (**infix**  $\langle \sqsubseteq \rangle$  *l0*) **and**  $Y f$   
**assumes** *chain*: *Complete-Partial-Order.chain* *le-b*  $Y$   
**and** *mono1*:  $\bigwedge y. y \in Y \implies \text{monotone } \text{le-b } (\leq) (\lambda x. f x y)$   
**and** *mono2*:  $\bigwedge x a b. \llbracket x \in Y; a \sqsubseteq b; a \in Y; b \in Y \rrbracket \implies f x a \leq f x b$

**begin**

**lemma** *Sup-mono*:

**assumes** *le*:  $x \sqsubseteq y$  **and**  $x \in Y$  **and**  $y \in Y$   
**shows**  $\sqcup(f x \text{ ' } Y) \leq \sqcup(f y \text{ ' } Y)$  (**is**  $- \leq$  *?rhs*)

*<proof>*

**lemma** *diag-Sup*:  $\sqcup((\lambda x. \sqcup(f x \text{ ' } Y)) \text{ ' } Y) = \sqcup((\lambda x. f x x) \text{ ' } Y)$  (**is** *?lhs = ?rhs*)

*<proof>*

**end**

**lemma** *Sup-image-mono-le*:

**fixes** *le-b* (**infix**  $\langle \sqsubseteq \rangle$  *l0*) **and** *Sup-b* ( $\langle \bigvee \rangle$ )  
**assumes** *ccpo*: *class.ccpo* *Sup-b* ( $\sqsubseteq$ ) *lt-b*  
**assumes** *chain*: *Complete-Partial-Order.chain* ( $\sqsubseteq$ )  $Y$   
**and** *mono*:  $\bigwedge x y. \llbracket x \sqsubseteq y; x \in Y \rrbracket \implies f x \leq f y$   
**shows** *Sup* ( $f \text{ ' } Y$ )  $\leq f (\bigvee Y)$

*<proof>*

**lemma** *swap-Sup*:

**fixes** *le-b* (**infix**  $\langle \sqsubseteq \rangle$  *l0*)  
**assumes**  $Y$ : *Complete-Partial-Order.chain* ( $\sqsubseteq$ )  $Y$   
**and**  $Z$ : *Complete-Partial-Order.chain* (*fun-ord* ( $\leq$ ))  $Z$   
**and** *mono*:  $\bigwedge f. f \in Z \implies \text{monotone } (\sqsubseteq) (\leq) f$   
**shows**  $\sqcup((\lambda x. \sqcup(x \text{ ' } Y)) \text{ ' } Z) = \sqcup((\lambda x. \sqcup((\lambda f. f x) \text{ ' } Z)) \text{ ' } Y)$   
 (**is** *?lhs = ?rhs*)

*<proof>*

**lemma** *fixp-mono*:

**assumes** *fg*: *fun-ord* ( $\leq$ ) *f g*  
**and** *f*: *monotone* ( $\leq$ ) ( $\leq$ ) *f*  
**and** *g*: *monotone* ( $\leq$ ) ( $\leq$ ) *g*  
**shows** *ccpo-class.fixp f*  $\leq$  *ccpo-class.fixp g*  
 $\langle$ *proof* $\rangle$

**context** **fixes** *ordb* :: *'b*  $\Rightarrow$  *'b*  $\Rightarrow$  *bool* (**infix**  $\langle \sqsubseteq \rangle$  60) **begin**

**lemma** *iterates-mono*:

**assumes** *f*: *f*  $\in$  *ccpo.iterates* (*fun-lub Sup*) (*fun-ord* ( $\leq$ )) *F*  
**and** *mono*:  $\bigwedge f. \text{monotone } (\sqsubseteq) (\leq) f \implies \text{monotone } (\sqsubseteq) (\leq) (F f)$   
**shows** *monotone* ( $\sqsubseteq$ ) ( $\leq$ ) *f*  
 $\langle$ *proof* $\rangle$

**lemma** *fixp-preserves-mono*:

**assumes** *mono*:  $\bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. F f x)$   
**and** *mono2*:  $\bigwedge f. \text{monotone } (\sqsubseteq) (\leq) f \implies \text{monotone } (\sqsubseteq) (\leq) (F f)$   
**shows** *monotone* ( $\sqsubseteq$ ) ( $\leq$ ) (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord* ( $\leq$ )) *F*)  
**(is** *monotone* - - *?fixp*)  
 $\langle$ *proof* $\rangle$

**end**

**end**

**lemma** *monotone2monotone*:

**assumes** *2*:  $\bigwedge x. \text{monotone } \text{ordb } \text{ordc } (\lambda y. f x y)$   
**and** *t*: *monotone* *orda ordb* ( $\lambda x. t x$ )  
**and** *1*:  $\bigwedge y. \text{monotone } \text{orda } \text{ordc } (\lambda x. f x y)$   
**and** *trans*: *transp ordc*  
**shows** *monotone orda ordc* ( $\lambda x. f x (t x)$ )  
 $\langle$ *proof* $\rangle$

## 15.1 Continuity

**definition** *cont* :: (*'a set*  $\Rightarrow$  *'a*)  $\Rightarrow$  (*'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *bool*)  $\Rightarrow$  (*'b set*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'b*  $\Rightarrow$  *'b*  $\Rightarrow$  *bool*)  $\Rightarrow$  (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  *bool*

**where**

*cont luba orda lubb ordb f*  $\longleftrightarrow$   
 $(\forall Y. \text{Complete-Partial-Order.chain } \text{orda } Y \longrightarrow Y \neq \{\} \longrightarrow f (\text{luba } Y) = \text{lubb } (f \cdot Y))$

**definition** *mcont* :: (*'a set*  $\Rightarrow$  *'a*)  $\Rightarrow$  (*'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *bool*)  $\Rightarrow$  (*'b set*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'b*  $\Rightarrow$  *'b*  $\Rightarrow$  *bool*)  $\Rightarrow$  (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  *bool*

**where**

*mcont luba orda lubb ordb f*  $\longleftrightarrow$   
*monotone orda ordb f*  $\wedge$  *cont luba orda lubb ordb f*

**15.1.1 Theorem collection *cont-intro***

**named-theorems** *cont-intro continuity and admissibility intro rules*  
 ⟨ML⟩

**lemmas** [*cont-intro*] =  
*call-mono*  
*let-mono*  
*if-mono*  
*option.const-mono*  
*tailrec.const-mono*  
*bind-mono*

**experiment begin**

The following proof by simplification diverges if variables are not handled properly.

**lemma** ( $\wedge f. \text{monotone } R \ S \ f \implies \text{thesis}$ )  $\implies \text{monotone } R \ S \ g \implies \text{thesis}$   
 ⟨*proof*⟩

**end**

**declare** *if-mono[simp]*

**lemma** *monotone-id'* [*cont-intro*]: *monotone ord ord* ( $\lambda x. x$ )  
 ⟨*proof*⟩

**lemma** *monotone-applyI*:  
*monotone orda ordb F*  $\implies \text{monotone (fun-ord orda) ordb } (\lambda f. F (f x))$   
 ⟨*proof*⟩

**lemma** *monotone-if-fun* [*partial-function-mono*]:  
 $\llbracket \text{monotone (fun-ord orda) (fun-ord ordb) } F; \text{monotone (fun-ord orda) (fun-ord ordb) } G \rrbracket$   
 $\implies \text{monotone (fun-ord orda) (fun-ord ordb) } (\lambda f n. \text{if } c \ n \ \text{then } F \ f \ n \ \text{else } G \ f \ n)$   
 ⟨*proof*⟩

**lemma** *monotone-fun-apply-fun* [*partial-function-mono*]:  
*monotone (fun-ord (fun-ord ord)) (fun-ord ord)* ( $\lambda f n. f \ t \ (g \ n)$ )  
 ⟨*proof*⟩

**lemma** *monotone-fun-ord-apply*:  
*monotone orda (fun-ord ordb) f*  $\longleftrightarrow (\forall x. \text{monotone orda ordb } (\lambda y. f \ y \ x))$   
 ⟨*proof*⟩

**context** *preorder* **begin**

**declare** *transp-on-le*[*cont-intro*]

**lemma** *monotone-const* [*simp, cont-intro*]: *monotone ord* ( $\leq$ ) ( $\lambda-. c$ )

*<proof>*

**end**

**lemma** *transp-le* [*cont-intro*, *simp*]:  
*class.preorder ord (mk-less ord)  $\implies$  transp ord*  
*<proof>*

**context** *partial-function-definitions* **begin**

**declare** *const-mono* [*cont-intro*, *simp*]

**lemma** *transp-le* [*cont-intro*, *simp*]: *transp leq*  
*<proof>*

**lemma** *preorder* [*cont-intro*, *simp*]: *class.preorder leq (mk-less leq)*  
*<proof>*

**declare** *ccpo*[*cont-intro*, *simp*]

**end**

**lemma** *contI* [*intro?*]:  
 $(\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain } \text{orda } Y; Y \neq \{\} \rrbracket \implies f (\text{luba } Y) = \text{lubb } (f \text{ ' } Y))$   
 $\implies \text{cont luba } \text{orda } \text{lubb } \text{ordb } f$   
*<proof>*

**lemma** *contD*:  
 $\llbracket \text{cont luba } \text{orda } \text{lubb } \text{ordb } f; \text{Complete-Partial-Order.chain } \text{orda } Y; Y \neq \{\} \rrbracket$   
 $\implies f (\text{luba } Y) = \text{lubb } (f \text{ ' } Y)$   
*<proof>*

**lemma** *cont-id* [*simp*, *cont-intro*]:  $\bigwedge \text{Sup. cont Sup ord Sup ord id}$   
*<proof>*

**lemma** *cont-id'* [*simp*, *cont-intro*]:  $\bigwedge \text{Sup. cont Sup ord Sup ord } (\lambda x. x)$   
*<proof>*

**lemma** *cont-applyI* [*cont-intro*]:  
**assumes** *cont*: *cont luba orda lubb ordb g*  
**shows** *cont (fun-lub luba) (fun-ord orda) lubb ordb*  $(\lambda f. g (f x))$   
*<proof>*

**lemma** *call-cont*: *cont (fun-lub lub) (fun-ord ord) lub ord*  $(\lambda f. f t)$   
*<proof>*

**lemma** *cont-if* [*cont-intro*]:  
 $\llbracket \text{cont luba } \text{orda } \text{lubb } \text{ordb } f; \text{cont luba } \text{orda } \text{lubb } \text{ordb } g \rrbracket$



$\implies \text{cont luba orda lubb ordb } (\lambda x. \text{ if } c \text{ then } f x \text{ else } g x)$   
 $\langle \text{proof} \rangle$

**lemma** *mcontI* [*intro?*]:

$\llbracket \text{ monotone orda ordb } f; \text{ cont luba orda lubb ordb } f \rrbracket \implies \text{mcont luba orda lubb ordb } f$   
 $\langle \text{proof} \rangle$

**lemma** *mcont-mono*:  $\text{mcont luba orda lubb ordb } f \implies \text{monotone orda ordb } f$   
 $\langle \text{proof} \rangle$

**lemma** *mcont-cont* [*simp*]:  $\text{mcont luba orda lubb ordb } f \implies \text{cont luba orda lubb ordb } f$   
 $\langle \text{proof} \rangle$

**lemma** *mcont-monoD*:

$\llbracket \text{mcont luba orda lubb ordb } f; \text{ orda } x y \rrbracket \implies \text{ordb } (f x) (f y)$   
 $\langle \text{proof} \rangle$

**lemma** *mcont-contD*:

$\llbracket \text{mcont luba orda lubb ordb } f; \text{ Complete-Partial-Order.chain orda } Y; Y \neq \{\} \rrbracket$   
 $\implies f (\text{luba } Y) = \text{lubb } (f ' Y)$   
 $\langle \text{proof} \rangle$

**lemma** *mcont-call* [*cont-intro*, *simp*]:

$\text{mcont } (\text{fun-lub lub}) (\text{fun-ord ord}) \text{ lub ord } (\lambda f. f t)$   
 $\langle \text{proof} \rangle$

**lemma** *mcont-id'* [*cont-intro*, *simp*]:  $\text{mcont lub ord lub ord } (\lambda x. x)$   
 $\langle \text{proof} \rangle$

**lemma** *mcont-applyI*:

$\text{mcont luba orda lubb ordb } (\lambda x. F x) \implies \text{mcont } (\text{fun-lub luba}) (\text{fun-ord orda}) \text{ lubb ordb } (\lambda f. F (f x))$   
 $\langle \text{proof} \rangle$

**lemma** *mcont-if* [*cont-intro*, *simp*]:

$\llbracket \text{mcont luba orda lubb ordb } (\lambda x. f x); \text{mcont luba orda lubb ordb } (\lambda x. g x) \rrbracket$   
 $\implies \text{mcont luba orda lubb ordb } (\lambda x. \text{ if } c \text{ then } f x \text{ else } g x)$   
 $\langle \text{proof} \rangle$

**lemma** *cont-fun-lub-apply*:

$\text{cont luba orda } (\text{fun-lub lubb}) (\text{fun-ord ordb}) f \longleftrightarrow (\forall x. \text{cont luba orda lubb ordb } (\lambda y. f y x))$   
 $\langle \text{proof} \rangle$

**lemma** *mcont-fun-lub-apply*:

$\text{mcont luba orda } (\text{fun-lub lubb}) (\text{fun-ord ordb}) f \longleftrightarrow (\forall x. \text{mcont luba orda lubb ordb } (\lambda y. f y x))$

$\langle proof \rangle$

**context** *ccpo* **begin**

**lemma** *cont-const* [*simp*, *cont-intro*]: *cont luba orda Sup* ( $\leq$ ) ( $\lambda x. c$ )  
 $\langle proof \rangle$

**lemma** *mcont-const* [*cont-intro*, *simp*]:  
*mcont luba orda Sup* ( $\leq$ ) ( $\lambda x. c$ )  
 $\langle proof \rangle$

**lemma** *cont-apply*:  
**assumes** 2:  $\bigwedge x. cont\ lubb\ ordb\ Sup\ (\leq)\ (\lambda y. f\ x\ y)$   
**and** *t*: *cont luba orda lubb ordb* ( $\lambda x. t\ x$ )  
**and** 1:  $\bigwedge y. cont\ luba\ orda\ Sup\ (\leq)\ (\lambda x. f\ x\ y)$   
**and** *mono*: *monotone orda ordb* ( $\lambda x. t\ x$ )  
**and** *mono2*:  $\bigwedge x. monotone\ ordb\ (\leq)\ (\lambda y. f\ x\ y)$   
**and** *mono1*:  $\bigwedge y. monotone\ orda\ (\leq)\ (\lambda x. f\ x\ y)$   
**shows** *cont luba orda Sup* ( $\leq$ ) ( $\lambda x. f\ x\ (t\ x)$ )  
 $\langle proof \rangle$

**lemma** *mcont2mcont'*:  
 $\llbracket \bigwedge x. mcont\ lub'\ ord'\ Sup\ (\leq)\ (\lambda y. f\ x\ y);$   
 $\bigwedge y. mcont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. f\ x\ y);$   
 $mcont\ lub\ ord\ lub'\ ord'\ (\lambda y. t\ y) \rrbracket$   
 $\implies mcont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. f\ x\ (t\ x))$   
 $\langle proof \rangle$

**lemma** *mcont2mcont*:  
 $\llbracket mcont\ lub'\ ord'\ Sup\ (\leq)\ (\lambda x. f\ x); mcont\ lub\ ord\ lub'\ ord'\ (\lambda x. t\ x) \rrbracket$   
 $\implies mcont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. f\ (t\ x))$   
 $\langle proof \rangle$

**context**

**fixes** *ord* :: 'b  $\Rightarrow$  'b  $\Rightarrow$  bool (**infix**  $\langle \sqsubseteq \rangle$  60)  
**and** *lub* :: 'b set  $\Rightarrow$  'b ( $\langle \bigvee \rangle$ )

**begin**

**lemma** *cont-fun-lub-Sup*:  
**assumes** *chainM*: *Complete-Partial-Order.chain* (*fun-ord* ( $\leq$ )) *M*  
**and** *mcont* [*rule-format*]:  $\forall f \in M. mcont\ lub\ (\sqsubseteq)\ Sup\ (\leq)\ f$   
**shows** *cont lub* ( $\sqsubseteq$ ) *Sup* ( $\leq$ ) (*fun-lub Sup M*)  
 $\langle proof \rangle$

**lemma** *mcont-fun-lub-Sup*:  
 $\llbracket Complete-Partial-Order.chain\ (fun-ord\ (\leq))\ M;$   
 $\forall f \in M. mcont\ lub\ ord\ Sup\ (\leq)\ f \rrbracket$   
 $\implies mcont\ lub\ (\sqsubseteq)\ Sup\ (\leq)\ (fun-lub\ Sup\ M)$   
 $\langle proof \rangle$

**lemma** *iterates-mcont*:

**assumes**  $f: f \in \text{ccpo.iterates } (\text{fun-lub Sup}) \ (\text{fun-ord } (\leq)) \ F$   
**and**  $\text{mono}: \bigwedge f. \text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ f \implies \text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ (F f)$   
**shows**  $\text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ f$   
 $\langle \text{proof} \rangle$

**lemma** *fixp-preserves-mcont*:

**assumes**  $\text{mono}: \bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) \ (\leq) \ (\lambda f. F f x)$   
**and**  $\text{mcont}: \bigwedge f. \text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ f \implies \text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ (F f)$   
**shows**  $\text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ (\text{ccpo.fixp } (\text{fun-lub Sup}) \ (\text{fun-ord } (\leq)) \ F)$   
**(is mcont - - - ?fixp)**  
 $\langle \text{proof} \rangle$

**end**

**context**

**fixes**  $F :: 'c \Rightarrow 'c$  **and**  $U :: 'c \Rightarrow 'b \Rightarrow 'a$  **and**  $C :: ('b \Rightarrow 'a) \Rightarrow 'c$  **and**  $f$   
**assumes**  $\text{mono}: \bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) \ (\leq) \ (\lambda f. U (F (C f)) x)$   
**and**  $\text{eq}: f \equiv C (\text{ccpo.fixp } (\text{fun-lub Sup}) \ (\text{fun-ord } (\leq)) \ (\lambda f. U (F (C f))))$   
**and**  $\text{inverse}: \bigwedge f. U (C f) = f$

**begin**

**lemma** *fixp-preserves-mono-uc*:

**assumes**  $\text{mono2}: \bigwedge f. \text{monotone ord } (\leq) \ (U f) \implies \text{monotone ord } (\leq) \ (U (F f))$   
**shows**  $\text{monotone ord } (\leq) \ (U f)$   
 $\langle \text{proof} \rangle$

**lemma** *fixp-preserves-mcont-uc*:

**assumes**  $\text{mcont}: \bigwedge f. \text{mcont lubb ordb Sup } (\leq) \ (U f) \implies \text{mcont lubb ordb Sup } (\leq) \ (U (F f))$   
**shows**  $\text{mcont lubb ordb Sup } (\leq) \ (U f)$   
 $\langle \text{proof} \rangle$

**end**

**lemmas** *fixp-preserves-mono1* = *fixp-preserves-mono-uc*[of  $\lambda x. x - \lambda x. x$ , *OF - - refl*]

**lemmas** *fixp-preserves-mono2* =  
*fixp-preserves-mono-uc*[of *case-prod - curry*, *unfolded case-prod-curry curry-case-prod*,  
*OF - - refl*]

**lemmas** *fixp-preserves-mono3* =  
*fixp-preserves-mono-uc*[of  $\lambda f. \text{case-prod } (\text{case-prod } f) - \lambda f. \text{curry } (\text{curry } f)$ , *un-*  
*folded case-prod-curry curry-case-prod*, *OF - - refl*]

**lemmas** *fixp-preserves-mono4* =  
*fixp-preserves-mono-uc*[of  $\lambda f. \text{case-prod } (\text{case-prod } (\text{case-prod } f)) - \lambda f. \text{curry } (\text{curry } (\text{curry } f))$ , *unfolded case-prod-curry curry-case-prod*, *OF - - refl*]

**lemmas** *fixp-preserves-mcont1* = *fixp-preserves-mcont-uc*[of  $\lambda x. x - \lambda x. x$ , *OF - -*

refl]

**lemmas** *fixp-preserves-mcont2* =

*fixp-preserves-mcont-uc*[of *case-prod - curry, unfolded case-prod-curry curry-case-prod, OF - - refl*]

**lemmas** *fixp-preserves-mcont3* =

*fixp-preserves-mcont-uc*[of  $\lambda f. \text{case-prod } (\text{case-prod } f) - \lambda f. \text{curry } (\text{curry } f)$ , *unfolded case-prod-curry curry-case-prod, OF - - refl*]

**lemmas** *fixp-preserves-mcont4* =

*fixp-preserves-mcont-uc*[of  $\lambda f. \text{case-prod } (\text{case-prod } (\text{case-prod } f)) - \lambda f. \text{curry } (\text{curry } (\text{curry } f))$ , *unfolded case-prod-curry curry-case-prod, OF - - refl*]

**end**

**lemma** (in *preorder*) *monotone-if-bot*:

**fixes** *bot*

**assumes** *mono*:  $\bigwedge x y. \llbracket x \leq y; \neg (x \leq \text{bound}) \rrbracket \implies \text{ord } (f x) (f y)$

**and** *bot*:  $\bigwedge x. \neg x \leq \text{bound} \implies \text{ord } \text{bot } (f x) \text{ ord } \text{bot } \text{bot}$

**shows** *monotone* ( $\leq$ ) *ord* ( $\lambda x. \text{if } x \leq \text{bound} \text{ then } \text{bot} \text{ else } f x$ )

*<proof>*

**lemma** (in *ccpo*) *mcont-if-bot*:

**fixes** *bot* **and** *lub* ( $\langle \bigvee \rangle$ ) **and** *ord* (**infix**  $\langle \sqsubseteq \rangle$  60)

**assumes** *ccpo*: *class.ccpo lub* ( $\sqsubseteq$ ) *lt*

**and** *mono*:  $\bigwedge x y. \llbracket x \leq y; \neg x \leq \text{bound} \rrbracket \implies f x \sqsubseteq f y$

**and** *cont*:  $\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain } (\leq) Y; Y \neq \{\}; \bigwedge x. x \in Y \implies \neg x \leq \text{bound} \rrbracket \implies f (\bigsqcup Y) = \bigvee (f \text{ ` } Y)$

**and** *bot*:  $\bigwedge x. \neg x \leq \text{bound} \implies \text{bot} \sqsubseteq f x$

**shows** *mcont Sup* ( $\leq$ ) *lub* ( $\sqsubseteq$ ) ( $\lambda x. \text{if } x \leq \text{bound} \text{ then } \text{bot} \text{ else } f x$ ) (**is** *mcont - - ?g*)

*<proof>*

**context** *partial-function-definitions* **begin**

**lemma** *mcont-const* [*cont-intro, simp*]:

*mcont luba orda lub leq* ( $\lambda x. c$ )

*<proof>*

**lemmas** [*cont-intro, simp*] =

*ccpo.cont-const*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

**lemma** *mono2mono*:

**assumes** *monotone ordb leq* ( $\lambda y. f y$ ) *monotone orda ordb* ( $\lambda x. t x$ )

**shows** *monotone orda leq* ( $\lambda x. f (t x)$ )

*<proof>*

**lemmas** *mcont2mcont'* = *ccpo.mcont2mcont'*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

**lemmas** *mcont2mcont* = *ccpo.mcont2mcont*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

**lemmas** *fixp-preserves-mono1* = *ccpo.fixp-preserves-mono1* [*OF Partial-Function.ccpo* [*OF partial-function-definitions-axioms*]]  
**lemmas** *fixp-preserves-mono2* = *ccpo.fixp-preserves-mono2* [*OF Partial-Function.ccpo* [*OF partial-function-definitions-axioms*]]  
**lemmas** *fixp-preserves-mono3* = *ccpo.fixp-preserves-mono3* [*OF Partial-Function.ccpo* [*OF partial-function-definitions-axioms*]]  
**lemmas** *fixp-preserves-mono4* = *ccpo.fixp-preserves-mono4* [*OF Partial-Function.ccpo* [*OF partial-function-definitions-axioms*]]  
**lemmas** *fixp-preserves-mcont1* = *ccpo.fixp-preserves-mcont1* [*OF Partial-Function.ccpo* [*OF partial-function-definitions-axioms*]]  
**lemmas** *fixp-preserves-mcont2* = *ccpo.fixp-preserves-mcont2* [*OF Partial-Function.ccpo* [*OF partial-function-definitions-axioms*]]  
**lemmas** *fixp-preserves-mcont3* = *ccpo.fixp-preserves-mcont3* [*OF Partial-Function.ccpo* [*OF partial-function-definitions-axioms*]]  
**lemmas** *fixp-preserves-mcont4* = *ccpo.fixp-preserves-mcont4* [*OF Partial-Function.ccpo* [*OF partial-function-definitions-axioms*]]

**lemma** *monotone-if-bot*:

**fixes** *bot*  
**assumes** *g*:  $\bigwedge x. g\ x = (\text{if } \text{leq } x\ \text{bound} \text{ then } \text{bot} \text{ else } f\ x)$   
**and** *mono*:  $\bigwedge x\ y. \llbracket \text{leq } x\ y; \neg \text{leq } x\ \text{bound} \rrbracket \implies \text{ord } (f\ x)\ (f\ y)$   
**and** *bot*:  $\bigwedge x. \neg \text{leq } x\ \text{bound} \implies \text{ord } \text{bot}\ (f\ x)\ \text{ord } \text{bot}\ \text{bot}$   
**shows** *monotone leq ord g*  
*<proof>*

**lemma** *mcont-if-bot*:

**fixes** *bot*  
**assumes** *ccpo*: *class.ccpo lub' ord (mk-less ord)*  
**and** *bot*:  $\bigwedge x. \neg \text{leq } x\ \text{bound} \implies \text{ord } \text{bot}\ (f\ x)$   
**and** *g*:  $\bigwedge x. g\ x = (\text{if } \text{leq } x\ \text{bound} \text{ then } \text{bot} \text{ else } f\ x)$   
**and** *mono*:  $\bigwedge x\ y. \llbracket \text{leq } x\ y; \neg \text{leq } x\ \text{bound} \rrbracket \implies \text{ord } (f\ x)\ (f\ y)$   
**and** *cont*:  $\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain } \text{leq } Y; Y \neq \{\}; \bigwedge x. x \in Y \implies \neg \text{leq } x\ \text{bound} \rrbracket \implies f\ (\text{lub } Y) = \text{lub}'\ (f\ ` Y)$   
**shows** *mcont lub leq lub' ord g*  
*<proof>*

**end**

## 15.2 Admissibility

**lemma** *admissible-subst*:

**assumes** *adm*: *ccpo.admissible luba orda*  $(\lambda x. P\ x)$   
**and** *mcont*: *mcont lubb ordb luba orda f*  
**shows** *ccpo.admissible lubb ordb*  $(\lambda x. P\ (f\ x))$   
*<proof>*

**lemmas** [*simp, cont-intro*] =  
*admissible-all*  
*admissible-ball*

*admissible-const*  
*admissible-conj*

**lemma** *admissible-disj'* [*simp, cont-intro*]:  
 [[ *class.ccpo lub ord (mk-less ord)*; *ccpo.admissible lub ord P*; *ccpo.admissible lub ord Q* ]]  
 $\implies$  *ccpo.admissible lub ord* ( $\lambda x. P x \vee Q x$ )  
 ⟨*proof*⟩

**lemma** *admissible-imp'* [*cont-intro*]:  
 [[ *class.ccpo lub ord (mk-less ord)*;  
   *ccpo.admissible lub ord* ( $\lambda x. \neg P x$ );  
   *ccpo.admissible lub ord* ( $\lambda x. Q x$ ) ]]  
 $\implies$  *ccpo.admissible lub ord* ( $\lambda x. P x \longrightarrow Q x$ )  
 ⟨*proof*⟩

**lemma** *admissible-imp* [*cont-intro*]:  
 ( $Q \implies$  *ccpo.admissible lub ord* ( $\lambda x. P x$ ))  
 $\implies$  *ccpo.admissible lub ord* ( $\lambda x. Q \longrightarrow P x$ )  
 ⟨*proof*⟩

**lemma** *admissible-not-mem'* [*THEN admissible-subst, cont-intro, simp*]:  
**shows** *admissible-not-mem*: *ccpo.admissible Union* ( $\subseteq$ ) ( $\lambda A. x \notin A$ )  
 ⟨*proof*⟩

**lemma** *admissible-eqI*:  
**assumes** *f*: *cont luba orda lub ord* ( $\lambda x. f x$ )  
   **and** *g*: *cont luba orda lub ord* ( $\lambda x. g x$ )  
**shows** *ccpo.admissible luba orda* ( $\lambda x. f x = g x$ )  
 ⟨*proof*⟩

**corollary** *admissible-eq-mcontI* [*cont-intro*]:  
 [[ *mcont luba orda lub ord* ( $\lambda x. f x$ );  
   *mcont luba orda lub ord* ( $\lambda x. g x$ ) ]]  
 $\implies$  *ccpo.admissible luba orda* ( $\lambda x. f x = g x$ )  
 ⟨*proof*⟩

**lemma** *admissible-iff* [*cont-intro, simp*]:  
 [[ *ccpo.admissible lub ord* ( $\lambda x. P x \longrightarrow Q x$ ); *ccpo.admissible lub ord* ( $\lambda x. Q x \longrightarrow P x$ ) ]]  
 $\implies$  *ccpo.admissible lub ord* ( $\lambda x. P x \longleftrightarrow Q x$ )  
 ⟨*proof*⟩

**context** *ccpo begin*

**lemma** *admissible-leI*:  
**assumes** *f*: *mcont luba orda Sup* ( $\leq$ ) ( $\lambda x. f x$ )  
   **and** *g*: *mcont luba orda Sup* ( $\leq$ ) ( $\lambda x. g x$ )  
**shows** *ccpo.admissible luba orda* ( $\lambda x. f x \leq g x$ )

*<proof>*

**end**

**lemma** *admissible-leI*:

**fixes** *ord* (**infix**  $\sqsubseteq$  60) **and** *lub* ( $\sqcup$ )  
**assumes** *class.ccpo lub* ( $\sqsubseteq$ ) (*mk-less* ( $\sqsubseteq$ ))  
**and** *mcont luba orda lub* ( $\sqsubseteq$ ) ( $\lambda x. f x$ )  
**and** *mcont luba orda lub* ( $\sqsubseteq$ ) ( $\lambda x. g x$ )  
**shows** *ccpo.admissible luba orda* ( $\lambda x. f x \sqsubseteq g x$ )

*<proof>*

**declare** *ccpo-class.admissible-leI*[*cont-intro*]

**context** *ccpo begin*

**lemma** *admissible-not-below*: *ccpo.admissible Sup* ( $\leq$ ) ( $\lambda x. \neg (\leq) x y$ )

*<proof>*

**end**

**lemma** (**in** *preorder*) *preorder* [*cont-intro, simp*]: *class.preorder* ( $\leq$ ) (*mk-less* ( $\leq$ ))

*<proof>*

**context** *partial-function-definitions begin*

**lemmas** [*cont-intro, simp*] =

*admissible-leI*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]  
*ccpo.admissible-not-below*[*THEN admissible-subst, OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

**end**

*<ML>*

**inductive** *compact* :: (*'a set*  $\Rightarrow$  *'a*)  $\Rightarrow$  (*'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *bool*)  $\Rightarrow$  *'a*  $\Rightarrow$  *bool*

**for** *lub ord x*

**where** *compact*:

$\llbracket$  *ccpo.admissible lub ord* ( $\lambda y. \neg \text{ord } x y$ );  
*ccpo.admissible lub ord* ( $\lambda y. x \neq y$ )  $\rrbracket$   
 $\Rightarrow$  *compact lub ord x*

*<ML>*

**context** *ccpo begin*

**lemma** *compactI*:

**assumes** *ccpo.admissible Sup* ( $\leq$ ) ( $\lambda y. \neg x \leq y$ )  
**shows** *ccpo.compact Sup* ( $\leq$ ) *x*

*<proof>*

**lemma** *compact-bot*:

**assumes**  $x = \text{Sup } \{\}$

**shows**  $\text{ccpo.compact Sup } (\leq) x$

*<proof>*

**end**

**lemma** *admissible-compact-neq'* [*THEN* *admissible-subst*, *cont-intro*, *simp*]:

**shows** *admissible-compact-neq*:  $\text{ccpo.compact lub ord } k \implies \text{ccpo.admissible lub ord } (\lambda x. k \neq x)$

*<proof>*

**lemma** *admissible-neq-compact'* [*THEN* *admissible-subst*, *cont-intro*, *simp*]:

**shows** *admissible-neq-compact*:  $\text{ccpo.compact lub ord } k \implies \text{ccpo.admissible lub ord } (\lambda x. x \neq k)$

*<proof>*

**context** *partial-function-definitions* **begin**

**lemmas** [*cont-intro*, *simp*] = *ccpo.compact-bot*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

**end**

**context** *ccpo* **begin**

**lemma** *fixp-strong-induct*:

**assumes** [*cont-intro*]:  $\text{ccpo.admissible Sup } (\leq) P$

**and** *mono*:  $\text{monotone } (\leq) (\leq) f$

**and** *bot*:  $P (\bigsqcup \{\})$

**and** *step*:  $\bigwedge x. \llbracket x \leq \text{ccpo-class.fixp } f; P x \rrbracket \implies P (f x)$

**shows**  $P (\text{ccpo-class.fixp } f)$

*<proof>*

**end**

**context** *partial-function-definitions* **begin**

**lemma** *fixp-strong-induct-uc*:

**fixes**  $F :: 'c \Rightarrow 'c$

**and**  $U :: 'c \Rightarrow 'b \Rightarrow 'a$

**and**  $C :: ('b \Rightarrow 'a) \Rightarrow 'c$

**and**  $P :: ('b \Rightarrow 'a) \Rightarrow \text{bool}$

**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*:  $\text{ccpo.admissible lub-fun le-fun } P$



```

and bot: P (λ-. lub {})
and step: ∧f'. [ P (U f'); le-fun (U f') (U f) ] ⇒ P (U (F f'))
shows P (U f)
⟨proof⟩

```

**end**

### 15.3 (=) as order

```

definition lub-singleton :: ('a set ⇒ 'a) ⇒ bool
where lub-singleton lub ↔ (∀ a. lub {a} = a)

```

```

definition the-Sup :: 'a set ⇒ 'a
where the-Sup A = (THE a. a ∈ A)

```

```

lemma lub-singleton-the-Sup [cont-intro, simp]: lub-singleton the-Sup
⟨proof⟩

```

```

lemma (in ccpo) lub-singleton: lub-singleton Sup
⟨proof⟩

```

```

lemma (in partial-function-definitions) lub-singleton [cont-intro, simp]: lub-singleton
lub
⟨proof⟩

```

```

lemma preorder-eq [cont-intro, simp]:
class.preorder (=) (mk-less (=))
⟨proof⟩

```

```

lemma monotone-eqI [cont-intro]:
assumes class.preorder ord (mk-less ord)
shows monotone (=) ord f
⟨proof⟩

```

```

lemma cont-eqI [cont-intro]:
fixes f :: 'a ⇒ 'b
assumes lub-singleton lub
shows cont the-Sup (=) lub ord f
⟨proof⟩

```

```

lemma mcont-eqI [cont-intro, simp]:
[ class.preorder ord (mk-less ord); lub-singleton lub ]
⇒ mcont the-Sup (=) lub ord f
⟨proof⟩

```

### 15.4 ccpo for products

```

definition prod-lub :: ('a set ⇒ 'a) ⇒ ('b set ⇒ 'b) ⇒ ('a × 'b) set ⇒ 'a × 'b
where prod-lub Sup-a Sup-b Y = (Sup-a (fst ` Y), Sup-b (snd ` Y))

```

**lemma** *lub-singleton-prod-lub* [*cont-intro*, *simp*]:  
 $\llbracket \text{lub-singleton } \text{luba}; \text{lub-singleton } \text{lubb} \rrbracket \implies \text{lub-singleton } (\text{prod-lub } \text{luba } \text{lubb})$   
 ⟨*proof*⟩

**lemma** *prod-lub-empty* [*simp*]:  $\text{prod-lub } \text{luba } \text{lubb } \{\} = (\text{luba } \{\}, \text{lubb } \{\})$   
 ⟨*proof*⟩

**lemma** *preorder-rel-prodI* [*cont-intro*, *simp*]:  
**assumes** *class.preorder* *orda* (*mk-less orda*)  
**and** *class.preorder* *ordb* (*mk-less ordb*)  
**shows** *class.preorder* (*rel-prod orda ordb*) (*mk-less (rel-prod orda ordb)*)  
 ⟨*proof*⟩

**lemma** *order-rel-prodI*:  
**assumes** *a*: *class.order* *orda* (*mk-less orda*)  
**and** *b*: *class.order* *ordb* (*mk-less ordb*)  
**shows** *class.order* (*rel-prod orda ordb*) (*mk-less (rel-prod orda ordb)*)  
 (*is class.order ?ord ?ord^*)  
 ⟨*proof*⟩

**lemma** *monotone-rel-prodI*:  
**assumes** *mono2*:  $\bigwedge a. \text{monotone } \text{ordb } \text{ordc } (\lambda b. f (a, b))$   
**and** *mono1*:  $\bigwedge b. \text{monotone } \text{orda } \text{ordc } (\lambda a. f (a, b))$   
**and** *a*: *class.preorder* *orda* (*mk-less orda*)  
**and** *b*: *class.preorder* *ordb* (*mk-less ordb*)  
**and** *c*: *class.preorder* *ordc* (*mk-less ordc*)  
**shows** *monotone* (*rel-prod orda ordb*) *ordc f*  
 ⟨*proof*⟩

**lemma** *monotone-rel-prodD1*:  
**assumes** *mono*: *monotone* (*rel-prod orda ordb*) *ordc f*  
**and** *preorder*: *class.preorder* *ordb* (*mk-less ordb*)  
**shows** *monotone* *orda ordc* ( $\lambda a. f (a, b)$ )  
 ⟨*proof*⟩

**lemma** *monotone-rel-prodD2*:  
**assumes** *mono*: *monotone* (*rel-prod orda ordb*) *ordc f*  
**and** *preorder*: *class.preorder* *orda* (*mk-less orda*)  
**shows** *monotone* *ordb ordc* ( $\lambda b. f (a, b)$ )  
 ⟨*proof*⟩

**lemma** *monotone-case-prodI*:  
 $\llbracket \bigwedge a. \text{monotone } \text{ordb } \text{ordc } (f a); \bigwedge b. \text{monotone } \text{orda } \text{ordc } (\lambda a. f a b);$   
*class.preorder* *orda* (*mk-less orda*); *class.preorder* *ordb* (*mk-less ordb*);  
*class.preorder* *ordc* (*mk-less ordc*)  $\rrbracket$   
 $\implies \text{monotone } (\text{rel-prod } \text{orda } \text{ordb}) \text{ordc } (\text{case-prod } f)$   
 ⟨*proof*⟩

**lemma** *monotone-case-prodD1*:

**assumes** *mono*: *monotone* (*rel-prod* *orda* *ordb*) *ordc* (*case-prod* *f*)  
**and** *preorder*: *class.preorder* *ordb* (*mk-less* *ordb*)  
**shows** *monotone* *orda* *ordc* ( $\lambda a. f a b$ )  
 $\langle$ *proof* $\rangle$

**lemma** *monotone-case-prodD2*:

**assumes** *mono*: *monotone* (*rel-prod* *orda* *ordb*) *ordc* (*case-prod* *f*)  
**and** *preorder*: *class.preorder* *orda* (*mk-less* *orda*)  
**shows** *monotone* *ordb* *ordc* (*f* *a*)  
 $\langle$ *proof* $\rangle$

**context**

**fixes** *orda* *ordb* *ordc*  
**assumes** *a*: *class.preorder* *orda* (*mk-less* *orda*)  
**and** *b*: *class.preorder* *ordb* (*mk-less* *ordb*)  
**and** *c*: *class.preorder* *ordc* (*mk-less* *ordc*)

**begin**

**lemma** *monotone-rel-prod-iff*:

*monotone* (*rel-prod* *orda* *ordb*) *ordc* *f*  $\longleftrightarrow$   
 $(\forall a. \textit{monotone} \textit{ordb} \textit{ordc} (\lambda b. f (a, b))) \wedge$   
 $(\forall b. \textit{monotone} \textit{orda} \textit{ordc} (\lambda a. f (a, b)))$   
 $\langle$ *proof* $\rangle$

**lemma** *monotone-case-prod-iff* [*simp*]:

*monotone* (*rel-prod* *orda* *ordb*) *ordc* (*case-prod* *f*)  $\longleftrightarrow$   
 $(\forall a. \textit{monotone} \textit{ordb} \textit{ordc} (f a)) \wedge (\forall b. \textit{monotone} \textit{orda} \textit{ordc} (\lambda a. f a b))$   
 $\langle$ *proof* $\rangle$

**end**

**lemma** *monotone-case-prod-apply-iff*:

*monotone* *orda* *ordb* ( $\lambda x. (\textit{case-prod} f x) y$ )  $\longleftrightarrow$  *monotone* *orda* *ordb* (*case-prod*  
 $(\lambda a b. f a b y)$ )  
 $\langle$ *proof* $\rangle$

**lemma** *monotone-case-prod-applyD*:

*monotone* *orda* *ordb* ( $\lambda x. (\textit{case-prod} f x) y$ )  
 $\implies$  *monotone* *orda* *ordb* (*case-prod* ( $\lambda a b. f a b y$ ))  
 $\langle$ *proof* $\rangle$

**lemma** *monotone-case-prod-applyI*:

*monotone* *orda* *ordb* (*case-prod* ( $\lambda a b. f a b y$ ))  
 $\implies$  *monotone* *orda* *ordb* ( $\lambda x. (\textit{case-prod} f x) y$ )  
 $\langle$ *proof* $\rangle$

**lemma** *cont-case-prod-apply-iff*:

*cont luba* *orda* *lubb* *ordb* ( $\lambda x. (\textit{case-prod} f x) y$ )  $\longleftrightarrow$  *cont luba* *orda* *lubb* *ordb*

(*case-prod* ( $\lambda a b. f a b y$ ))  
 ⟨*proof*⟩

**lemma** *cont-case-prod-applyI*:

*cont luba orda lubb ordb* (*case-prod* ( $\lambda a b. f a b y$ ))  
 $\implies$  *cont luba orda lubb ordb* ( $\lambda x. (\text{case-prod } f x) y$ )  
 ⟨*proof*⟩

**lemma** *cont-case-prod-applyD*:

*cont luba orda lubb ordb* ( $\lambda x. (\text{case-prod } f x) y$ )  
 $\implies$  *cont luba orda lubb ordb* (*case-prod* ( $\lambda a b. f a b y$ ))  
 ⟨*proof*⟩

**lemma** *mcont-case-prod-apply-iff* [*simp*]:

*mcont luba orda lubb ordb* ( $\lambda x. (\text{case-prod } f x) y$ )  $\longleftrightarrow$   
*mcont luba orda lubb ordb* (*case-prod* ( $\lambda a b. f a b y$ ))  
 ⟨*proof*⟩

**lemma** *cont-prodD1*:

**assumes** *cont*: *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lucb ordc f*  
**and** *class.preorder orda* (*mk-less orda*)  
**and** *luba*: *lub-singleton luba*  
**shows** *cont lubb ordb lucb ordc* ( $\lambda y. f (x, y)$ )  
 ⟨*proof*⟩

**lemma** *cont-prodD2*:

**assumes** *cont*: *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lucb ordc f*  
**and** *class.preorder ordb* (*mk-less ordb*)  
**and** *lubb*: *lub-singleton lubb*  
**shows** *cont luba orda lucb ordc* ( $\lambda x. f (x, y)$ )  
 ⟨*proof*⟩

**lemma** *cont-case-prodD1*:

**assumes** *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lucb ordc* (*case-prod f*)  
**and** *class.preorder orda* (*mk-less orda*)  
**and** *lub-singleton luba*  
**shows** *cont lubb ordb lucb ordc* (*f x*)  
 ⟨*proof*⟩

**lemma** *cont-case-prodD2*:

**assumes** *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lucb ordc* (*case-prod f*)  
**and** *class.preorder ordb* (*mk-less ordb*)  
**and** *lub-singleton lubb*  
**shows** *cont luba orda lucb ordc* ( $\lambda x. f x y$ )  
 ⟨*proof*⟩

**context** *ccpo* **begin**

**lemma** *cont-prodI*:

**assumes** *mono*: *monotone* (*rel-prod orda ordb*) ( $\leq$ ) *f*  
**and** *cont1*:  $\bigwedge x. \text{cont lubb ordb Sup } (\leq) (\lambda y. f (x, y))$   
**and** *cont2*:  $\bigwedge y. \text{cont luba orda Sup } (\leq) (\lambda x. f (x, y))$   
**and** *class.preorder orda* (*mk-less orda*)  
**and** *class.preorder ordb* (*mk-less ordb*)  
**shows** *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup* ( $\leq$ ) *f*  
 <proof>

**lemma** *cont-case-prodI*:

**assumes** *monotone* (*rel-prod orda ordb*) ( $\leq$ ) (*case-prod f*)  
**and**  $\bigwedge x. \text{cont lubb ordb Sup } (\leq) (\lambda y. f x y)$   
**and**  $\bigwedge y. \text{cont luba orda Sup } (\leq) (\lambda x. f x y)$   
**and** *class.preorder orda* (*mk-less orda*)  
**and** *class.preorder ordb* (*mk-less ordb*)  
**shows** *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup* ( $\leq$ ) (*case-prod f*)  
 <proof>

**lemma** *cont-case-prod-iff*:

[ *monotone* (*rel-prod orda ordb*) ( $\leq$ ) (*case-prod f*);  
   *class.preorder orda* (*mk-less orda*); *lub-singleton luba*;  
   *class.preorder ordb* (*mk-less ordb*); *lub-singleton lubb* ]  
 $\implies \text{cont } (\text{prod-lub luba lubb}) (\text{rel-prod orda ordb}) \text{Sup } (\leq) (\text{case-prod } f) \longleftrightarrow$   
 $(\forall x. \text{cont lubb ordb Sup } (\leq) (\lambda y. f x y)) \wedge (\forall y. \text{cont luba orda Sup } (\leq) (\lambda x. f x$   
*y*)  
 <proof>

**end**

**context** *partial-function-definitions* **begin**

**lemma** *mono2mono2*:

**assumes** *f*: *monotone* (*rel-prod ordb ordc*) *leq* ( $\lambda(x, y). f x y$ )  
**and** *t*: *monotone orda ordb* ( $\lambda x. t x$ )  
**and** *t'*: *monotone orda ordc* ( $\lambda x. t' x$ )  
**shows** *monotone orda leq* ( $\lambda x. f (t x) (t' x)$ )  
 <proof>

**lemma** *cont-case-prodI* [*cont-intro*]:

[ *monotone* (*rel-prod orda ordb*) *leq* (*case-prod f*);  
    $\bigwedge x. \text{cont lubb ordb lub leq } (\lambda y. f x y)$ ;  
    $\bigwedge y. \text{cont luba orda lub leq } (\lambda x. f x y)$ ;  
   *class.preorder orda* (*mk-less orda*);  
   *class.preorder ordb* (*mk-less ordb*) ]  
 $\implies \text{cont } (\text{prod-lub luba lubb}) (\text{rel-prod orda ordb}) \text{lub leq } (\text{case-prod } f)$   
 <proof>

**lemma** *cont-case-prod-iff*:

[ *monotone* (*rel-prod orda ordb*) *leq* (*case-prod f*);  
   *class.preorder orda* (*mk-less orda*); *lub-singleton luba*;

```

class.preorder ordb (mk-less ordb); lub-singleton lubb ]
 $\implies$  cont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod f)  $\longleftrightarrow$ 
 $(\forall x. \text{cont lubb ordb lub leq } (\lambda y. f x y)) \wedge (\forall y. \text{cont luba orda lub leq } (\lambda x. f x y))$ 
<proof>

```

**lemma** *mcont-case-prod-iff* [simp]:

```

[ class.preorder orda (mk-less orda); lub-singleton luba;
  class.preorder ordb (mk-less ordb); lub-singleton lubb ]
 $\implies$  mcont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod f)  $\longleftrightarrow$ 
 $(\forall x. \text{mcont lubb ordb lub leq } (\lambda y. f x y)) \wedge (\forall y. \text{mcont luba orda lub leq } (\lambda x. f x y))$ 
<proof>

```

**end**

**lemma** *mono2mono-case-prod* [cont-intro]:

```

assumes  $\bigwedge x y. \text{monotone orda ordb } (\lambda f. \text{pair } f x y)$ 
shows monotone orda ordb  $(\lambda f. \text{case-prod } (\text{pair } f) x)$ 
<proof>

```

## 15.5 Complete lattices as ccpo

**context** *complete-lattice* **begin**

**lemma** *complete-lattice-ccpo*: class.ccpo Sup  $(\leq)$   $(<)$   
<proof>

**lemma** *complete-lattice-ccpo'*: class.ccpo Sup  $(\leq)$  (mk-less  $(\leq)$ )  
<proof>

**lemma** *complete-lattice-partial-function-definitions*:

```

partial-function-definitions  $(\leq)$  Sup
<proof>

```

**lemma** *complete-lattice-partial-function-definitions-dual*:

```

partial-function-definitions  $(\geq)$  Inf
<proof>

```

**lemmas** [cont-intro, simp] =

```

Partial-Function.ccpo[OF complete-lattice-partial-function-definitions]
Partial-Function.ccpo[OF complete-lattice-partial-function-definitions-dual]

```

**lemma** *mono2mono-inf*:

```

assumes f: monotone ord  $(\leq)$   $(\lambda x. f x)$ 
and g: monotone ord  $(\leq)$   $(\lambda x. g x)$ 
shows monotone ord  $(\leq)$   $(\lambda x. f x \sqcap g x)$ 
<proof>

```

**lemma** *mcont-const* [simp]: mcont lub ord Sup  $(\leq)$   $(\lambda-. c)$

*<proof>*

**lemma** *mono2mono-sup*:

**assumes** *f*: *monotone ord* ( $\leq$ ) ( $\lambda x. f x$ )  
**and** *g*: *monotone ord* ( $\leq$ ) ( $\lambda x. g x$ )  
**shows** *monotone ord* ( $\leq$ ) ( $\lambda x. f x \sqcup g x$ )  
*<proof>*

**lemma** *Sup-image-sup*:

**assumes**  $Y \neq \{\}$   
**shows**  $\sqcup ((\sqcup) x \text{ ‘ } Y) = x \sqcup \sqcup Y$   
*<proof>*

**lemma** *mcont-sup1*: *mcont Sup* ( $\leq$ ) *Sup* ( $\leq$ ) ( $\lambda y. x \sqcup y$ )  
*<proof>*

**lemma** *mcont-sup2*: *mcont Sup* ( $\leq$ ) *Sup* ( $\leq$ ) ( $\lambda x. x \sqcup y$ )  
*<proof>*

**lemma** *mcont2mcont-sup* [*cont-intro*, *simp*]:

$\llbracket$  *mcont lub ord Sup* ( $\leq$ ) ( $\lambda x. f x$ );  
*mcont lub ord Sup* ( $\leq$ ) ( $\lambda x. g x$ )  $\rrbracket$   
 $\implies$  *mcont lub ord Sup* ( $\leq$ ) ( $\lambda x. f x \sqcup g x$ )  
*<proof>*

**end**

**lemmas** [*cont-intro*] = *admissible-leI*[*OF complete-lattice-ccpo*]

**context** *complete-distrib-lattice* **begin**

**lemma** *mcont-inf1*: *mcont Sup* ( $\leq$ ) *Sup* ( $\leq$ ) ( $\lambda y. x \sqcap y$ )  
*<proof>*

**lemma** *mcont-inf2*: *mcont Sup* ( $\leq$ ) *Sup* ( $\leq$ ) ( $\lambda x. x \sqcap y$ )  
*<proof>*

**lemma** *mcont2mcont-inf* [*cont-intro*, *simp*]:

$\llbracket$  *mcont lub ord Sup* ( $\leq$ ) ( $\lambda x. f x$ );  
*mcont lub ord Sup* ( $\leq$ ) ( $\lambda x. g x$ )  $\rrbracket$   
 $\implies$  *mcont lub ord Sup* ( $\leq$ ) ( $\lambda x. f x \sqcap g x$ )  
*<proof>*

**end**

**interpretation** *lfp*: *partial-function-definitions* ( $\leq$ ) :: - :: *complete-lattice*  $\implies$  - *Sup*  
*<proof>*

*<ML>*

**interpretation** *gfp: partial-function-definitions*  $(\geq) :: - :: \text{complete-lattice} \Rightarrow - \text{Inf}$   
 ⟨proof⟩

⟨ML⟩

**lemma** *insert-mono* [*partial-function-mono*]:  
 $\text{monotone } (\text{fun-ord } (\subseteq)) (\subseteq) A \implies \text{monotone } (\text{fun-ord } (\subseteq)) (\subseteq) (\lambda y. \text{insert } x (A y))$   
 ⟨proof⟩

**lemma** *mono2mono-insert* [*THEN lfp.mono2mono, cont-intro, simp*]:  
**shows** *monotone-insert*:  $\text{monotone } (\subseteq) (\subseteq) (\text{insert } x)$   
 ⟨proof⟩

**lemma** *mcont2mcont-insert* [*THEN lfp.mcont2mcont, cont-intro, simp*]:  
**shows** *mcont-insert*:  $\text{mcont } \text{Union } (\subseteq) \text{Union } (\subseteq) (\text{insert } x)$   
 ⟨proof⟩

**lemma** *mono2mono-image* [*THEN lfp.mono2mono, cont-intro, simp*]:  
**shows** *monotone-image*:  $\text{monotone } (\subseteq) (\subseteq) ((\cdot) f)$   
 ⟨proof⟩

**lemma** *cont-image*:  $\text{cont } \text{Union } (\subseteq) \text{Union } (\subseteq) ((\cdot) f)$   
 ⟨proof⟩

**lemma** *mcont2mcont-image* [*THEN lfp.mcont2mcont, cont-intro, simp*]:  
**shows** *mcont-image*:  $\text{mcont } \text{Union } (\subseteq) \text{Union } (\subseteq) ((\cdot) f)$   
 ⟨proof⟩

**context** *complete-lattice* **begin**

**lemma** *monotone-Sup* [*cont-intro, simp*]:  
 $\text{monotone ord } (\subseteq) f \implies \text{monotone ord } (\leq) (\lambda x. \bigsqcup f x)$   
 ⟨proof⟩

**lemma** *cont-Sup*:  
**assumes** *cont lub ord Union*  $(\subseteq) f$   
**shows** *cont lub ord Sup*  $(\leq) (\lambda x. \bigsqcup f x)$   
 ⟨proof⟩

**lemma** *mcont-Sup*:  $\text{mcont lub ord Union } (\subseteq) f \implies \text{mcont lub ord Sup } (\leq) (\lambda x. \bigsqcup f x)$   
 ⟨proof⟩

**lemma** *monotone-SUP*:  
 $\llbracket \text{monotone ord } (\subseteq) f; \bigwedge y. \text{monotone ord } (\leq) (\lambda x. g x y) \rrbracket \implies \text{monotone ord } (\leq) (\lambda x. \bigsqcup_{y \in f x} g x y)$   
 ⟨proof⟩



**lemma** *monotone-SUP2*:

$(\bigwedge y. y \in A \implies \text{monotone ord } (\leq) (\lambda x. g x y)) \implies \text{monotone ord } (\leq) (\lambda x. \bigsqcup_{y \in A}. g x y)$   
 ⟨proof⟩

**lemma** *cont-SUP*:

**assumes**  $f: \text{mcont lub ord Union } (\subseteq) f$   
**and**  $g: \bigwedge y. \text{mcont lub ord Sup } (\leq) (\lambda x. g x y)$   
**shows**  $\text{cont lub ord Sup } (\leq) (\lambda x. \bigsqcup_{y \in f x}. g x y)$   
 ⟨proof⟩

**lemma** *mcont-SUP* [*cont-intro, simp*]:

$\llbracket \text{mcont lub ord Union } (\subseteq) f; \bigwedge y. \text{mcont lub ord Sup } (\leq) (\lambda x. g x y) \rrbracket$   
 $\implies \text{mcont lub ord Sup } (\leq) (\lambda x. \bigsqcup_{y \in f x}. g x y)$   
 ⟨proof⟩

**end**

**lemma** *admissible-Ball* [*cont-intro, simp*]:

$\llbracket \bigwedge x. \text{ccpo.admissible lub ord } (\lambda A. P A x);$   
 $\text{mcont lub ord Union } (\subseteq) f;$   
 $\text{class.ccpo lub ord } (\text{mk-less ord}) \rrbracket$   
 $\implies \text{ccpo.admissible lub ord } (\lambda A. \forall x \in f A. P A x)$   
 ⟨proof⟩

**lemma** *admissible-Bex*'[*THEN admissible-subst, cont-intro, simp*]:

**shows** *admissible-Bex*:  $\text{ccpo.admissible Union } (\subseteq) (\lambda A. \exists x \in A. P x)$   
 ⟨proof⟩

## 15.6 Parallel fixpoint induction

**context**

**fixes**  $\text{luba} :: 'a \text{ set} \Rightarrow 'a$   
**and**  $\text{orda} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$   
**and**  $\text{lubb} :: 'b \text{ set} \Rightarrow 'b$   
**and**  $\text{ordb} :: 'b \Rightarrow 'b \Rightarrow \text{bool}$   
**assumes**  $a: \text{class.ccpo luba orda } (\text{mk-less orda})$   
**and**  $b: \text{class.ccpo lubb ordb } (\text{mk-less ordb})$

**begin**

**interpretation**  $a: \text{ccpo luba orda mk-less orda}$  ⟨proof⟩

**interpretation**  $b: \text{ccpo lubb ordb mk-less ordb}$  ⟨proof⟩

**lemma** *ccpo-rel-prodI*:

$\text{class.ccpo } (\text{prod-lub luba lubb}) (\text{rel-prod orda ordb}) (\text{mk-less } (\text{rel-prod orda ordb}))$   
 (is  $\text{class.ccpo ?lub ?ord ?ord'}$ )  
 ⟨proof⟩

**interpretation** *ab*: *ccpo prod-lub luba lubb rel-prod orda ordb mk-less (rel-prod orda ordb)*  
 ⟨*proof*⟩

**lemma** *monotone-map-prod* [*simp*]:  
*monotone (rel-prod orda ordb) (rel-prod ordc ordd) (map-prod f g)  $\longleftrightarrow$*   
*monotone orda ordc f  $\wedge$  monotone ordb ordd g*  
 ⟨*proof*⟩

**lemma** *parallel-fixp-induct*:  
**assumes** *adm*: *ccpo.admissible (prod-lub luba lubb) (rel-prod orda ordb) ( $\lambda x. P$  (fst x) (snd x))*  
**and** *f*: *monotone orda orda f*  
**and** *g*: *monotone ordb ordb g*  
**and** *bot*: *P (luba {}) (lubb {})*  
**and** *step*:  $\bigwedge x y. P x y \implies P (f x) (g y)$   
**shows** *P (ccpo.fixp luba orda f) (ccpo.fixp lubb ordb g)*  
 ⟨*proof*⟩

**end**

**lemma** *parallel-fixp-induct-uc*:  
**assumes** *a*: *partial-function-definitions orda luba*  
**and** *b*: *partial-function-definitions ordb lubb*  
**and** *F*:  $\bigwedge x. \text{monotone (fun-ord orda) orda } (\lambda f. U1 (F (C1 f)) x)$   
**and** *G*:  $\bigwedge y. \text{monotone (fun-ord ordb) ordb } (\lambda g. U2 (G (C2 g)) y)$   
**and** *eq1*:  $f \equiv C1 (\text{ccpo.fixp (fun-lub luba) (fun-ord orda) } (\lambda f. U1 (F (C1 f))))$   
**and** *eq2*:  $g \equiv C2 (\text{ccpo.fixp (fun-lub lubb) (fun-ord ordb) } (\lambda g. U2 (G (C2 g))))$   
**and** *inverse*:  $\bigwedge f. U1 (C1 f) = f$   
**and** *inverse2*:  $\bigwedge g. U2 (C2 g) = g$   
**and** *adm*: *ccpo.admissible (prod-lub (fun-lub luba) (fun-lub lubb)) (rel-prod (fun-ord orda) (fun-ord ordb)) ( $\lambda x. P$  (fst x) (snd x))*  
**and** *bot*: *P ( $\lambda \cdot. luba \{\}$ ) ( $\lambda \cdot. lubb \{\}$ )*  
**and** *step*:  $\bigwedge f g. P (U1 f) (U2 g) \implies P (U1 (F f)) (U2 (G g))$   
**shows** *P (U1 f) (U2 g)*  
 ⟨*proof*⟩

**lemmas** *parallel-fixp-induct-1-1 = parallel-fixp-induct-uc*[  
*of - - -  $\lambda x. x - \lambda x. x \lambda x. x - \lambda x. x,$*   
*OF - - - - - refl refl]*

**lemmas** *parallel-fixp-induct-2-2 = parallel-fixp-induct-uc*[  
*of - - - case-prod - curry case-prod - curry,*  
**where** *P =  $\lambda f g. P$  (curry f) (curry g),*  
*unfolded case-prod-curry curry-case-prod curry-K,*  
*OF - - - - - refl refl]*  
**for** *P*

**lemma** *monotone-fst*: *monotone (rel-prod orda ordb) orda fst*

$\langle proof \rangle$

**lemma** *mcont-fst*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *luba orda fst*  
 $\langle proof \rangle$

**lemma** *mcont2mcont-fst* [*cont-intro, simp*]:  
*mcont lub ord* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *t*  
 $\implies$  *mcont lub ord luba orda* ( $\lambda x. \text{fst } (t x)$ )  
 $\langle proof \rangle$

**lemma** *monotone-snd*: *monotone* (*rel-prod orda ordb*) *ordb snd*  
 $\langle proof \rangle$

**lemma** *mcont-snd*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubb ordb snd*  
 $\langle proof \rangle$

**lemma** *mcont2mcont-snd* [*cont-intro, simp*]:  
*mcont lub ord* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *t*  
 $\implies$  *mcont lub ord lubb ordb* ( $\lambda x. \text{snd } (t x)$ )  
 $\langle proof \rangle$

**lemma** *monotone-Pair*:  
 $\llbracket \text{monotone ord orda } f; \text{monotone ord ordb } g \rrbracket$   
 $\implies$  *monotone ord* (*rel-prod orda ordb*) ( $\lambda x. (f x, g x)$ )  
 $\langle proof \rangle$

**lemma** *cont-Pair*:  
 $\llbracket \text{cont lub ord luba orda } f; \text{cont lub ord lubb ordb } g \rrbracket$   
 $\implies$  *cont lub ord* (*prod-lub luba lubb*) (*rel-prod orda ordb*) ( $\lambda x. (f x, g x)$ )  
 $\langle proof \rangle$

**lemma** *mcont-Pair*:  
 $\llbracket \text{mcont lub ord luba orda } f; \text{mcont lub ord lubb ordb } g \rrbracket$   
 $\implies$  *mcont lub ord* (*prod-lub luba lubb*) (*rel-prod orda ordb*) ( $\lambda x. (f x, g x)$ )  
 $\langle proof \rangle$

**context** *partial-function-definitions*  
**begin**

Specialised versions of *mcont-call* for admissibility proofs for parallel  
fixpoint inductions

**lemmas** *mcont-call-fst* [*cont-intro*] = *mcont-call*[*THEN mcont2mcont, OF mcont-fst*]  
**lemmas** *mcont-call-snd* [*cont-intro*] = *mcont-call*[*THEN mcont2mcont, OF mcont-snd*]  
**end**

**lemma** *map-option-mono* [*partial-function-mono*]:  
*mono-option B*  $\implies$  *mono-option* ( $\lambda f. \text{map-option } g (B f)$ )  
 $\langle proof \rangle$

**lemma** *compact-flat-lub* [*cont-intro*]: *ccpo.compact* (*flat-lub* *x*) (*flat-ord* *x*) *y*  
 ⟨*proof*⟩

**end**

**theory** *Conditional-Parametricity*  
**imports** *Main*  
**keywords** *parametric-constant* :: *thy-decl*  
**begin**

**context includes** *lifting-syntax* **begin**

**qualified definition** *Rel-match* :: (*'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *bool*)  $\Rightarrow$  *'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *bool* **where**  
*Rel-match* *R* *x* *y* = *R* *x* *y*

**named-theorems** *parametricity-preprocess*

**lemma** *bi-unique-Rel-match* [*parametricity-preprocess*]:  
*bi-unique* *A* = *Rel-match* (*A*  $\Longrightarrow$  *A*  $\Longrightarrow$  (=)) (=) (=)  
 ⟨*proof*⟩

**lemma** *bi-total-Rel-match* [*parametricity-preprocess*]:  
*bi-total* *A* = *Rel-match* ((*A*  $\Longrightarrow$  (=))  $\Longrightarrow$  (=)) *All* *All*  
 ⟨*proof*⟩

**lemma** *is-equality-Rel*: *is-equality* *A*  $\Longrightarrow$  *Transfer.Rel* *A* *t* *t*  
 ⟨*proof*⟩

**lemma** *Rel-Rel-match*: *Transfer.Rel* *R* *x* *y*  $\Longrightarrow$  *Rel-match* *R* *x* *y*  
 ⟨*proof*⟩

**lemma** *Rel-match-Rel*: *Rel-match* *R* *x* *y*  $\Longrightarrow$  *Transfer.Rel* *R* *x* *y*  
 ⟨*proof*⟩

**lemma** *Rel-Rel-match-eq*: *Transfer.Rel* *R* *x* *y* = *Rel-match* *R* *x* *y*  
 ⟨*proof*⟩

**lemma** *Rel-match-app*:  
**assumes** *Rel-match* (*A*  $\Longrightarrow$  *B*) *f* *g* **and** *Transfer.Rel* *A* *x* *y*  
**shows** *Rel-match* *B* (*f* *x*) (*g* *y*)  
 ⟨*proof*⟩

**end**

⟨*ML*⟩

**end**  
**theory** *Confluence* **imports**

*Main*  
begin

## 16 Confluence

**definition** *semiconfluentp* :: ('a ⇒ 'a ⇒ bool) ⇒ bool **where**  
*semiconfluentp* r ↔ r<sup>-1-1</sup> OO r<sup>\*\*</sup> ≤ r<sup>\*\*</sup> OO r<sup>-1-1\*\*</sup>

**definition** *confluentp* :: ('a ⇒ 'a ⇒ bool) ⇒ bool **where**  
*confluentp* r ↔ r<sup>-1-1\*\*</sup> OO r<sup>\*\*</sup> ≤ r<sup>\*\*</sup> OO r<sup>-1-1\*\*</sup>

**definition** *strong-confluentp* :: ('a ⇒ 'a ⇒ bool) ⇒ bool **where**  
*strong-confluentp* r ↔ r<sup>-1-1</sup> OO r ≤ r<sup>\*\*</sup> OO (r<sup>-1-1</sup>)<sup>=</sup>

**lemma** *semiconfluentpI* [intro?]:  
*semiconfluentp* r **if** ∧x y z. [ r x y; r<sup>\*\*</sup> x z ] ⇒ ∃ u. r<sup>\*\*</sup> y u ∧ r<sup>\*\*</sup> z u  
 ⟨proof⟩

**lemma** *semiconfluentpD*: ∃ u. r<sup>\*\*</sup> y u ∧ r<sup>\*\*</sup> z u **if** *semiconfluentp* r r x y r<sup>\*\*</sup> x z  
 ⟨proof⟩

**lemma** *confluentpI*:  
*confluentp* r **if** ∧x y z. [ r<sup>\*\*</sup> x y; r<sup>\*\*</sup> x z ] ⇒ ∃ u. r<sup>\*\*</sup> y u ∧ r<sup>\*\*</sup> z u  
 ⟨proof⟩

**lemma** *confluentpD*: ∃ u. r<sup>\*\*</sup> y u ∧ r<sup>\*\*</sup> z u **if** *confluentp* r r<sup>\*\*</sup> x y r<sup>\*\*</sup> x z  
 ⟨proof⟩

**lemma** *strong-confluentpI* [intro?]:  
*strong-confluentp* r **if** ∧x y z. [ r x y; r x z ] ⇒ ∃ u. r<sup>\*\*</sup> y u ∧ r<sup>=</sup> z u  
 ⟨proof⟩

**lemma** *strong-confluentpD*: ∃ u. r<sup>\*\*</sup> y u ∧ r<sup>=</sup> z u **if** *strong-confluentp* r r x y r x z  
 ⟨proof⟩

**lemma** *semiconfluentp-imp-confluentp*: *confluentp* r **if** r: *semiconfluentp* r  
 ⟨proof⟩

**lemma** *confluentp-imp-semiconfluentp*: *semiconfluentp* r **if** *confluentp* r  
 ⟨proof⟩

**lemma** *confluentp-eq-semiconfluentp*: *confluentp* r ↔ *semiconfluentp* r  
 ⟨proof⟩

**lemma** *confluentp-conv-strong-confluentp-rtranclp*:  
*confluentp* r ↔ *strong-confluentp* (r<sup>\*\*</sup>)  
 ⟨proof⟩

**lemma** *strong-confluentp-into-semiconfluentp*:  
*semiconfluentp r* **if** *r*: *strong-confluentp r*  
 ⟨*proof*⟩

**lemma** *strong-confluentp-imp-confluentp*: *confluentp r* **if** *strong-confluentp r*  
 ⟨*proof*⟩

**lemma** *semiconfluentp-equivclp*: *equivclp r = r\*\* OO r<sup>-1-1\*\*</sup>* **if** *r*: *semiconfluentp r*  
 ⟨*proof*⟩

**end**

**theory** *Confluent-Quotient* **imports**

*Confluence*

**begin**

Functors with finite setters preserve wide intersection for any equivalence relation that respects the mapper.

**lemma** *Inter-finite-subset*:  
**assumes**  $\forall A \in \mathcal{A}. \text{finite } A$   
**shows**  $\exists \mathcal{B} \subseteq \mathcal{A}. \text{finite } \mathcal{B} \wedge (\bigcap \mathcal{B}) = (\bigcap \mathcal{A})$   
 ⟨*proof*⟩

**locale** *wide-intersection-finite* =  
**fixes** *E* ::  $'Fa \Rightarrow 'Fa \Rightarrow \text{bool}$   
**and** *mapFa* ::  $('a \Rightarrow 'a) \Rightarrow 'Fa \Rightarrow 'Fa$   
**and** *setFa* ::  $'Fa \Rightarrow 'a \text{ set}$   
**assumes** *equiv*: *equivp E*  
**and** *map-E*:  $E \ x \ y \Longrightarrow E \ (\text{mapFa } f \ x) \ (\text{mapFa } f \ y)$   
**and** *map-id*:  $\text{mapFa } \text{id} \ x = x$   
**and** *map-cong*:  $\forall a \in \text{setFa } x. f \ a = g \ a \Longrightarrow \text{mapFa } f \ x = \text{mapFa } g \ x$   
**and** *set-map*:  $\text{setFa} \ (\text{mapFa } f \ x) = f \ ` \ \text{setFa } x$   
**and** *finite*: *finite (setFa x)*

**begin**

**lemma** *binary-intersection*:  
**assumes** *E* *y z* **and** *y*:  $\text{setFa } y \subseteq Y$  **and** *z*:  $\text{setFa } z \subseteq Z$  **and** *a*:  $a \in Y \ a \in Z$   
**shows**  $\exists x. E \ x \ y \wedge \text{setFa } x \subseteq Y \wedge \text{setFa } x \subseteq Z$   
 ⟨*proof*⟩

**lemma** *finite-intersection*:  
**assumes** *E*:  $\forall y \in A. E \ y \ z$   
**and** *fin*: *finite A*  
**and** *sub*:  $\forall y \in A. \text{setFa } y \subseteq Y \ y \wedge a \in Y \ y$   
**shows**  $\exists x. E \ x \ z \wedge (\forall y \in A. \text{setFa } x \subseteq Y \ y)$   
 ⟨*proof*⟩

**lemma** *wide-intersection*:  
**assumes** *inter-nonempty*:  $\bigcap Ss \neq \{\}$

**shows**  $(\bigcap As \in Ss. \{(x, x'). E x x'\} \text{ “ } \{x. setFa x \subseteq As\} \subseteq \{(x, x'). E x x'\} \text{ “ } \{x. setFa x \subseteq \bigcap Ss\} \text{ (is ?lhs } \subseteq \text{ ?rhs)}$   
 ⟨proof⟩

**end**

Subdistributivity for quotients via confluence

**lemma** *rtranclp-transp-reflp*:  $R^{**} = R$  if *transp R reflp R*  
 ⟨proof⟩

**lemma** *rtranclp-equivp*:  $R^{**} = R$  if *equivp R*  
 ⟨proof⟩

**locale** *confluent-quotient* =

**fixes**  $Rb :: 'Fb \Rightarrow 'Fb \Rightarrow bool$   
**and**  $Ea :: 'Fa \Rightarrow 'Fa \Rightarrow bool$   
**and**  $Eb :: 'Fb \Rightarrow 'Fb \Rightarrow bool$   
**and**  $Ec :: 'Fc \Rightarrow 'Fc \Rightarrow bool$   
**and**  $Eab :: 'Fab \Rightarrow 'Fab \Rightarrow bool$   
**and**  $Ebc :: 'Fbc \Rightarrow 'Fbc \Rightarrow bool$   
**and**  $\pi\text{-Faba} :: 'Fab \Rightarrow 'Fa$   
**and**  $\pi\text{-Fabb} :: 'Fab \Rightarrow 'Fb$   
**and**  $\pi\text{-Fbcb} :: 'Fbc \Rightarrow 'Fb$   
**and**  $\pi\text{-Fbcc} :: 'Fbc \Rightarrow 'Fc$   
**and**  $rel\text{-Fab} :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'Fa \Rightarrow 'Fb \Rightarrow bool$   
**and**  $rel\text{-Fbc} :: ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow 'Fb \Rightarrow 'Fc \Rightarrow bool$   
**and**  $rel\text{-Fac} :: ('a \Rightarrow 'c \Rightarrow bool) \Rightarrow 'Fa \Rightarrow 'Fc \Rightarrow bool$   
**and**  $set\text{-Fab} :: 'Fab \Rightarrow ('a \times 'b) \text{ set}$   
**and**  $set\text{-Fbc} :: 'Fbc \Rightarrow ('b \times 'c) \text{ set}$   
**assumes** *confluent*: *confluentp Rb*  
**and** *retract1-ab*:  $\bigwedge x y. Rb (\pi\text{-Fabb } x) y \implies \exists z. Eab x z \wedge y = \pi\text{-Fabb } z \wedge set\text{-Fab } z \subseteq set\text{-Fab } x$   
**and** *retract1-bc*:  $\bigwedge x y. Rb (\pi\text{-Fbcb } x) y \implies \exists z. Ebc x z \wedge y = \pi\text{-Fbcb } z \wedge set\text{-Fbc } z \subseteq set\text{-Fbc } x$   
**and** *generated-b*:  $Eb \leq equivclp Rb$   
**and** *transp-a*: *transp Ea*  
**and** *transp-c*: *transp Ec*  
**and** *equivp-ab*: *equivp Eab*  
**and** *equivp-bc*: *equivp Ebc*  
**and** *in-rel-Fab*:  $\bigwedge A x y. rel\text{-Fab } A x y \longleftrightarrow (\exists z. z \in \{x. set\text{-Fab } x \subseteq \{(x, y). A x y\}\} \wedge \pi\text{-Faba } z = x \wedge \pi\text{-Fabb } z = y)$   
**and** *in-rel-Fbc*:  $\bigwedge B x y. rel\text{-Fbc } B x y \longleftrightarrow (\exists z. z \in \{x. set\text{-Fbc } x \subseteq \{(x, y). B x y\}\} \wedge \pi\text{-Fbcb } z = x \wedge \pi\text{-Fbcc } z = y)$   
**and** *rel-comp*:  $\bigwedge A B. rel\text{-Fac } (A \text{ OO } B) = rel\text{-Fab } A \text{ OO } rel\text{-Fbc } B$   
**and**  $\pi\text{-Faba-respect}$ : *rel-fun Eab Ea  $\pi\text{-Faba}$   $\pi\text{-Faba}$*   
**and**  $\pi\text{-Fbcc-respect}$ : *rel-fun Ebc Ec  $\pi\text{-Fbcc}$   $\pi\text{-Fbcc}$*

**begin**

**lemma** *retract-ab*:  $Rb^{**} (\pi\text{-Fabb } x) y \implies \exists z. Eab x z \wedge y = \pi\text{-Fabb } z \wedge set\text{-Fab } z \subseteq set\text{-Fab } x$

$z \subseteq \text{set-Fab } x$   
 ⟨proof⟩

**lemma** *retract-bc*:  $Rb^{**} (\pi\text{-Fbcb } x) y \implies \exists z. Ebc \ x \ z \wedge y = \pi\text{-Fbcb } z \wedge \text{set-Fbc } z \subseteq \text{set-Fbc } x$   
 ⟨proof⟩

**lemma** *subdistributivity*:  $\text{rel-Fab } A \ \text{OO } Eb \ \text{OO } \text{rel-Fbc } B \leq Ea \ \text{OO } \text{rel-Fac } (A \ \text{OO } B) \ \text{OO } Ec$   
 ⟨proof⟩

**end**

**end**

## 17 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

**theory** *Old-Datatype*  
**imports** *Main*  
**begin**

### 17.1 The datatype universe

**definition** *Node* =  $\{p. \exists f \ x \ k. p = (f :: \text{nat} \Rightarrow 'b + \text{nat}, x :: 'a + \text{nat}) \wedge f \ k = \text{Inr } 0\}$

**typedef**  $('a, 'b) \ \text{node} = \text{Node} :: ((\text{nat} \Rightarrow 'b + \text{nat}) * ('a + \text{nat})) \ \text{set}$   
**morphisms** *Rep-Node Abs-Node*  
 ⟨proof⟩

Datatypes will be represented by sets of type *node*

**type-synonym**  $'a \ \text{item} = ('a, \text{unit}) \ \text{node} \ \text{set}$   
**type-synonym**  $('a, 'b) \ \text{dtree} = ('a, 'b) \ \text{node} \ \text{set}$

**definition** *Push* ::  $[('b + \text{nat}), \text{nat} \Rightarrow ('b + \text{nat})] \Rightarrow (\text{nat} \Rightarrow ('b + \text{nat}))$

**where** *Push* ==  $(\%b \ h. \ \text{case-nat } b \ h)$

**definition** *Push-Node* ::  $[('b + \text{nat}), ('a, 'b) \ \text{node}] \Rightarrow ('a, 'b) \ \text{node}$   
**where** *Push-Node* ==  $(\%n \ x. \ \text{Abs-Node} (\text{apfst } (\text{Push } n) (\text{Rep-Node } x)))$

**definition** *Atom* ::  $('a + \text{nat}) \Rightarrow ('a, 'b) \ \text{dtree}$   
**where** *Atom* ==  $(\%x. \ \{\text{Abs-Node}(\%k. \ \text{Inr } 0, \ x)\})$



**definition**  $Scons :: [('a, 'b) dtree, ('a, 'b) dtree] => ('a, 'b) dtree$   
**where**  $Scons M N == (Push-Node (Inr 1) ' M) Un (Push-Node (Inr (Suc 1)) ' N)$

**definition**  $Leaf :: 'a => ('a, 'b) dtree$   
**where**  $Leaf == Atom \circ Inl$   
**definition**  $Numb :: nat => ('a, 'b) dtree$   
**where**  $Numb == Atom \circ Inr$

**definition**  $In0 :: ('a, 'b) dtree => ('a, 'b) dtree$   
**where**  $In0(M) == Scons (Numb 0) M$   
**definition**  $In1 :: ('a, 'b) dtree => ('a, 'b) dtree$   
**where**  $In1(M) == Scons (Numb 1) M$

**definition**  $Lim :: ('b => ('a, 'b) dtree) => ('a, 'b) dtree$   
**where**  $Lim f == \bigcup \{z. \exists x. z = Push-Node (Inl x) ' (f x)\}$

**definition**  $ndepth :: ('a, 'b) node => nat$   
**where**  $ndepth(n) == (\% (f,x). LEAST k. f k = Inr 0) (Rep-Node n)$   
**definition**  $ntrunc :: [nat, ('a, 'b) dtree] => ('a, 'b) dtree$   
**where**  $ntrunc k N == \{n. n \in N \wedge ndepth(n) < k\}$

**definition**  $uprod :: [('a, 'b) dtree set, ('a, 'b) dtree set] => ('a, 'b) dtree set$   
**where**  $uprod A B == UN x:A. UN y:B. \{ Scons x y \}$   
**definition**  $usum :: [('a, 'b) dtree set, ('a, 'b) dtree set] => ('a, 'b) dtree set$   
**where**  $usum A B == In0'A Un In1'B$

**definition**  $Split :: [ (('a, 'b) dtree, ('a, 'b) dtree) => 'c, ('a, 'b) dtree ] => 'c$   
**where**  $Split c M == THE u. \exists x y. M = Scons x y \wedge u = c x y$

**definition**  $Case :: [ (('a, 'b) dtree) => 'c, (('a, 'b) dtree) => 'c, ('a, 'b) dtree ] => 'c$   
**where**  $Case c d M == THE u. (\exists x. M = In0(x) \wedge u = c(x)) \vee (\exists y. M = In1(y) \wedge u = d(y))$

**definition**  $dprod :: [ (('a, 'b) dtree * ('a, 'b) dtree) set, (('a, 'b) dtree * ('a, 'b) dtree) set ]$   
 $=> (('a, 'b) dtree * ('a, 'b) dtree) set$   
**where**  $dprod r s == UN (x,x'):r. UN (y,y'):s. \{(Scons x y, Scons x' y')\}$

**definition**  $dsum :: [ (('a, 'b) dtree * ('a, 'b) dtree) set, (('a, 'b) dtree * ('a, 'b) dtree) set ]$

$dtree)set]$   
 $=> (('a, 'b) dtree * ('a, 'b) dtree)set$   
**where**  $dsum\ r\ s == (UN\ (x,x'):r.\ \{(In0(x),In0(x'))\})\ Un\ (UN\ (y,y'):s.\ \{(In1(y),In1(y'))\})$

**lemma** *apfst-convE*:

$[[\ q = apfst\ f\ p;\ \forall x\ y.\ [[\ p = (x,y);\ q = (f(x),y) ] ] ==> R$   
 $]] ==> R$   
 $\langle proof \rangle$

**lemma** *Push-inject1*:  $Push\ i\ f = Push\ j\ g ==> i=j$   
 $\langle proof \rangle$

**lemma** *Push-inject2*:  $Push\ i\ f = Push\ j\ g ==> f=g$   
 $\langle proof \rangle$

**lemma** *Push-inject*:

$[[\ Push\ i\ f = Push\ j\ g;\ [[\ i=j;\ f=g ] ] ==> P ] ] ==> P$   
 $\langle proof \rangle$

**lemma** *Push-neq-K0*:  $Push\ (Inr\ (Suc\ k))\ f = (%z.\ Inr\ 0) ==> P$   
 $\langle proof \rangle$

**lemmas** *Abs-Node-inj = Abs-Node-inject [THEN [2] rev-iffD1]*

**lemma** *Node-K0-I*:  $(\lambda k.\ Inr\ 0,\ a) \in Node$   
 $\langle proof \rangle$

**lemma** *Node-Push-I*:  $p \in Node \implies apfst\ (Push\ i)\ p \in Node$   
 $\langle proof \rangle$

## 17.2 Freeness: Distinctness of Constructors

**lemma** *Scons-not-Atom [iff]*:  $Scons\ M\ N \neq Atom(a)$   
 $\langle proof \rangle$

**lemmas** *Atom-not-Scons [iff] = Scons-not-Atom [THEN not-sym]*

**lemma** *inj-Atom*:  $inj(Atom)$

*<proof>*

**lemmas** *Atom-inject = inj-Atom [THEN injD]*

**lemma** *Atom-Atom-eq [iff]: (Atom(a)=Atom(b)) = (a=b)*

*<proof>*

**lemma** *inj-Leaf: inj(Leaf)*

*<proof>*

**lemmas** *Leaf-inject [dest!] = inj-Leaf [THEN injD]*

**lemma** *inj-Numb: inj(Numb)*

*<proof>*

**lemmas** *Numb-inject [dest!] = inj-Numb [THEN injD]*

**lemma** *Push-Node-inject:*

$$[[ \text{Push-Node } i \ m = \text{Push-Node } j \ n; \ [[ \ i=j; \ m=n \ ] ] \ ==> \ P$$

*<proof>*

**lemma** *Scons-inject-lemma1: Scons M N <= Scons M' N' ==> M<=M'*

*<proof>*

**lemma** *Scons-inject-lemma2: Scons M N <= Scons M' N' ==> N<=N'*

*<proof>*

**lemma** *Scons-inject1: Scons M N = Scons M' N' ==> M=M'*

*<proof>*

**lemma** *Scons-inject2: Scons M N = Scons M' N' ==> N=N'*

*<proof>*

**lemma** *Scons-inject:*

$$[[ \text{Scons } M \ N = \text{Scons } M' \ N'; \ [[ \ M=M'; \ N=N' \ ] ] \ ==> \ P \ ] ] \ ==> \ P$$

*<proof>*

**lemma** *Scons-Scons-eq [iff]: (Scons M N = Scons M' N') = (M=M'  $\wedge$  N=N')*

*<proof>*

**lemma** *Scons-not-Leaf* [iff]:  $Scons\ M\ N \neq Leaf(a)$   
 ⟨proof⟩

**lemmas** *Leaf-not-Scons* [iff] = *Scons-not-Leaf* [THEN not-sym]

**lemma** *Scons-not-Numb* [iff]:  $Scons\ M\ N \neq Numb(k)$   
 ⟨proof⟩

**lemmas** *Numb-not-Scons* [iff] = *Scons-not-Numb* [THEN not-sym]

**lemma** *Leaf-not-Numb* [iff]:  $Leaf(a) \neq Numb(k)$   
 ⟨proof⟩

**lemmas** *Numb-not-Leaf* [iff] = *Leaf-not-Numb* [THEN not-sym]

**lemma** *ndepth-K0*:  $ndepth\ (Abs-Node(\%k.\ Inr\ 0,\ x)) = 0$   
 ⟨proof⟩

**lemma** *ndepth-Push-Node-aux*:  
 $case-nat\ (Inr\ (Suc\ i))\ f\ k = Inr\ 0 \longrightarrow Suc(LEAST\ x.\ f\ x = Inr\ 0) \leq k$   
 ⟨proof⟩

**lemma** *ndepth-Push-Node*:  
 $ndepth\ (Push-Node\ (Inr\ (Suc\ i))\ n) = Suc(ndepth(n))$   
 ⟨proof⟩

**lemma** *ntrunc-0* [simp]:  $ntrunc\ 0\ M = \{\}$   
 ⟨proof⟩

**lemma** *ntrunc-Atom* [simp]:  $ntrunc\ (Suc\ k)\ (Atom\ a) = Atom(a)$   
 ⟨proof⟩

**lemma** *ntrunc-Leaf* [simp]:  $ntrunc\ (Suc\ k)\ (Leaf\ a) = Leaf(a)$   
 ⟨proof⟩

**lemma** *ntrunc-Numb* [simp]:  $ntrunc\ (Suc\ k)\ (Numb\ i) = Numb(i)$   
 ⟨proof⟩

**lemma** *ntrunc-Scons* [*simp*]:

$$\text{ntrunc } (\text{Suc } k) (\text{Scons } M N) = \text{Scons } (\text{ntrunc } k M) (\text{ntrunc } k N)$$

*<proof>*

**lemma** *ntrunc-one-In0* [*simp*]:  $\text{ntrunc } (\text{Suc } 0) (\text{In0 } M) = \{\}$

*<proof>*

**lemma** *ntrunc-In0* [*simp*]:  $\text{ntrunc } (\text{Suc}(\text{Suc } k)) (\text{In0 } M) = \text{In0 } (\text{ntrunc } (\text{Suc } k) M)$

*<proof>*

**lemma** *ntrunc-one-In1* [*simp*]:  $\text{ntrunc } (\text{Suc } 0) (\text{In1 } M) = \{\}$

*<proof>*

**lemma** *ntrunc-In1* [*simp*]:  $\text{ntrunc } (\text{Suc}(\text{Suc } k)) (\text{In1 } M) = \text{In1 } (\text{ntrunc } (\text{Suc } k) M)$

*<proof>*

### 17.3 Set Constructions

**lemma** *uprodI* [*intro!*]:  $\llbracket M \in A; N \in B \rrbracket \implies \text{Scons } M N \in \text{uprod } A B$

*<proof>*

**lemma** *uprodE* [*elim!*]:

$$\begin{aligned} & \llbracket c \in \text{uprod } A B; \\ & \quad \bigwedge x y. \llbracket x \in A; y \in B; c = \text{Scons } x y \rrbracket \implies P \\ & \rrbracket \implies P \end{aligned}$$

*<proof>*

**lemma** *uprodE2*:  $\llbracket \text{Scons } M N \in \text{uprod } A B; \llbracket M \in A; N \in B \rrbracket \implies P \rrbracket \implies P$

*<proof>*

**lemma** *usum-In0I* [*intro*]:  $M \in A \implies \text{In0}(M) \in \text{usum } A B$

*<proof>*

**lemma** *usum-In1I* [*intro*]:  $N \in B \implies \text{In1}(N) \in \text{usum } A B$

*<proof>*

**lemma** *usumE* [*elim!*]:  

$$\begin{aligned} & \llbracket u \in \text{usum } A \ B; \\ & \quad \bigwedge x. \llbracket x \in A; u = \text{In0}(x) \rrbracket \implies P; \\ & \quad \bigwedge y. \llbracket y \in B; u = \text{In1}(y) \rrbracket \implies P \\ & \rrbracket \implies P \end{aligned}$$
 $\langle \text{proof} \rangle$

**lemma** *In0-not-In1* [*iff*]:  $\text{In0}(M) \neq \text{In1}(N)$   
 $\langle \text{proof} \rangle$

**lemmas** *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*]

**lemma** *In0-inject*:  $\text{In0}(M) = \text{In0}(N) \implies M = N$   
 $\langle \text{proof} \rangle$

**lemma** *In1-inject*:  $\text{In1}(M) = \text{In1}(N) \implies M = N$   
 $\langle \text{proof} \rangle$

**lemma** *In0-eq* [*iff*]:  $(\text{In0 } M = \text{In0 } N) = (M = N)$   
 $\langle \text{proof} \rangle$

**lemma** *In1-eq* [*iff*]:  $(\text{In1 } M = \text{In1 } N) = (M = N)$   
 $\langle \text{proof} \rangle$

**lemma** *inj-In0*: *inj In0*  
 $\langle \text{proof} \rangle$

**lemma** *inj-In1*: *inj In1*  
 $\langle \text{proof} \rangle$

**lemma** *Lim-inject*:  $\text{Lim } f = \text{Lim } g \implies f = g$   
 $\langle \text{proof} \rangle$

**lemma** *ntrunc-subsetI*:  $\text{ntrunc } k \ M \leq M$   
 $\langle \text{proof} \rangle$

**lemma** *ntrunc-subsetD*:  $(!!k. \text{ntrunc } k \ M \leq N) \implies M \leq N$   
 $\langle \text{proof} \rangle$

**lemma** *ntrunc-equality*:  $(!!k. \text{ntrunc } k \ M = \text{ntrunc } k \ N) \implies M=N$   
 $\langle \text{proof} \rangle$

**lemma** *ntrunc-o-equality*:  
 $(!!k. (\text{ntrunc}(k) \circ h1) = (\text{ntrunc}(k) \circ h2)) \implies h1=h2$   
 $\langle \text{proof} \rangle$

**lemma** *uprod-mono*:  $([ A \leq A'; B \leq B' ]) \implies \text{uprod } A \ B \leq \text{uprod } A' \ B'$   
 $\langle \text{proof} \rangle$

**lemma** *usum-mono*:  $([ A \leq A'; B \leq B' ]) \implies \text{usum } A \ B \leq \text{usum } A' \ B'$   
 $\langle \text{proof} \rangle$

**lemma** *Scons-mono*:  $([ M \leq M'; N \leq N' ]) \implies \text{Scons } M \ N \leq \text{Scons } M' \ N'$   
 $\langle \text{proof} \rangle$

**lemma** *In0-mono*:  $M \leq N \implies \text{In0}(M) \leq \text{In0}(N)$   
 $\langle \text{proof} \rangle$

**lemma** *In1-mono*:  $M \leq N \implies \text{In1}(M) \leq \text{In1}(N)$   
 $\langle \text{proof} \rangle$

**lemma** *Split [simp]*:  $\text{Split } c \ (\text{Scons } M \ N) = c \ M \ N$   
 $\langle \text{proof} \rangle$

**lemma** *Case-In0 [simp]*:  $\text{Case } c \ d \ (\text{In0 } M) = c(M)$   
 $\langle \text{proof} \rangle$

**lemma** *Case-In1 [simp]*:  $\text{Case } c \ d \ (\text{In1 } N) = d(N)$   
 $\langle \text{proof} \rangle$

**lemma** *ntrunc-UN1*:  $\text{ntrunc } k \ (\text{UN } x. f(x)) = (\text{UN } x. \text{ntrunc } k \ (f \ x))$   
 $\langle \text{proof} \rangle$

**lemma** *Scons-UN1-x*:  $\text{Scons } (\text{UN } x. f \ x) \ M = (\text{UN } x. \text{Scons } (f \ x) \ M)$   
 $\langle \text{proof} \rangle$

**lemma** *Scons-UN1-y*:  $\text{Scons } M \ (\text{UN } x. f \ x) = (\text{UN } x. \text{Scons } M \ (f \ x))$   
 $\langle \text{proof} \rangle$

**lemma** *In0-UN1*:  $In0(UN\ x.\ f(x)) = (UN\ x.\ In0(f(x)))$   
 ⟨proof⟩

**lemma** *In1-UN1*:  $In1(UN\ x.\ f(x)) = (UN\ x.\ In1(f(x)))$   
 ⟨proof⟩

**lemma** *dprodI* [*intro!*]:  
 $\llbracket (M, M') \in r; (N, N') \in s \rrbracket \implies (Scons\ M\ N, Scons\ M'\ N') \in dprod\ r\ s$   
 ⟨proof⟩

**lemma** *dprodE* [*elim!*]:  
 $\llbracket c \in dprod\ r\ s; \bigwedge x\ y\ x'\ y'. \llbracket (x, x') \in r; (y, y') \in s; c = (Scons\ x\ y, Scons\ x'\ y') \rrbracket \implies P \rrbracket \implies P$   
 ⟨proof⟩

**lemma** *dsum-In0I* [*intro*]:  $(M, M') \in r \implies (In0(M), In0(M')) \in dsum\ r\ s$   
 ⟨proof⟩

**lemma** *dsum-In1I* [*intro*]:  $(N, N') \in s \implies (In1(N), In1(N')) \in dsum\ r\ s$   
 ⟨proof⟩

**lemma** *dsumE* [*elim!*]:  
 $\llbracket w \in dsum\ r\ s; \bigwedge x\ x'. \llbracket (x, x') \in r; w = (In0(x), In0(x')) \rrbracket \implies P; \bigwedge y\ y'. \llbracket (y, y') \in s; w = (In1(y), In1(y')) \rrbracket \implies P \rrbracket \implies P$   
 ⟨proof⟩

**lemma** *dprod-mono*:  $\llbracket r \leq r'; s \leq s' \rrbracket \implies dprod\ r\ s \leq dprod\ r'\ s'$   
 ⟨proof⟩

**lemma** *dsum-mono*:  $\llbracket r \leq r'; s \leq s' \rrbracket \implies dsum\ r\ s \leq dsum\ r'\ s'$   
 ⟨proof⟩



**lemma** *dprod-Sigma*:  $(dprod (A \times B) (C \times D)) \leq (uprod A C) \times (uprod B D)$   
 ⟨proof⟩

**lemmas** *dprod-subset-Sigma* = subset-trans [OF dprod-mono dprod-Sigma]

**lemma** *dprod-subset-Sigma2*:

$(dprod (Sigma A B) (Sigma C D)) \leq Sigma (uprod A C) (Split (\%x y. uprod (B x) (D y)))$   
 ⟨proof⟩

**lemma** *dsum-Sigma*:  $(dsum (A \times B) (C \times D)) \leq (usum A C) \times (usum B D)$   
 ⟨proof⟩

**lemmas** *dsum-subset-Sigma* = subset-trans [OF dsum-mono dsum-Sigma]

**lemma** *Domain-dprod [simp]*:  $Domain (dprod r s) = uprod (Domain r) (Domain s)$   
 ⟨proof⟩

**lemma** *Domain-dsum [simp]*:  $Domain (dsum r s) = usum (Domain r) (Domain s)$   
 ⟨proof⟩

hides popular names

**hide-type** (open) *node item*

**hide-const** (open) *Push Node Atom Leaf Numb Lim Split Case*

⟨ML⟩

end

## 18 Bijections between natural numbers and other types

**theory** *Nat-Bijection*

**imports** *Main*

**begin**

### 18.1 Type $nat \times nat$

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

**definition** *triangle* ::  $nat \Rightarrow nat$

**where** *triangle*  $n = (n * Suc n) div 2$

**lemma** *triangle-0* [*simp*]: *triangle 0 = 0*  
 ⟨*proof*⟩

**lemma** *triangle-Suc* [*simp*]: *triangle (Suc n) = triangle n + Suc n*  
 ⟨*proof*⟩

**definition** *prod-encode* :: *nat × nat ⇒ nat*  
**where** *prod-encode* =  $(\lambda(m, n). \text{triangle } (m + n) + m)$

In this auxiliary function, *triangle k + m* is an invariant.

**fun** *prod-decode-aux* :: *nat ⇒ nat ⇒ nat × nat*  
**where** *prod-decode-aux* *k m* =  
 (if  $m \leq k$  then  $(m, k - m)$  else *prod-decode-aux* (Suc *k*) (*m* - Suc *k*))

**declare** *prod-decode-aux.simps* [*simp del*]

**definition** *prod-decode* :: *nat ⇒ nat × nat*  
**where** *prod-decode* = *prod-decode-aux 0*

**lemma** *prod-encode-prod-decode-aux*: *prod-encode (prod-decode-aux k m) = triangle k + m*  
 ⟨*proof*⟩

**lemma** *prod-decode-inverse* [*simp*]: *prod-encode (prod-decode n) = n*  
 ⟨*proof*⟩

**lemma** *prod-decode-triangle-add*: *prod-decode (triangle k + m) = prod-decode-aux k m*  
 ⟨*proof*⟩

**lemma** *prod-encode-inverse* [*simp*]: *prod-decode (prod-encode x) = x*  
 ⟨*proof*⟩

**lemma** *inj-prod-encode*: *inj-on prod-encode A*  
 ⟨*proof*⟩

**lemma** *inj-prod-decode*: *inj-on prod-decode A*  
 ⟨*proof*⟩

**lemma** *surj-prod-encode*: *surj prod-encode*  
 ⟨*proof*⟩

**lemma** *surj-prod-decode*: *surj prod-decode*  
 ⟨*proof*⟩

**lemma** *bij-prod-encode*: *bij prod-encode*  
 ⟨*proof*⟩

**lemma** *bij-prod-decode*: *bij prod-decode*  
 ⟨*proof*⟩

**lemma** *prod-encode-eq* [*simp*]: *prod-encode x = prod-encode y*  $\longleftrightarrow$  *x = y*  
 ⟨*proof*⟩

**lemma** *prod-decode-eq* [*simp*]: *prod-decode x = prod-decode y*  $\longleftrightarrow$  *x = y*  
 ⟨*proof*⟩

Ordering properties

**lemma** *le-prod-encode-1*: *a*  $\leq$  *prod-encode (a, b)*  
 ⟨*proof*⟩

**lemma** *le-prod-encode-2*: *b*  $\leq$  *prod-encode (a, b)*  
 ⟨*proof*⟩

## 18.2 Type *nat + nat*

**definition** *sum-encode* :: *nat + nat*  $\Rightarrow$  *nat*  
 where *sum-encode x* = (*case x of Inl a*  $\Rightarrow$  *2 \* a* | *Inr b*  $\Rightarrow$  *Suc (2 \* b)*)

**definition** *sum-decode* :: *nat*  $\Rightarrow$  *nat + nat*  
 where *sum-decode n* = (*if even n then Inl (n div 2) else Inr (n div 2)*)

**lemma** *sum-encode-inverse* [*simp*]: *sum-decode (sum-encode x)* = *x*  
 ⟨*proof*⟩

**lemma** *sum-decode-inverse* [*simp*]: *sum-encode (sum-decode n)* = *n*  
 ⟨*proof*⟩

**lemma** *inj-sum-encode*: *inj-on sum-encode A*  
 ⟨*proof*⟩

**lemma** *inj-sum-decode*: *inj-on sum-decode A*  
 ⟨*proof*⟩

**lemma** *surj-sum-encode*: *surj sum-encode*  
 ⟨*proof*⟩

**lemma** *surj-sum-decode*: *surj sum-decode*  
 ⟨*proof*⟩

**lemma** *bij-sum-encode*: *bij sum-encode*  
 ⟨*proof*⟩

**lemma** *bij-sum-decode*: *bij sum-decode*  
 ⟨*proof*⟩

**lemma** *sum-encode-eq*: *sum-encode x = sum-encode y*  $\longleftrightarrow$  *x = y*

*<proof>*

**lemma** *sum-decode-eq*:  $\text{sum-decode } x = \text{sum-decode } y \longleftrightarrow x = y$   
*<proof>*

### 18.3 Type *int*

**definition** *int-encode* ::  $\text{int} \Rightarrow \text{nat}$

**where** *int-encode*  $i = \text{sum-encode}$  (if  $0 \leq i$  then *Inl* (*nat*  $i$ ) else *Inr* (*nat* ( $- i - 1$ )))

**definition** *int-decode* ::  $\text{nat} \Rightarrow \text{int}$

**where** *int-decode*  $n = (\text{case } \text{sum-decode } n \text{ of } \text{Inl } a \Rightarrow \text{int } a \mid \text{Inr } b \Rightarrow - \text{int } b - 1)$

**lemma** *int-encode-inverse* [*simp*]:  $\text{int-decode } (\text{int-encode } x) = x$   
*<proof>*

**lemma** *int-decode-inverse* [*simp*]:  $\text{int-encode } (\text{int-decode } n) = n$   
*<proof>*

**lemma** *inj-int-encode*: *inj-on int-encode A*  
*<proof>*

**lemma** *inj-int-decode*: *inj-on int-decode A*  
*<proof>*

**lemma** *surj-int-encode*: *surj int-encode*  
*<proof>*

**lemma** *surj-int-decode*: *surj int-decode*  
*<proof>*

**lemma** *bij-int-encode*: *bij int-encode*  
*<proof>*

**lemma** *bij-int-decode*: *bij int-decode*  
*<proof>*

**lemma** *int-encode-eq*:  $\text{int-encode } x = \text{int-encode } y \longleftrightarrow x = y$   
*<proof>*

**lemma** *int-decode-eq*:  $\text{int-decode } x = \text{int-decode } y \longleftrightarrow x = y$   
*<proof>*

### 18.4 Type *nat list*

**fun** *list-encode* ::  $\text{nat list} \Rightarrow \text{nat}$

**where**

*list-encode* [] = 0

|  $list-encode (x \# xs) = Suc (prod-encode (x, list-encode xs))$

**function**  $list-decode :: nat \Rightarrow nat\ list$

**where**

$list-decode\ 0 = []$

|  $list-decode (Suc\ n) = (case\ prod-decode\ n\ of\ (x, y) \Rightarrow x \# list-decode\ y)$   
 $\langle proof \rangle$

**termination**  $list-decode$

$\langle proof \rangle$

**lemma**  $list-encode-inverse [simp]: list-decode (list-encode\ x) = x$

$\langle proof \rangle$

**lemma**  $list-decode-inverse [simp]: list-encode (list-decode\ n) = n$

$\langle proof \rangle$

**lemma**  $inj-list-encode: inj-on\ list-encode\ A$

$\langle proof \rangle$

**lemma**  $inj-list-decode: inj-on\ list-decode\ A$

$\langle proof \rangle$

**lemma**  $surj-list-encode: surj\ list-encode$

$\langle proof \rangle$

**lemma**  $surj-list-decode: surj\ list-decode$

$\langle proof \rangle$

**lemma**  $bij-list-encode: bij\ list-encode$

$\langle proof \rangle$

**lemma**  $bij-list-decode: bij\ list-decode$

$\langle proof \rangle$

**lemma**  $list-encode-eq: list-encode\ x = list-encode\ y \longleftrightarrow x = y$

$\langle proof \rangle$

**lemma**  $list-decode-eq: list-decode\ x = list-decode\ y \longleftrightarrow x = y$

$\langle proof \rangle$

## 18.5 Finite sets of naturals

### 18.5.1 Preliminaries

**lemma**  $finite-vimage-Suc-iff: finite (Suc -' F) \longleftrightarrow finite\ F$

$\langle proof \rangle$

**lemma**  $vimage-Suc-insert-0: Suc -' insert\ 0\ A = Suc -' A$

$\langle proof \rangle$

**lemma** *vimage-Suc-insert-Suc*:  $Suc - ' insert (Suc n) A = insert n (Suc - ' A)$   
 ⟨*proof*⟩

**lemma** *div2-even-ext-nat*:  
 fixes  $x y :: nat$   
 assumes  $x div 2 = y div 2$   
 and  $even x \longleftrightarrow even y$   
 shows  $x = y$   
 ⟨*proof*⟩

### 18.5.2 From sets to naturals

**definition** *set-encode* ::  $nat set \Rightarrow nat$   
 where  $set-encode = sum ((\ ) 2)$

**lemma** *set-encode-empty* [*simp*]:  $set-encode \{\} = 0$   
 ⟨*proof*⟩

**lemma** *set-encode-inf*:  $\neg finite A \Longrightarrow set-encode A = 0$   
 ⟨*proof*⟩

**lemma** *set-encode-insert* [*simp*]:  $finite A \Longrightarrow n \notin A \Longrightarrow set-encode (insert n A) = 2^n + set-encode A$   
 ⟨*proof*⟩

**lemma** *even-set-encode-iff*:  $finite A \Longrightarrow even (set-encode A) \longleftrightarrow 0 \notin A$   
 ⟨*proof*⟩

**lemma** *set-encode-vimage-Suc*:  $set-encode (Suc - ' A) = set-encode A div 2$   
 ⟨*proof*⟩

**lemmas** *set-encode-div-2 = set-encode-vimage-Suc* [*symmetric*]

### 18.5.3 From naturals to sets

**definition** *set-decode* ::  $nat \Rightarrow nat set$   
 where  $set-decode x = \{n. odd (x div 2 ^ n)\}$

**lemma** *set-decode-0* [*simp*]:  $0 \in set-decode x \longleftrightarrow odd x$   
 ⟨*proof*⟩

**lemma** *set-decode-Suc* [*simp*]:  $Suc n \in set-decode x \longleftrightarrow n \in set-decode (x div 2)$   
 ⟨*proof*⟩

**lemma** *set-decode-zero* [*simp*]:  $set-decode 0 = \{\}$   
 ⟨*proof*⟩

**lemma** *set-decode-div-2*:  $set-decode (x div 2) = Suc - ' set-decode x$   
 ⟨*proof*⟩

**lemma** *set-decode-plus-power-2*:

$n \notin \text{set-decode } z \implies \text{set-decode } (2 \wedge n + z) = \text{insert } n (\text{set-decode } z)$   
 ⟨proof⟩

**lemma** *finite-set-decode [simp]*:  $\text{finite } (\text{set-decode } n)$

⟨proof⟩

#### 18.5.4 Proof of isomorphism

**lemma** *set-decode-inverse [simp]*:  $\text{set-encode } (\text{set-decode } n) = n$

⟨proof⟩

**lemma** *set-encode-inverse [simp]*:  $\text{finite } A \implies \text{set-decode } (\text{set-encode } A) = A$

⟨proof⟩

**lemma** *inj-on-set-encode*:  $\text{inj-on } \text{set-encode } (\text{Collect } \text{finite})$

⟨proof⟩

**lemma** *set-encode-eq*:  $\text{finite } A \implies \text{finite } B \implies \text{set-encode } A = \text{set-encode } B \iff A = B$

⟨proof⟩

**lemma** *subset-decode-imp-le*:

**assumes**  $\text{set-decode } m \subseteq \text{set-decode } n$

**shows**  $m \leq n$

⟨proof⟩

end

## 19 Encoding (almost) everything into natural numbers

**theory** *Countable*

**imports** *Old-Datatype HOL.Rat Nat-Bijection*

**begin**

### 19.1 The class of countable types

**class** *countable* =

**assumes**  $\text{ex-inj}: \exists \text{to-nat} :: 'a \Rightarrow \text{nat}. \text{inj } \text{to-nat}$

**lemma** *countable-classI*:

**fixes**  $f :: 'a \Rightarrow \text{nat}$

**assumes**  $\bigwedge x y. f x = f y \implies x = y$

**shows**  $\text{OFCLASS}('a, \text{countable-class})$

⟨proof⟩

## 19.2 Conversion functions

**definition** *to-nat* :: 'a::countable  $\Rightarrow$  nat **where**  
*to-nat* = (SOME f. inj f)

**definition** *from-nat* :: nat  $\Rightarrow$  'a::countable **where**  
*from-nat* = inv (to-nat :: 'a  $\Rightarrow$  nat)

**lemma** *inj-to-nat* [simp]: inj to-nat  
 ⟨proof⟩

**lemma** *inj-on-to-nat*[simp, intro]: inj-on to-nat S  
 ⟨proof⟩

**lemma** *surj-from-nat* [simp]: surj from-nat  
 ⟨proof⟩

**lemma** *to-nat-split* [simp]: to-nat x = to-nat y  $\longleftrightarrow$  x = y  
 ⟨proof⟩

**lemma** *from-nat-to-nat* [simp]:  
 from-nat (to-nat x) = x  
 ⟨proof⟩

## 19.3 Finite types are countable

**subclass** (in finite) countable  
 ⟨proof⟩

## 19.4 Automatically proving countability of old-style datatypes

**context**  
**begin**

**qualified inductive** *finite-item* :: 'a Old-Datatype.item  $\Rightarrow$  bool **where**  
 undefined: finite-item undefined  
 | In0: finite-item x  $\Longrightarrow$  finite-item (Old-Datatype.In0 x)  
 | In1: finite-item x  $\Longrightarrow$  finite-item (Old-Datatype.In1 x)  
 | Leaf: finite-item (Old-Datatype.Leaf a)  
 | Scons:  $\llbracket$ finite-item x; finite-item y $\rrbracket \Longrightarrow$  finite-item (Old-Datatype.Scons x y)

**qualified function** *nth-item* :: nat  $\Rightarrow$  ('a::countable) Old-Datatype.item  
**where**  
 nth-item 0 = undefined  
 | nth-item (Suc n) =  
 (case sum-decode n of  
 Inl i  $\Rightarrow$   
 (case sum-decode i of  
 Inl j  $\Rightarrow$  Old-Datatype.In0 (nth-item j)  
 | Inr j  $\Rightarrow$  Old-Datatype.In1 (nth-item j))



```

| Inr i ⇒
  (case sum-decode i of
    Inl j ⇒ Old-Datatype.Leaf (from-nat j)
  | Inr j ⇒
    (case prod-decode j of
      (a, b) ⇒ Old-Datatype.Scons (nth-item a) (nth-item b))))
⟨proof⟩

```

**lemma** *le-sum-encode-Inl*:  $x \leq y \implies x \leq \text{sum-encode } (\text{Inl } y)$   
 ⟨proof⟩

**lemma** *le-sum-encode-Inr*:  $x \leq y \implies x \leq \text{sum-encode } (\text{Inr } y)$   
 ⟨proof⟩ **termination**  
 ⟨proof⟩

**lemma** *nth-item-covers*:  $\text{finite-item } x \implies \exists n. \text{nth-item } n = x$   
 ⟨proof⟩

**theorem** *countable-datatype*:  
**fixes** *Rep* :: 'b ⇒ ('a::countable) Old-Datatype.item  
**fixes** *Abs* :: ('a::countable) Old-Datatype.item ⇒ 'b  
**fixes** *rep-set* :: ('a::countable) Old-Datatype.item ⇒ bool  
**assumes** *type*: type-definition *Rep Abs* (Collect *rep-set*)  
**assumes** *finite-item*:  $\bigwedge x. \text{rep-set } x \implies \text{finite-item } x$   
**shows** OFCLASS('b, countable-class)  
 ⟨proof⟩

⟨ML⟩

end

## 19.5 Automatically proving countability of datatypes

⟨ML⟩

## 19.6 More Countable types

Naturals

**instance** *nat* :: countable  
 ⟨proof⟩

Pairs

**instance** *prod* :: (countable, countable) countable  
 ⟨proof⟩

Sums

**instance** *sum* :: (countable, countable) countable  
 ⟨proof⟩

Integers

**instance** *int* :: *countable*  
 ⟨*proof*⟩

Options

**instance** *option* :: (*countable*) *countable*  
 ⟨*proof*⟩

Lists

**instance** *list* :: (*countable*) *countable*  
 ⟨*proof*⟩

String literals

**instance** *String.literal* :: *countable*  
 ⟨*proof*⟩

Functions

**instance** *fun* :: (*finite*, *countable*) *countable*  
 ⟨*proof*⟩

Typereps

**instance** *typerep* :: *countable*  
 ⟨*proof*⟩

## 19.7 The rationals are countably infinite

**definition** *nat-to-rat-surj* :: *nat* ⇒ *rat* **where**

*nat-to-rat-surj* *n* = (let (*a*, *b*) = *prod-decode* *n* in *Fract* (*int-decode* *a*) (*int-decode* *b*))

**lemma** *surj-nat-to-rat-surj*: *surj nat-to-rat-surj*  
 ⟨*proof*⟩

**lemma** *Rats-eq-range-nat-to-rat-surj*:  $\mathbb{Q} = \text{range } \text{nat-to-rat-surj}$   
 ⟨*proof*⟩

**context** *field-char-0*  
**begin**

**lemma** *Rats-eq-range-of-rat-o-nat-to-rat-surj*:  
 $\mathbb{Q} = \text{range } (\text{of-rat} \circ \text{nat-to-rat-surj})$   
 ⟨*proof*⟩

**lemma** *surj-of-rat-nat-to-rat-surj*:  
 $r \in \mathbb{Q} \implies \exists n. r = \text{of-rat } (\text{nat-to-rat-surj } n)$   
 ⟨*proof*⟩

**end**

**instance** *rat* :: *countable*

*<proof>*

**theorem** *rat-denum*:  $\exists f :: \text{nat} \Rightarrow \text{rat. surj } f$

*<proof>*

**end**

## 20 Infinite Sets and Related Concepts

**theory** *Infinite-Set*

**imports** *Main*

**begin**

### 20.1 The set of natural numbers is infinite

**lemma** *infinite-nat-iff-unbounded-le*:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n \geq m. n \in S)$

**for**  $S :: \text{nat set}$

*<proof>*

**lemma** *infinite-nat-iff-unbounded*:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n > m. n \in S)$

**for**  $S :: \text{nat set}$

*<proof>*

**lemma** *finite-nat-iff-bounded*:  $\text{finite } S \longleftrightarrow (\exists k. S \subseteq \{..<k\})$

**for**  $S :: \text{nat set}$

*<proof>*

**lemma** *finite-nat-iff-bounded-le*:  $\text{finite } S \longleftrightarrow (\exists k. S \subseteq \{.. k\})$

**for**  $S :: \text{nat set}$

*<proof>*

**lemma** *finite-nat-bounded*:  $\text{finite } S \Longrightarrow \exists k. S \subseteq \{..<k\}$

**for**  $S :: \text{nat set}$

*<proof>*

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some  $k$ , there is some larger number that is an element of the set.

**lemma** *unbounded-k-infinite*:  $\forall m > k. \exists n > m. n \in S \Longrightarrow \text{infinite } (S :: \text{nat set})$

*<proof>*

**lemma** *nat-not-finite*:  $\text{finite } (\text{UNIV} :: \text{nat set}) \Longrightarrow R$

*<proof>*

**lemma** *range-inj-infinite*:

**fixes**  $f :: \text{nat} \Rightarrow 'a$

**assumes** *inj*  $f$

**shows** *infinite*  $(\text{range } f)$

*<proof>*

## 20.2 The set of integers is also infinite

**lemma** *infinite-int-iff-infinite-nat-abs*:  $\text{infinite } S \longleftrightarrow \text{infinite } ((\text{nat} \circ \text{abs}) \text{ ` } S)$   
**for**  $S :: \text{int set}$   
 ⟨proof⟩

**proposition** *infinite-int-iff-unbounded-le*:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n. |n| \geq m \wedge n \in S)$   
**for**  $S :: \text{int set}$   
 ⟨proof⟩

**proposition** *infinite-int-iff-unbounded*:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n. |n| > m \wedge n \in S)$   
**for**  $S :: \text{int set}$   
 ⟨proof⟩

**proposition** *finite-int-iff-bounded*:  $\text{finite } S \longleftrightarrow (\exists k. \text{abs ` } S \subseteq \{..<k\})$   
**for**  $S :: \text{int set}$   
 ⟨proof⟩

**proposition** *finite-int-iff-bounded-le*:  $\text{finite } S \longleftrightarrow (\exists k. \text{abs ` } S \subseteq \{.. k\})$   
**for**  $S :: \text{int set}$   
 ⟨proof⟩

## 20.3 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

**lemma** *not-INFM [simp]*:  $\neg (\text{INFM } x. P x) \longleftrightarrow (\text{MOST } x. \neg P x)$   
 ⟨proof⟩

**lemma** *not-MOST [simp]*:  $\neg (\text{MOST } x. P x) \longleftrightarrow (\text{INFM } x. \neg P x)$   
 ⟨proof⟩

**lemma** *INFM-const [simp]*:  $(\text{INFM } x::'a. P) \longleftrightarrow P \wedge \text{infinite } (\text{UNIV}::'a \text{ set})$   
 ⟨proof⟩

**lemma** *MOST-const [simp]*:  $(\text{MOST } x::'a. P) \longleftrightarrow P \vee \text{finite } (\text{UNIV}::'a \text{ set})$   
 ⟨proof⟩

**lemma** *INFM-imp-distrib*:  $(\text{INFM } x. P x \longrightarrow Q x) \longleftrightarrow ((\text{MOST } x. P x) \longrightarrow (\text{INFM } x. Q x))$   
 ⟨proof⟩

**lemma** *MOST-imp-iff*:  $\text{MOST } x. P x \Longrightarrow (\text{MOST } x. P x \longrightarrow Q x) \longleftrightarrow (\text{MOST } x. Q x)$   
 ⟨proof⟩

**lemma** *INFM-conjI*:  $INFM\ x.\ P\ x \implies MOST\ x.\ Q\ x \implies INFM\ x.\ P\ x \wedge Q\ x$   
 ⟨proof⟩

Properties of quantifiers with injective functions.

**lemma** *INFM-inj*:  $INFM\ x.\ P\ (f\ x) \implies inj\ f \implies INFM\ x.\ P\ x$   
 ⟨proof⟩

**lemma** *MOST-inj*:  $MOST\ x.\ P\ x \implies inj\ f \implies MOST\ x.\ P\ (f\ x)$   
 ⟨proof⟩

Properties of quantifiers with singletons.

**lemma** *not-INFM-eq [simp]*:  
 $\neg (INFM\ x.\ x = a)$   
 $\neg (INFM\ x.\ a = x)$   
 ⟨proof⟩

**lemma** *MOST-peq [simp]*:  
 $MOST\ x.\ x \neq a$   
 $MOST\ x.\ a \neq x$   
 ⟨proof⟩

**lemma** *INFM-peq [simp]*:  
 $(INFM\ x::'a.\ x \neq a) \longleftrightarrow infinite\ (UNIV::'a\ set)$   
 $(INFM\ x::'a.\ a \neq x) \longleftrightarrow infinite\ (UNIV::'a\ set)$   
 ⟨proof⟩

**lemma** *MOST-peq [simp]*:  
 $(MOST\ x::'a.\ x = a) \longleftrightarrow finite\ (UNIV::'a\ set)$   
 $(MOST\ x::'a.\ a = x) \longleftrightarrow finite\ (UNIV::'a\ set)$   
 ⟨proof⟩

**lemma** *MOST-peq-imp*:  
 $MOST\ x.\ x = a \longrightarrow P\ x$   
 $MOST\ x.\ a = x \longrightarrow P\ x$   
 ⟨proof⟩

Properties of quantifiers over the naturals.

**lemma** *MOST-nat*:  $(\forall_{\infty} n.\ P\ n) \longleftrightarrow (\exists m.\ \forall n > m.\ P\ n)$   
 for  $P :: nat \Rightarrow bool$   
 ⟨proof⟩

**lemma** *MOST-nat-le*:  $(\forall_{\infty} n.\ P\ n) \longleftrightarrow (\exists m.\ \forall n \geq m.\ P\ n)$   
 for  $P :: nat \Rightarrow bool$   
 ⟨proof⟩

**lemma** *INFM-nat*:  $(\exists_{\infty} n.\ P\ n) \longleftrightarrow (\forall m.\ \exists n > m.\ P\ n)$   
 for  $P :: nat \Rightarrow bool$   
 ⟨proof⟩

**lemma** *INFM-nat-le*:  $(\exists_{\infty} n. P n) \longleftrightarrow (\forall m. \exists n \geq m. P n)$   
**for**  $P :: \text{nat} \Rightarrow \text{bool}$   
 ⟨proof⟩

**lemma** *MOST-INFM*:  $\text{infinite } (UNIV :: 'a \text{ set}) \Longrightarrow \text{MOST } x :: 'a. P x \Longrightarrow \text{INFM}$   
 $x :: 'a. P x$   
 ⟨proof⟩

**lemma** *MOST-Suc-iff*:  $(\text{MOST } n. P (\text{Suc } n)) \longleftrightarrow (\text{MOST } n. P n)$   
 ⟨proof⟩

**lemma** *MOST-SucI*:  $\text{MOST } n. P n \Longrightarrow \text{MOST } n. P (\text{Suc } n)$   
**and** *MOST-SucD*:  $\text{MOST } n. P (\text{Suc } n) \Longrightarrow \text{MOST } n. P n$   
 ⟨proof⟩

**lemma** *MOST-ge-nat*:  $\text{MOST } n :: \text{nat}. m \leq n$   
 ⟨proof⟩

**lemma** *Inf-many-def*:  $\text{Inf-many } P \longleftrightarrow \text{infinite } \{x. P x\}$  ⟨proof⟩

**lemma** *Alm-all-def*:  $\text{Alm-all } P \longleftrightarrow \neg (\text{INFM } x. \neg P x)$  ⟨proof⟩

**lemma** *INFM-iff-infinite*:  $(\text{INFM } x. P x) \longleftrightarrow \text{infinite } \{x. P x\}$  ⟨proof⟩

**lemma** *MOST-iff-cofinite*:  $(\text{MOST } x. P x) \longleftrightarrow \text{finite } \{x. \neg P x\}$  ⟨proof⟩

**lemma** *INFM-EX*:  $(\exists_{\infty} x. P x) \Longrightarrow (\exists x. P x)$  ⟨proof⟩

**lemma** *ALL-MOST*:  $\forall x. P x \Longrightarrow \forall_{\infty} x. P x$  ⟨proof⟩

**lemma** *INFM-mono*:  $\exists_{\infty} x. P x \Longrightarrow (\bigwedge x. P x \Longrightarrow Q x) \Longrightarrow \exists_{\infty} x. Q x$  ⟨proof⟩

**lemma** *MOST-mono*:  $\forall_{\infty} x. P x \Longrightarrow (\bigwedge x. P x \Longrightarrow Q x) \Longrightarrow \forall_{\infty} x. Q x$  ⟨proof⟩

**lemma** *INFM-disj-distrib*:  $(\exists_{\infty} x. P x \vee Q x) \longleftrightarrow (\exists_{\infty} x. P x) \vee (\exists_{\infty} x. Q x)$   
 ⟨proof⟩

**lemma** *MOST-rev-mp*:  $\forall_{\infty} x. P x \Longrightarrow \forall_{\infty} x. P x \longrightarrow Q x \Longrightarrow \forall_{\infty} x. Q x$  ⟨proof⟩

**lemma** *MOST-conj-distrib*:  $(\forall_{\infty} x. P x \wedge Q x) \longleftrightarrow (\forall_{\infty} x. P x) \wedge (\forall_{\infty} x. Q x)$   
 ⟨proof⟩

**lemma** *MOST-conjI*:  $\text{MOST } x. P x \Longrightarrow \text{MOST } x. Q x \Longrightarrow \text{MOST } x. P x \wedge Q x$   
 ⟨proof⟩

**lemma** *INFM-finite-Bex-distrib*:  $\text{finite } A \Longrightarrow (\text{INFM } y. \exists x \in A. P x y) \longleftrightarrow (\exists x \in A. \text{INFM } y. P x y)$  ⟨proof⟩

**lemma** *MOST-finite-Ball-distrib*:  $\text{finite } A \Longrightarrow (\text{MOST } y. \forall x \in A. P x y) \longleftrightarrow (\forall x \in A. \text{MOST } y. P x y)$  ⟨proof⟩

**lemma** *INFM-E*:  $\text{INFM } x. P x \Longrightarrow (\bigwedge x. P x \Longrightarrow \text{thesis}) \Longrightarrow \text{thesis}$  ⟨proof⟩

**lemma** *MOST-I*:  $(\bigwedge x. P x) \Longrightarrow \text{MOST } x. P x$  ⟨proof⟩

**lemmas** *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

## 20.4 Enumeration of an Infinite Set

The set’s element type must be wellordered (e.g. the natural numbers).

Could be generalized to  $\text{enumerate}' S n = (\text{SOME } t. t \in s \wedge \text{finite } \{s \in S. s < t\} \wedge \text{card } \{s \in S. s < t\} = n)$ .

**primrec** (in wellorder) *enumerate* ::  $'a \text{ set} \Rightarrow \text{nat} \Rightarrow 'a$

**where**

*enumerate-0*:  $\text{enumerate } S 0 = (\text{LEAST } n. n \in S)$

| *enumerate-Suc*:  $\text{enumerate } S (\text{Suc } n) = \text{enumerate } (S - \{\text{LEAST } n. n \in S\}) n$

**lemma** *enumerate-Suc'*:  $\text{enumerate } S (\text{Suc } n) = \text{enumerate } (S - \{\text{enumerate } S 0\}) n$   
 ⟨proof⟩

**lemma** *enumerate-in-set*:  $\text{infinite } S \implies \text{enumerate } S n \in S$   
 ⟨proof⟩

**declare** *enumerate-0* [simp del] *enumerate-Suc* [simp del]

**lemma** *enumerate-step*:  $\text{infinite } S \implies \text{enumerate } S n < \text{enumerate } S (\text{Suc } n)$   
 ⟨proof⟩

**lemma** *enumerate-mono*:  $m < n \implies \text{infinite } S \implies \text{enumerate } S m < \text{enumerate } S n$   
 ⟨proof⟩

**lemma** *enumerate-mono-iff* [simp]:  
 $\text{infinite } S \implies \text{enumerate } S m < \text{enumerate } S n \longleftrightarrow m < n$   
 ⟨proof⟩

**lemma** *enumerate-mono-le-iff* [simp]:  
 $\text{infinite } S \implies \text{enumerate } S m \leq \text{enumerate } S n \longleftrightarrow m \leq n$   
 ⟨proof⟩

**lemma** *le-enumerate*:  
**assumes**  $S$ : *infinite*  $S$   
**shows**  $n \leq \text{enumerate } S n$   
 ⟨proof⟩

**lemma** *infinite-enumerate*:  
**assumes**  $fS$ : *infinite*  $S$   
**shows**  $\exists r :: \text{nat} \Rightarrow \text{nat}. \text{strict-mono } r \wedge (\forall n. r n \in S)$   
 ⟨proof⟩

**lemma** *enumerate-Suc''*:  
**fixes**  $S :: 'a :: \text{wellorder set}$   
**assumes** *infinite*  $S$   
**shows**  $\text{enumerate } S (\text{Suc } n) = (\text{LEAST } s. s \in S \wedge \text{enumerate } S n < s)$   
 ⟨proof⟩

**lemma** *enumerate-Ex*:  
**fixes**  $S :: \text{nat set}$   
**assumes**  $S$ : *infinite*  $S$   
**and**  $s \in S$   
**shows**  $\exists n. \text{enumerate } S n = s$   
 ⟨proof⟩

**lemma** *inj-enumerate*:  
**fixes**  $S :: 'a::wellorder\ set$   
**assumes**  $S: infinite\ S$   
**shows**  $inj\ (enumerate\ S)$   
 $\langle proof \rangle$

To generalise this, we’d need a condition that all initial segments were finite

**lemma** *bij-enumerate*:  
**fixes**  $S :: nat\ set$   
**assumes**  $S: infinite\ S$   
**shows**  $bij\ betw\ (enumerate\ S)\ UNIV\ S$   
 $\langle proof \rangle$

**lemma**  
**fixes**  $S :: nat\ set$   
**assumes**  $S: infinite\ S$   
**shows**  $range\ enumerate: range\ (enumerate\ S) = S$   
**and**  $strict\ mono\ enumerate: strict\ mono\ (enumerate\ S)$   
 $\langle proof \rangle$

A pair of weird and wonderful lemmas from HOL Light.

**lemma** *finite-transitivity-chain*:  
**assumes**  $finite\ A$   
**and**  $R: \bigwedge x. \neg R\ x\ x \wedge x\ y\ z. \llbracket R\ x\ y; R\ y\ z \rrbracket \implies R\ x\ z$   
**and**  $A: \bigwedge x. x \in A \implies \exists y. y \in A \wedge R\ x\ y$   
**shows**  $A = \{\}$   
 $\langle proof \rangle$

**corollary** *Union-maximal-sets*:  
**assumes**  $finite\ \mathcal{F}$   
**shows**  $\bigcup \{T \in \mathcal{F}. \forall U \in \mathcal{F}. \neg T \subset U\} = \bigcup \mathcal{F}$   
**(is ?lhs = ?rhs)**  
 $\langle proof \rangle$

## 20.5 Properties of *wellorder-class.enumerate* on finite sets

**lemma** *finite-enumerate-in-set*:  $\llbracket finite\ S; n < card\ S \rrbracket \implies enumerate\ S\ n \in S$   
 $\langle proof \rangle$

**lemma** *finite-enumerate-step*:  $\llbracket finite\ S; Suc\ n < card\ S \rrbracket \implies enumerate\ S\ n < enumerate\ S\ (Suc\ n)$   
 $\langle proof \rangle$

**lemma** *finite-enumerate-mono*:  $\llbracket m < n; finite\ S; n < card\ S \rrbracket \implies enumerate\ S\ m < enumerate\ S\ n$   
 $\langle proof \rangle$

**lemma** *finite-enumerate-mono-iff* [*simp*]:



$\llbracket \text{finite } S; m < \text{card } S; n < \text{card } S \rrbracket \implies \text{enumerate } S \ m < \text{enumerate } S \ n \longleftrightarrow m < n$   
 ⟨proof⟩

**lemma** *finite-le-enumerate*:  
**assumes** *finite*  $S$   $n < \text{card } S$   
**shows**  $n \leq \text{enumerate } S \ n$   
 ⟨proof⟩

**lemma** *finite-enumerate*:  
**assumes**  $fS$ : *finite*  $S$   
**shows**  $\exists r :: \text{nat} \implies \text{nat. strict-mono-on } \{.. < \text{card } S\} \ r \wedge (\forall n < \text{card } S. r \ n \in S)$   
 ⟨proof⟩

**lemma** *finite-enumerate-Suc''*:  
**fixes**  $S :: 'a :: \text{wellorder set}$   
**assumes** *finite*  $S$   $\text{Suc } n < \text{card } S$   
**shows**  $\text{enumerate } S \ (\text{Suc } n) = (\text{LEAST } s. s \in S \wedge \text{enumerate } S \ n < s)$   
 ⟨proof⟩

**lemma** *finite-enumerate-initial-segment*:  
**fixes**  $S :: 'a :: \text{wellorder set}$   
**assumes** *finite*  $S$  **and**  $n$ :  $n < \text{card } (S \cap \{.. < s\})$   
**shows**  $\text{enumerate } (S \cap \{.. < s\}) \ n = \text{enumerate } S \ n$   
 ⟨proof⟩

**lemma** *finite-enumerate-Ex*:  
**fixes**  $S :: 'a :: \text{wellorder set}$   
**assumes**  $S$ : *finite*  $S$   
**and**  $s$ :  $s \in S$   
**shows**  $\exists n < \text{card } S. \text{enumerate } S \ n = s$   
 ⟨proof⟩

**lemma** *finite-enum-subset*:  
**assumes**  $\bigwedge i. i < \text{card } X \implies \text{enumerate } X \ i = \text{enumerate } Y \ i$  **and** *finite*  $X$  *finite*  $Y$   $\text{card } X \leq \text{card } Y$   
**shows**  $X \subseteq Y$   
 ⟨proof⟩

**lemma** *finite-enum-ext*:  
**assumes**  $\bigwedge i. i < \text{card } X \implies \text{enumerate } X \ i = \text{enumerate } Y \ i$  **and** *finite*  $X$  *finite*  $Y$   $\text{card } X = \text{card } Y$   
**shows**  $X = Y$   
 ⟨proof⟩

**lemma** *finite-bij-enumerate*:  
**fixes**  $S :: 'a :: \text{wellorder set}$   
**assumes**  $S$ : *finite*  $S$   
**shows** *bij-betw*  $(\text{enumerate } S) \ \{.. < \text{card } S\} \ S$

*<proof>*

**lemma** *ex-bij-betw-strict-mono-card*:

**fixes**  $M :: 'a::wellorder\ set$

**assumes** *finite*  $M$

**obtains**  $h$  **where** *bij-betw*  $h \{..\lt card\ M\} M$  **and** *strict-mono-on*  $\{..\lt card\ M\} h$   
*<proof>*

**end**

## 21 Countable sets

**theory** *Countable-Set*

**imports** *Countable Infinite-Set*

**begin**

### 21.1 Predicate for countable sets

**definition** *countable*  $:: 'a\ set \Rightarrow bool$  **where**  
*countable*  $S \longleftrightarrow (\exists f::'a \Rightarrow nat.\ inj\text{-on}\ f\ S)$

**lemma** *countable-as-injective-image-subset*: *countable*  $S \longleftrightarrow (\exists f.\ \exists K::nat\ set.\ S = f\ `K \wedge inj\text{-on}\ f\ K)$   
*<proof>*

**lemma** *countableE*:

**assumes**  $S: countable\ S$  **obtains**  $f :: 'a \Rightarrow nat$  **where** *inj-on*  $f\ S$   
*<proof>*

**lemma** *countableI*: *inj-on*  $(f::'a \Rightarrow nat)\ S \Longrightarrow countable\ S$   
*<proof>*

**lemma** *countableI'*: *inj-on*  $(f::'a \Rightarrow 'b::countable)\ S \Longrightarrow countable\ S$   
*<proof>*

**lemma** *countableE-bij*:

**assumes**  $S: countable\ S$  **obtains**  $f :: nat \Rightarrow 'a$  **and**  $C :: nat\ set$  **where** *bij-betw*  $f\ C\ S$   
*<proof>*

**lemma** *countableI-bij*: *bij-betw*  $f\ (C::nat\ set)\ S \Longrightarrow countable\ S$   
*<proof>*

**lemma** *countable-finite*: *finite*  $S \Longrightarrow countable\ S$   
*<proof>*

**lemma** *countableI-bij1*: *bij-betw*  $f\ A\ B \Longrightarrow countable\ A \Longrightarrow countable\ B$   
*<proof>*

**lemma** *countableI-bij2*:  $\text{bij-betw } f \ B \ A \implies \text{countable } A \implies \text{countable } B$   
 ⟨proof⟩

**lemma** *countable-iff-bij[simp]*:  $\text{bij-betw } f \ A \ B \implies \text{countable } A \longleftrightarrow \text{countable } B$   
 ⟨proof⟩

**lemma** *countable-subset*:  $A \subseteq B \implies \text{countable } B \implies \text{countable } A$   
 ⟨proof⟩

**lemma** *countableI-type[intro, simp]*:  $\text{countable } (A :: 'a :: \text{countable set})$   
 ⟨proof⟩

## 21.2 Enumerate a countable set

**lemma** *countableE-infinite*:  
**assumes** *countable S infinite S*  
**obtains**  $e :: 'a \Rightarrow \text{nat}$  **where** *bij-betw e S UNIV*  
 ⟨proof⟩

**lemma** *countable-infiniteE'*:  
**assumes** *countable A infinite A*  
**obtains**  $g$  **where** *bij-betw g (UNIV :: nat set) A*  
 ⟨proof⟩

**lemma** *countable-enum-cases*:  
**assumes** *countable S*  
**obtains**  $(\text{finite}) \ f :: 'a \Rightarrow \text{nat}$  **where** *finite S bij-betw f S \{.. $\text{card } S$ \}*  
 |  $(\text{infinite}) \ f :: 'a \Rightarrow \text{nat}$  **where** *infinite S bij-betw f S UNIV*  
 ⟨proof⟩

**definition** *to-nat-on* ::  $'a \text{ set} \Rightarrow 'a \Rightarrow \text{nat}$  **where**  
*to-nat-on S = (SOME f. if finite S then bij-betw f S \{.. $\text{card } S$ \} else bij-betw f S UNIV)*

**definition** *from-nat-into* ::  $'a \text{ set} \Rightarrow \text{nat} \Rightarrow 'a$  **where**  
*from-nat-into S n = (if n  $\in$  to-nat-on S ' S then inv-into S (to-nat-on S) n else SOME s. s  $\in$  S)*

**lemma** *to-nat-on-finite*:  $\text{finite } S \implies \text{bij-betw } (\text{to-nat-on } S) \ S \ \{.. $\text{card } S$ \}$   
 ⟨proof⟩

**lemma** *to-nat-on-infinite*:  $\text{countable } S \implies \text{infinite } S \implies \text{bij-betw } (\text{to-nat-on } S) \ S \ \text{UNIV}$   
 ⟨proof⟩

**lemma** *bij-betw-from-nat-into-finite*:  $\text{finite } S \implies \text{bij-betw } (\text{from-nat-into } S) \ \{.. $\text{card } S$ \} \ S$   
 ⟨proof⟩

**lemma** *bij-betw-from-nat-into*:  $\text{countable } S \implies \text{infinite } S \implies \text{bij-betw } (\text{from-nat-into } S) \text{ UNIV } S$

*<proof>*

The sum/product over the enumeration of a finite set equals simply the sum/product over the set

**context** *comm-monoid-set*

**begin**

**lemma** *card-from-nat-into*:

$F (\lambda i. h (\text{from-nat-into } A i)) \{..<\text{card } A\} = F h A$

*<proof>*

**end**

**lemma** *countable-as-injective-image*:

**assumes** *countable*  $A$  *infinite*  $A$

**obtains**  $f :: \text{nat} \Rightarrow 'a$  **where**  $A = \text{range } f$  *inj*  $f$

*<proof>*

**lemma** *inj-on-to-nat-on[intro]*:  $\text{countable } A \implies \text{inj-on } (\text{to-nat-on } A) A$

*<proof>*

**lemma** *to-nat-on-inj[simp]*:

$\text{countable } A \implies a \in A \implies b \in A \implies \text{to-nat-on } A a = \text{to-nat-on } A b \iff a = b$

*<proof>*

**lemma** *from-nat-into-to-nat-on[simp]*:  $\text{countable } A \implies a \in A \implies \text{from-nat-into } A (\text{to-nat-on } A a) = a$

*<proof>*

**lemma** *subset-range-from-nat-into*:  $\text{countable } A \implies A \subseteq \text{range } (\text{from-nat-into } A)$

*<proof>*

**lemma** *from-nat-into*:  $A \neq \{\}$   $\implies \text{from-nat-into } A n \in A$

*<proof>*

**lemma** *range-from-nat-into-subset*:  $A \neq \{\} \implies \text{range } (\text{from-nat-into } A) \subseteq A$

*<proof>*

**lemma** *range-from-nat-into[simp]*:  $A \neq \{\} \implies \text{countable } A \implies \text{range } (\text{from-nat-into } A) = A$

*<proof>*

**lemma** *image-to-nat-on*:  $\text{countable } A \implies \text{infinite } A \implies \text{to-nat-on } A ' A = \text{UNIV}$

*<proof>*

**lemma** *to-nat-on-surj*:  $\text{countable } A \implies \text{infinite } A \implies \exists a \in A. \text{to-nat-on } A a = n$

*<proof>*

**lemma** *to-nat-on-from-nat-into[simp]*:  $n \in \text{to-nat-on } A \text{ ‘ } A \implies \text{to-nat-on } A \text{ (from-nat-into } A \text{ } n) = n$   
 ⟨proof⟩

**lemma** *to-nat-on-from-nat-into-infinite[simp]*:  
 $\text{countable } A \implies \text{infinite } A \implies \text{to-nat-on } A \text{ (from-nat-into } A \text{ } n) = n$   
 ⟨proof⟩

**lemma** *from-nat-into-inj*:  
 $\text{countable } A \implies m \in \text{to-nat-on } A \text{ ‘ } A \implies n \in \text{to-nat-on } A \text{ ‘ } A \implies$   
 $\text{from-nat-into } A \text{ } m = \text{from-nat-into } A \text{ } n \longleftrightarrow m = n$   
 ⟨proof⟩

**lemma** *from-nat-into-inj-infinite[simp]*:  
 $\text{countable } A \implies \text{infinite } A \implies \text{from-nat-into } A \text{ } m = \text{from-nat-into } A \text{ } n \longleftrightarrow m$   
 $= n$   
 ⟨proof⟩

**lemma** *eq-from-nat-into-iff*:  
 $\text{countable } A \implies x \in A \implies i \in \text{to-nat-on } A \text{ ‘ } A \implies x = \text{from-nat-into } A \text{ } i \longleftrightarrow$   
 $i = \text{to-nat-on } A \text{ } x$   
 ⟨proof⟩

**lemma** *from-nat-into-surj*:  $\text{countable } A \implies a \in A \implies \exists n. \text{from-nat-into } A \text{ } n = a$   
 ⟨proof⟩

**lemma** *from-nat-into-inject[simp]*:  
 $A \neq \{\} \implies \text{countable } A \implies B \neq \{\} \implies \text{countable } B \implies \text{from-nat-into } A =$   
 $\text{from-nat-into } B \longleftrightarrow A = B$   
 ⟨proof⟩

**lemma** *inj-on-from-nat-into*:  $\text{inj-on from-nat-into } (\{A. A \neq \{\} \wedge \text{countable } A\})$   
 ⟨proof⟩

### 21.3 Closure properties of countability

**lemma** *countable-SIGMA[intro, simp]*:  
 $\text{countable } I \implies (\bigwedge i. i \in I \implies \text{countable } (A \text{ } i)) \implies \text{countable } (\text{SIGMA } i : I. A$   
 $i)$   
 ⟨proof⟩

**lemma** *countable-image[intro, simp]*:  
**assumes** *countable*  $A$   
**shows** *countable*  $(f \text{ ‘ } A)$   
 ⟨proof⟩

**lemma** *countable-image-inj-on*:  $\text{countable } (f \text{ ‘ } A) \implies \text{inj-on } f \text{ } A \implies \text{countable } A$

*<proof>*

**lemma** *countable-image-inj-Int-vimage*:

$\llbracket \text{inj-on } f \ S; \text{ countable } A \rrbracket \implies \text{countable } (S \cap f^{-1} A)$   
*<proof>*

**lemma** *countable-image-inj-gen*:

$\llbracket \text{inj-on } f \ S; \text{ countable } A \rrbracket \implies \text{countable } \{x \in S. f \ x \in A\}$   
*<proof>*

**lemma** *countable-image-inj-eg*:

$\text{inj-on } f \ S \implies \text{countable}(f^{-1} S) \longleftrightarrow \text{countable } S$   
*<proof>*

**lemma** *countable-image-inj*:

$\llbracket \text{countable } A; \text{ inj } f \rrbracket \implies \text{countable } \{x. f \ x \in A\}$   
*<proof>*

**lemma** *countable-UN[intro, simp]*:

**fixes**  $I :: 'i \text{ set}$  **and**  $A :: 'a \text{ set}$

**assumes**  $I: \text{countable } I$

**assumes**  $A: \bigwedge i. i \in I \implies \text{countable } (A \ i)$

**shows**  $\text{countable } (\bigcup_{i \in I}. A \ i)$

*<proof>*

**lemma** *countable-Un[intro]*:  $\text{countable } A \implies \text{countable } B \implies \text{countable } (A \cup B)$

*<proof>*

**lemma** *countable-Un-iff[simp]*:  $\text{countable } (A \cup B) \longleftrightarrow \text{countable } A \wedge \text{countable } B$

*<proof>*

**lemma** *countable-Plus[intro, simp]*:

$\text{countable } A \implies \text{countable } B \implies \text{countable } (A <+> B)$

*<proof>*

**lemma** *countable-empty[intro, simp]*:  $\text{countable } \{\}$

*<proof>*

**lemma** *countable-insert[intro, simp]*:  $\text{countable } A \implies \text{countable } (\text{insert } a \ A)$

*<proof>*

**lemma** *countable-Int1[intro, simp]*:  $\text{countable } A \implies \text{countable } (A \cap B)$

*<proof>*

**lemma** *countable-Int2[intro, simp]*:  $\text{countable } B \implies \text{countable } (A \cap B)$

*<proof>*

**lemma** *countable-INT[intro, simp]*:  $i \in I \implies \text{countable } (A \ i) \implies \text{countable } (\bigcap_{i \in I}. A \ i)$

*<proof>*

**lemma** *countable-Diff*[*intro, simp*]: *countable A*  $\implies$  *countable (A - B)*  
*<proof>*

**lemma** *countable-insert-eq* [*simp*]: *countable (insert x A) = countable A*  
*<proof>*

**lemma** *countable-vimage*:  $B \subseteq \text{range } f \implies \text{countable } (f^{-1} B) \implies \text{countable } B$   
*<proof>*

**lemma** *surj-countable-vimage*: *surj f*  $\implies \text{countable } (f^{-1} B) \implies \text{countable } B$   
*<proof>*

**lemma** *countable-Collect*[*simp*]: *countable A*  $\implies \text{countable } \{a \in A. \varphi a\}$   
*<proof>*

**lemma** *countable-Image*:  
**assumes**  $\bigwedge y. y \in Y \implies \text{countable } (X^{-1} \{y\})$   
**assumes** *countable Y*  
**shows** *countable (X^{-1} Y)*  
*<proof>*

**lemma** *countable-relpow*:  
**fixes**  $X :: 'a \text{ rel}$   
**assumes** *Image-X*:  $\bigwedge Y. \text{countable } Y \implies \text{countable } (X^{-1} Y)$   
**assumes**  $Y: \text{countable } Y$   
**shows** *countable ((X<sup>~</sup>)<sup>n</sup> Y)*  
*<proof>*

**lemma** *countable-funpow*:  
**fixes**  $f :: 'a \text{ set} \Rightarrow 'a \text{ set}$   
**assumes**  $\bigwedge A. \text{countable } A \implies \text{countable } (f A)$   
**and** *countable A*  
**shows** *countable ((f<sup>~</sup>)<sup>n</sup> A)*  
*<proof>*

**lemma** *countable-rtrancl*:  
 $(\bigwedge Y. \text{countable } Y \implies \text{countable } (X^{-1} Y)) \implies \text{countable } Y \implies \text{countable } (X^*$   
 $^{-1} Y)$   
*<proof>*

**lemma** *countable-lists*[*intro, simp*]:  
**assumes**  $A: \text{countable } A$  **shows** *countable (lists A)*  
*<proof>*

**lemma** *Collect-finite-eq-lists*: *Collect finite = set^{-1} lists UNIV*  
*<proof>*

**lemma** *countable-Collect-finite*: *countable* (*Collect* (*finite*::'a::countable set $\Rightarrow$ bool))  
 ⟨*proof*⟩

**lemma** *countable-int*: *countable*  $\mathbf{Z}$   
 ⟨*proof*⟩

**lemma** *countable-rat*: *countable*  $\mathbf{Q}$   
 ⟨*proof*⟩

**lemma** *Collect-finite-subset-eq-lists*:  $\{A. \text{finite } A \wedge A \subseteq T\} = \text{set } \text{'lists } T$   
 ⟨*proof*⟩

**lemma** *countable-Collect-finite-subset*:  
*countable*  $T \Rightarrow \text{countable } \{A. \text{finite } A \wedge A \subseteq T\}$   
 ⟨*proof*⟩

**lemma** *countable-Fpow*: *countable*  $S \Rightarrow \text{countable } (\text{Fpow } S)$   
 ⟨*proof*⟩

**lemma** *countable-set-option* [*simp*]: *countable* (*set-option*  $x$ )  
 ⟨*proof*⟩

## 21.4 Misc lemmas

**lemma** *countable-subset-image*:  
*countable*  $B \wedge B \subseteq (f \text{' } A) \longleftrightarrow (\exists A'. \text{countable } A' \wedge A' \subseteq A \wedge (B = f \text{' } A'))$   
 (is ?lhs = ?rhs)  
 ⟨*proof*⟩

**lemma** *ex-subset-image-inj*:  
 $(\exists T. T \subseteq f \text{' } S \wedge P T) \longleftrightarrow (\exists T. T \subseteq S \wedge \text{inj-on } f T \wedge P (f \text{' } T))$   
 ⟨*proof*⟩

**lemma** *all-subset-image-inj*:  
 $(\forall T. T \subseteq f \text{' } S \longrightarrow P T) \longleftrightarrow (\forall T. T \subseteq S \wedge \text{inj-on } f T \longrightarrow P (f \text{' } T))$   
 ⟨*proof*⟩

**lemma** *ex-countable-subset-image-inj*:  
 $(\exists T. \text{countable } T \wedge T \subseteq f \text{' } S \wedge P T) \longleftrightarrow$   
 $(\exists T. \text{countable } T \wedge T \subseteq S \wedge \text{inj-on } f T \wedge P (f \text{' } T))$   
 ⟨*proof*⟩

**lemma** *all-countable-subset-image-inj*:  
 $(\forall T. \text{countable } T \wedge T \subseteq f \text{' } S \longrightarrow P T) \longleftrightarrow (\forall T. \text{countable } T \wedge T \subseteq S \wedge$   
 $\text{inj-on } f T \longrightarrow P (f \text{' } T))$   
 ⟨*proof*⟩

**lemma** *ex-countable-subset-image*:  
 $(\exists T. \text{countable } T \wedge T \subseteq f \text{' } S \wedge P T) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge P (f$



‘  $T$ ))  
 ⟨proof⟩

**lemma** *all-countable-subset-image*:

$(\forall T. \text{countable } T \wedge T \subseteq f \text{ ‘ } S \longrightarrow P \ T) \longleftrightarrow (\forall T. \text{countable } T \wedge T \subseteq S \longrightarrow P(f \text{ ‘ } T))$   
 ⟨proof⟩

**lemma** *countable-image-eq*:

$\text{countable}(f \text{ ‘ } S) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge f \text{ ‘ } S = f \text{ ‘ } T)$   
 ⟨proof⟩

**lemma** *countable-image-eq-inj*:

$\text{countable}(f \text{ ‘ } S) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge f \text{ ‘ } S = f \text{ ‘ } T \wedge \text{inj-on } f \ T)$   
 ⟨proof⟩

**lemma** *infinite-countable-subset'*:

**assumes**  $X$ : *infinite*  $X$  **shows**  $\exists C \subseteq X. \text{countable } C \wedge \text{infinite } C$   
 ⟨proof⟩

**lemma** *countable-all*:

**assumes**  $S$ : *countable*  $S$   
**shows**  $(\forall s \in S. P \ s) \longleftrightarrow (\forall n :: \text{nat}. \text{from-nat-into } S \ n \in S \longrightarrow P \ (\text{from-nat-into } S \ n))$   
 ⟨proof⟩

**lemma** *finite-sequence-to-countable-set*:

**assumes** *countable*  $X$   
**obtains**  $F$  **where**  $\bigwedge i. F \ i \subseteq X \ \bigwedge i. F \ i \subseteq F \ (\text{Suc } i) \ \bigwedge i. \text{finite } (F \ i) \ (\bigcup i. F \ i) = X$   
 ⟨proof⟩

**lemma** *transfer-countable[transfer-rule]*:

*bi-unique*  $R \implies \text{rel-fun } (\text{rel-set } R) \ (=) \text{countable countable}$   
 ⟨proof⟩

## 21.5 Uncountable

**abbreviation** *uncountable where*

$\text{uncountable } A \equiv \neg \text{countable } A$

**lemma** *uncountable-def*:  $\text{uncountable } A \longleftrightarrow A \neq \{\} \wedge \neg (\exists f :: (\text{nat} \Rightarrow 'a). \text{range } f = A)$   
 ⟨proof⟩

**lemma** *uncountable-bij-betw*:  $\text{bij-betw } f \ A \ B \implies \text{uncountable } B \implies \text{uncountable } A$   
 ⟨proof⟩

**lemma** *uncountable-infinite*:  $\text{uncountable } A \implies \text{infinite } A$

*<proof>*

**lemma** *uncountable-minus-countable*:

*uncountable A  $\implies$  countable B  $\implies$  uncountable (A - B)*

*<proof>*

**lemma** *countable-Diff-eq [simp]*: *countable (A - {x}) = countable A*

*<proof>*

Every infinite set can be covered by a pairwise disjoint family of infinite sets. This version doesn't achieve equality, as it only covers a countable subset

**lemma** *infinite-infinite-partition*:

**assumes** *infinite A*

**obtains** *C :: nat  $\implies$  'a set*

**where** *pairwise ( $\lambda i j. \text{disjnt } (C i) (C j)$ ) UNIV ( $\bigcup i. C i$ )  $\subseteq$  A  $\wedge$  i. infinite (C i)*

*<proof>*

**end**

## 22 Countable Complete Lattices

**theory** *Countable-Complete-Lattices*

**imports** *Main Countable-Set*

**begin**

**lemma** *UNIV-nat-eq*: *UNIV = insert 0 (range Suc)*

*<proof>*

**class** *countable-complete-lattice* = *lattice + Inf + Sup + bot + top +*

**assumes** *ccInf-lower*: *countable A  $\implies$  x  $\in$  A  $\implies$  Inf A  $\leq$  x*

**assumes** *ccInf-greatest*: *countable A  $\implies$  ( $\bigwedge x. x \in A \implies z \leq x$ )  $\implies$  z  $\leq$  Inf A*

**assumes** *ccSup-upper*: *countable A  $\implies$  x  $\in$  A  $\implies$  x  $\leq$  Sup A*

**assumes** *ccSup-least*: *countable A  $\implies$  ( $\bigwedge x. x \in A \implies x \leq z$ )  $\implies$  Sup A  $\leq$  z*

**assumes** *ccInf-empty [simp]*: *Inf {} = top*

**assumes** *ccSup-empty [simp]*: *Sup {} = bot*

**begin**

**subclass** *bounded-lattice*

*<proof>*

**lemma** *ccINF-lower*: *countable A  $\implies$  i  $\in$  A  $\implies$  (INF i  $\in$  A. f i)  $\leq$  f i*

*<proof>*

**lemma** *ccINF-greatest*: *countable A  $\implies$  ( $\bigwedge i. i \in A \implies u \leq f i$ )  $\implies$  u  $\leq$  (INF i  $\in$  A. f i)*

*<proof>*

**lemma** *ccSUP-upper*:  $\text{countable } A \implies i \in A \implies f i \leq (\text{SUP } i \in A. f i)$   
 ⟨proof⟩

**lemma** *ccSUP-least*:  $\text{countable } A \implies (\bigwedge i. i \in A \implies f i \leq u) \implies (\text{SUP } i \in A. f i) \leq u$   
 ⟨proof⟩

**lemma** *ccInf-lower2*:  $\text{countable } A \implies u \in A \implies u \leq v \implies \text{Inf } A \leq v$   
 ⟨proof⟩

**lemma** *ccINF-lower2*:  $\text{countable } A \implies i \in A \implies f i \leq u \implies (\text{INF } i \in A. f i) \leq u$   
 ⟨proof⟩

**lemma** *ccSup-upper2*:  $\text{countable } A \implies u \in A \implies v \leq u \implies v \leq \text{Sup } A$   
 ⟨proof⟩

**lemma** *ccSUP-upper2*:  $\text{countable } A \implies i \in A \implies u \leq f i \implies u \leq (\text{SUP } i \in A. f i)$   
 ⟨proof⟩

**lemma** *le-ccInf-iff*:  $\text{countable } A \implies b \leq \text{Inf } A \iff (\forall a \in A. b \leq a)$   
 ⟨proof⟩

**lemma** *le-ccINF-iff*:  $\text{countable } A \implies u \leq (\text{INF } i \in A. f i) \iff (\forall i \in A. u \leq f i)$   
 ⟨proof⟩

**lemma** *ccSup-le-iff*:  $\text{countable } A \implies \text{Sup } A \leq b \iff (\forall a \in A. a \leq b)$   
 ⟨proof⟩

**lemma** *ccSUP-le-iff*:  $\text{countable } A \implies (\text{SUP } i \in A. f i) \leq u \iff (\forall i \in A. f i \leq u)$   
 ⟨proof⟩

**lemma** *ccInf-insert [simp]*:  $\text{countable } A \implies \text{Inf } (\text{insert } a A) = \text{inf } a (\text{Inf } A)$   
 ⟨proof⟩

**lemma** *ccINF-insert [simp]*:  $\text{countable } A \implies (\text{INF } x \in \text{insert } a A. f x) = \text{inf } (f a) (\text{Inf } (f ` A))$   
 ⟨proof⟩

**lemma** *ccSup-insert [simp]*:  $\text{countable } A \implies \text{Sup } (\text{insert } a A) = \text{sup } a (\text{Sup } A)$   
 ⟨proof⟩

**lemma** *ccSUP-insert [simp]*:  $\text{countable } A \implies (\text{SUP } x \in \text{insert } a A. f x) = \text{sup } (f a) (\text{Sup } (f ` A))$   
 ⟨proof⟩

**lemma** *ccINF-empty [simp]*:  $(\text{INF } x \in \{\}. f x) = \text{top}$   
 ⟨proof⟩

**lemma** *ccSUP-empty* [*simp*]:  $(SUP\ x \in \{\}. f\ x) = bot$   
 ⟨*proof*⟩

**lemma** *ccInf-superset-mono*:  $countable\ A \implies B \subseteq A \implies Inf\ A \leq Inf\ B$   
 ⟨*proof*⟩

**lemma** *ccSup-subset-mono*:  $countable\ B \implies A \subseteq B \implies Sup\ A \leq Sup\ B$   
 ⟨*proof*⟩

**lemma** *ccInf-mono*:  
 assumes [*intro*]:  $countable\ B\ countable\ A$   
 assumes  $\bigwedge b. b \in B \implies \exists a \in A. a \leq b$   
 shows  $Inf\ A \leq Inf\ B$   
 ⟨*proof*⟩

**lemma** *ccINF-mono*:  
 $countable\ A \implies countable\ B \implies (\bigwedge m. m \in B \implies \exists n \in A. f\ n \leq g\ m) \implies (INF\ n \in A. f\ n) \leq (INF\ n \in B. g\ n)$   
 ⟨*proof*⟩

**lemma** *ccSup-mono*:  
 assumes [*intro*]:  $countable\ B\ countable\ A$   
 assumes  $\bigwedge a. a \in A \implies \exists b \in B. a \leq b$   
 shows  $Sup\ A \leq Sup\ B$   
 ⟨*proof*⟩

**lemma** *ccSUP-mono*:  
 $countable\ A \implies countable\ B \implies (\bigwedge n. n \in A \implies \exists m \in B. f\ n \leq g\ m) \implies (SUP\ n \in A. f\ n) \leq (SUP\ n \in B. g\ n)$   
 ⟨*proof*⟩

**lemma** *ccINF-superset-mono*:  
 $countable\ A \implies B \subseteq A \implies (\bigwedge x. x \in B \implies f\ x \leq g\ x) \implies (INF\ x \in A. f\ x) \leq (INF\ x \in B. g\ x)$   
 ⟨*proof*⟩

**lemma** *ccSUP-subset-mono*:  
 $countable\ B \implies A \subseteq B \implies (\bigwedge x. x \in A \implies f\ x \leq g\ x) \implies (SUP\ x \in A. f\ x) \leq (SUP\ x \in B. g\ x)$   
 ⟨*proof*⟩

**lemma** *less-eq-ccInf-inter*:  $countable\ A \implies countable\ B \implies sup\ (Inf\ A)\ (Inf\ B) \leq Inf\ (A \cap B)$   
 ⟨*proof*⟩

**lemma** *ccSup-inter-less-eq*:  $countable\ A \implies countable\ B \implies Sup\ (A \cap B) \leq inf\ (Sup\ A)\ (Sup\ B)$

*<proof>*

**lemma** *ccInf-union-distrib*:  $\text{countable } A \implies \text{countable } B \implies \text{Inf } (A \cup B) = \text{inf } (\text{Inf } A) (\text{Inf } B)$   
*<proof>*

**lemma** *ccINF-union*:  
 $\text{countable } A \implies \text{countable } B \implies (\text{INF } i \in A \cup B. M i) = \text{inf } (\text{INF } i \in A. M i) (\text{INF } i \in B. M i)$   
*<proof>*

**lemma** *ccSup-union-distrib*:  $\text{countable } A \implies \text{countable } B \implies \text{Sup } (A \cup B) = \text{sup } (\text{Sup } A) (\text{Sup } B)$   
*<proof>*

**lemma** *ccSUP-union*:  
 $\text{countable } A \implies \text{countable } B \implies (\text{SUP } i \in A \cup B. M i) = \text{sup } (\text{SUP } i \in A. M i) (\text{SUP } i \in B. M i)$   
*<proof>*

**lemma** *ccINF-inf-distrib*:  $\text{countable } A \implies \text{inf } (\text{INF } a \in A. f a) (\text{INF } a \in A. g a) = (\text{INF } a \in A. \text{inf } (f a) (g a))$   
*<proof>*

**lemma** *ccSUP-sup-distrib*:  $\text{countable } A \implies \text{sup } (\text{SUP } a \in A. f a) (\text{SUP } a \in A. g a) = (\text{SUP } a \in A. \text{sup } (f a) (g a))$   
*<proof>*

**lemma** *ccINF-const [simp]*:  $A \neq \{\} \implies (\text{INF } i \in A. f) = f$   
*<proof>*

**lemma** *ccSUP-const [simp]*:  $A \neq \{\} \implies (\text{SUP } i \in A. f) = f$   
*<proof>*

**lemma** *ccINF-top [simp]*:  $(\text{INF } x \in A. \text{top}) = \text{top}$   
*<proof>*

**lemma** *ccSUP-bot [simp]*:  $(\text{SUP } x \in A. \text{bot}) = \text{bot}$   
*<proof>*

**lemma** *ccINF-commute*:  $\text{countable } A \implies \text{countable } B \implies (\text{INF } i \in A. \text{INF } j \in B. f i j) = (\text{INF } j \in B. \text{INF } i \in A. f i j)$   
*<proof>*

**lemma** *ccSUP-commute*:  $\text{countable } A \implies \text{countable } B \implies (\text{SUP } i \in A. \text{SUP } j \in B. f i j) = (\text{SUP } j \in B. \text{SUP } i \in A. f i j)$   
*<proof>*

**end**

**context**

**fixes**  $a :: 'a::\{\text{countable-complete-lattice, linorder}\}$

**begin**

**lemma** *less-ccSup-iff*:  $\text{countable } S \implies a < \text{Sup } S \iff (\exists x \in S. a < x)$   
 ⟨proof⟩

**lemma** *less-ccSUP-iff*:  $\text{countable } A \implies a < (\text{SUP } i \in A. f i) \iff (\exists x \in A. a < f x)$   
 ⟨proof⟩

**lemma** *ccInf-less-iff*:  $\text{countable } S \implies \text{Inf } S < a \iff (\exists x \in S. x < a)$   
 ⟨proof⟩

**lemma** *ccINF-less-iff*:  $\text{countable } A \implies (\text{INF } i \in A. f i) < a \iff (\exists x \in A. f x < a)$   
 ⟨proof⟩

**end**

**class** *countable-complete-distrib-lattice* = *countable-complete-lattice* +

**assumes** *sup-ccInf*:  $\text{countable } B \implies \text{sup } a (\text{Inf } B) = (\text{INF } b \in B. \text{sup } a b)$

**assumes** *inf-ccSup*:  $\text{countable } B \implies \text{inf } a (\text{Sup } B) = (\text{SUP } b \in B. \text{inf } a b)$

**begin**

**lemma** *sup-ccINF*:

$\text{countable } B \implies \text{sup } a (\text{INF } b \in B. f b) = (\text{INF } b \in B. \text{sup } a (f b))$   
 ⟨proof⟩

**lemma** *inf-ccSUP*:

$\text{countable } B \implies \text{inf } a (\text{SUP } b \in B. f b) = (\text{SUP } b \in B. \text{inf } a (f b))$   
 ⟨proof⟩

**subclass** *distrib-lattice*

⟨proof⟩

**lemma** *ccInf-sup*:

$\text{countable } B \implies \text{sup } (\text{Inf } B) a = (\text{INF } b \in B. \text{sup } b a)$   
 ⟨proof⟩

**lemma** *ccSup-inf*:

$\text{countable } B \implies \text{inf } (\text{Sup } B) a = (\text{SUP } b \in B. \text{inf } b a)$   
 ⟨proof⟩

**lemma** *ccINF-sup*:

$\text{countable } B \implies \text{sup } (\text{INF } b \in B. f b) a = (\text{INF } b \in B. \text{sup } (f b) a)$   
 ⟨proof⟩

**lemma** *ccSUP-inf*:

$\text{countable } B \implies \text{inf } (\text{SUP } b \in B. f b) a = (\text{SUP } b \in B. \text{inf } (f b) a)$

*<proof>*

**lemma** *ccINF-sup-distrib2*:

*countable A  $\implies$  countable B  $\implies$  sup (INF a $\in$ A. f a) (INF b $\in$ B. g b) = (INF a $\in$ A. INF b $\in$ B. sup (f a) (g b))*

*<proof>*

**lemma** *ccSUP-inf-distrib2*:

*countable A  $\implies$  countable B  $\implies$  inf (SUP a $\in$ A. f a) (SUP b $\in$ B. g b) = (SUP a $\in$ A. SUP b $\in$ B. inf (f a) (g b))*

*<proof>*

**context**

**fixes** *f :: 'a  $\Rightarrow$  'b::countable-complete-lattice*

**assumes** *mono f*

**begin**

**lemma** *mono-ccInf*:

*countable A  $\implies$  f (Inf A)  $\leq$  (INF x $\in$ A. f x)*

*<proof>*

**lemma** *mono-ccSup*:

*countable A  $\implies$  (SUP x $\in$ A. f x)  $\leq$  f (Sup A)*

*<proof>*

**lemma** *mono-ccINF*:

*countable I  $\implies$  f (INF i  $\in$  I. A i)  $\leq$  (INF x  $\in$  I. f (A x))*

*<proof>*

**lemma** *mono-ccSUP*:

*countable I  $\implies$  (SUP x  $\in$  I. f (A x))  $\leq$  f (SUP i  $\in$  I. A i)*

*<proof>*

**end**

**end**

### 22.0.1 Instances of countable complete lattices

**instance** *fun :: (type, countable-complete-lattice) countable-complete-lattice*

*<proof>*

**subclass** (**in** *complete-lattice*) *countable-complete-lattice*

*<proof>*

**subclass** (**in** *complete-distrib-lattice*) *countable-complete-distrib-lattice*

*<proof>*

**end**

## 23 Type of (at Most) Countable Sets

```
theory Countable-Set-Type
imports Countable-Set
begin
```

### 23.1 Cardinal stuff

```
context
  includes cardinal-syntax
begin
```

```
lemma countable-card-of-nat: countable A  $\longleftrightarrow$   $|A| \leq_o |UNIV::nat set|$ 
  <proof>
```

```
lemma countable-card-le-natLeq: countable A  $\longleftrightarrow$   $|A| \leq_o natLeq$ 
  <proof>
```

```
lemma countable-or-card-of:
  assumes countable A
  shows (finite A  $\wedge$   $|A| <_o |UNIV::nat set|$ )  $\vee$ 
        (infinite A  $\wedge$   $|A| =_o |UNIV::nat set|$ )
  <proof>
```

```
lemma countable-cases-card-of[elim]:
  assumes countable A
  obtains (Fin) finite A  $|A| <_o |UNIV::nat set|$ 
         | (Inf) infinite A  $|A| =_o |UNIV::nat set|$ 
  <proof>
```

```
lemma countable-or:
  countable A  $\implies$  ( $\exists f::'a \Rightarrow nat.$  finite A  $\wedge$  inj-on f A)  $\vee$  ( $\exists f::'a \Rightarrow nat.$  infinite A
 $\wedge$  bij-betw f A UNIV)
  <proof>
```

```
lemma countable-cases[elim]:
  assumes countable A
  obtains (Fin) f :: 'a  $\Rightarrow$  nat where finite A inj-on f A
         | (Inf) f :: 'a  $\Rightarrow$  nat where infinite A bij-betw f A UNIV
  <proof>
```

```
lemma countable-ordLeq:
  assumes  $|A| \leq_o |B|$  and countable B
  shows countable A
  <proof>
```

```
lemma countable-ordLess:
  assumes AB:  $|A| <_o |B|$  and B: countable B
  shows countable A
  <proof>
```



end

## 23.2 The type of countable sets

**typedef** *'a cset* = {*A* :: *'a set. countable A*} **morphisms** *rcset acset*  
 ⟨*proof*⟩

**setup-lifting** *type-definition-cset*

**declare**

*rcset-inverse*[*simp*]  
*acset-inverse*[*Transfer.transferred, unfolded mem-Collect-eq, simp*]  
*acset-inject*[*Transfer.transferred, unfolded mem-Collect-eq, simp*]  
*rcset*[*Transfer.transferred, unfolded mem-Collect-eq, simp*]

**instantiation** *cset* :: (*type*) {*bounded-lattice-bot, distrib-lattice, minus*}  
**begin**

**lift-definition** *bot-cset* :: *'a cset* **is** {} **parametric** *empty-transfer* ⟨*proof*⟩

**lift-definition** *less-eq-cset* :: *'a cset* ⇒ *'a cset* ⇒ *bool*  
**is** *subset-eq* **parametric** *subset-transfer* ⟨*proof*⟩

**definition** *less-cset* :: *'a cset* ⇒ *'a cset* ⇒ *bool*  
**where** *xs < ys* ≡ *xs ≤ ys* ∧ *xs ≠ (ys::'a cset)*

**lemma** *less-cset-transfer*[*transfer-rule*]:

**includes** *lifting-syntax*  
**assumes** [*transfer-rule*]: *bi-unique A*  
**shows** ((*pcr-cset A*) ==> (*pcr-cset A*) ==> (=)) (C) (<)  
 ⟨*proof*⟩

**lift-definition** *sup-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*  
**is** *union* **parametric** *union-transfer* ⟨*proof*⟩

**lift-definition** *inf-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*  
**is** *inter* **parametric** *inter-transfer* ⟨*proof*⟩

**lift-definition** *minus-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*  
**is** *minus* **parametric** *Diff-transfer* ⟨*proof*⟩

**instance** ⟨*proof*⟩

end

**abbreviation** *empty* :: *'a cset* **where** *empty* ≡ *bot*

**abbreviation** *csubset-eq* :: *'a cset* ⇒ *'a cset* ⇒ *bool* **where** *csubset-eq xs ys* ≡ *xs*  
 ≤ *ys*

**abbreviation**  $csubset :: 'a \text{ cset} \Rightarrow 'a \text{ cset} \Rightarrow \text{bool}$  **where**  $csubset \ x \ y \equiv x < y$

**abbreviation**  $cUn :: 'a \text{ cset} \Rightarrow 'a \text{ cset} \Rightarrow 'a \text{ cset}$  **where**  $cUn \ x \ y \equiv \text{sup } x \ y$

**abbreviation**  $cInt :: 'a \text{ cset} \Rightarrow 'a \text{ cset} \Rightarrow 'a \text{ cset}$  **where**  $cInt \ x \ y \equiv \text{inf } x \ y$

**abbreviation**  $cDiff :: 'a \text{ cset} \Rightarrow 'a \text{ cset} \Rightarrow 'a \text{ cset}$  **where**  $cDiff \ x \ y \equiv \text{minus } x \ y$

**lift-definition**  $cin :: 'a \Rightarrow 'a \text{ cset} \Rightarrow \text{bool}$  **is**  $(\in)$  **parametric** *member-transfer*  
*<proof>*

**lift-definition**  $cinsert :: 'a \Rightarrow 'a \text{ cset} \Rightarrow 'a \text{ cset}$  **is** *insert* **parametric** *Lifting-Set.insert-transfer*  
*<proof>*

**abbreviation**  $csingle :: 'a \Rightarrow 'a \text{ cset}$  **where**  $csingle \ x \equiv cinsert \ x \ \text{empty}$

**lift-definition**  $cimage :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ cset} \Rightarrow 'b \text{ cset}$  **is**  $(\cdot)$  **parametric** *image-transfer*  
*<proof>*

**lift-definition**  $cBall :: 'a \text{ cset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$  **is** *Ball* **parametric** *Ball-transfer*  
*<proof>*

**lift-definition**  $cBex :: 'a \text{ cset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$  **is** *Bex* **parametric** *Bex-transfer*  
*<proof>*

**lift-definition**  $cUnion :: 'a \text{ cset} \text{ cset} \Rightarrow 'a \text{ cset}$  **is** *Union* **parametric** *Union-transfer*  
*<proof>*

**abbreviation** (*input*)  $cUNION :: 'a \text{ cset} \Rightarrow ('a \Rightarrow 'b \text{ cset}) \Rightarrow 'b \text{ cset}$   
**where**  $cUNION \ A \ f \equiv cUnion \ (cimage \ f \ A)$

**lemma** *Union-conv-UNION*:  $\bigcup A = \bigcup (id \cdot A)$   
*<proof>*

**lemmas**  $cset\text{-eq}I = \text{set}\text{-eq}I[\text{Transfer.transferred}]$

**lemmas**  $cset\text{-eq}\text{-iff}[no\text{-atp}] = \text{set}\text{-eq}\text{-iff}[\text{Transfer.transferred}]$

**lemmas**  $cBall[intro!] = ball[\text{Transfer.transferred}]$

**lemmas**  $cbspec[dest?] = bspec[\text{Transfer.transferred}]$

**lemmas**  $cBallE[elim] = ballE[\text{Transfer.transferred}]$

**lemmas**  $cBexI[intro] = bexI[\text{Transfer.transferred}]$

**lemmas**  $rev\text{-}cBexI[intro?] = rev\text{-}bexI[\text{Transfer.transferred}]$

**lemmas**  $cBexCI = bexCI[\text{Transfer.transferred}]$

**lemmas**  $cBexE[elim!] = bexE[\text{Transfer.transferred}]$

**lemmas**  $cBall\text{-triv}[simp] = ball\text{-triv}[\text{Transfer.transferred}]$

**lemmas**  $cBex\text{-triv}[simp] = bex\text{-triv}[\text{Transfer.transferred}]$

**lemmas**  $cBex\text{-triv}\text{-one}\text{-point}1[simp] = bex\text{-triv}\text{-one}\text{-point}1[\text{Transfer.transferred}]$

**lemmas**  $cBex\text{-triv}\text{-one}\text{-point}2[simp] = bex\text{-triv}\text{-one}\text{-point}2[\text{Transfer.transferred}]$

**lemmas**  $cBex\text{-one}\text{-point}1[simp] = bex\text{-one}\text{-point}1[\text{Transfer.transferred}]$

**lemmas**  $cBex\text{-one}\text{-point}2[simp] = bex\text{-one}\text{-point}2[\text{Transfer.transferred}]$

**lemmas**  $cBall\text{-one}\text{-point}1[simp] = ball\text{-one}\text{-point}1[\text{Transfer.transferred}]$

**lemmas**  $cBall\text{-one}\text{-point}2[simp] = ball\text{-one}\text{-point}2[\text{Transfer.transferred}]$

**lemmas**  $cBall\text{-conj}\text{-distrib} = ball\text{-conj}\text{-distrib}[\text{Transfer.transferred}]$

**lemmas**  $cBex\text{-disj}\text{-distrib} = bex\text{-disj}\text{-distrib}[\text{Transfer.transferred}]$

**lemmas**  $cBall\text{-cong} = ball\text{-cong}[\text{Transfer.transferred}]$

**lemmas**  $cBex\text{-cong} = bex\text{-cong}[\text{Transfer.transferred}]$

**lemmas**  $csubsetI[intro!] = subsetI[\text{Transfer.transferred}]$

**lemmas**  $csubsetD[elim, intro?] = subsetD[\text{Transfer.transferred}]$

**lemmas**  $rev-csubsetD[no-atp,intro?] = rev-subsetD[Transfer.transferred]$   
**lemmas**  $csubsetCE[no-atp,elim] = subsetCE[Transfer.transferred]$   
**lemmas**  $csubset-eq[no-atp] = subset-eq[Transfer.transferred]$   
**lemmas**  $contra-csubsetD[no-atp] = contra-subsetD[Transfer.transferred]$   
**lemmas**  $csubset-refl = subset-refl[Transfer.transferred]$   
**lemmas**  $csubset-trans = subset-trans[Transfer.transferred]$   
**lemmas**  $cset-rev-mp = rev-subsetD[Transfer.transferred]$   
**lemmas**  $cset-mp = subsetD[Transfer.transferred]$   
**lemmas**  $csubset-not-fsubset-eq[code] = subset-not-subset-eq[Transfer.transferred]$   
**lemmas**  $eq-cmem-trans = eq-mem-trans[Transfer.transferred]$   
**lemmas**  $csubset-antisym[intro!] = subset-antisym[Transfer.transferred]$   
**lemmas**  $cequalityD1 = equalityD1[Transfer.transferred]$   
**lemmas**  $cequalityD2 = equalityD2[Transfer.transferred]$   
**lemmas**  $cequalityE = equalityE[Transfer.transferred]$   
**lemmas**  $cequalityCE[elim] = equalityCE[Transfer.transferred]$   
**lemmas**  $eqcset-imp-iff = eqset-imp-iff[Transfer.transferred]$   
**lemmas**  $eqelem-imp-iff = eqelem-imp-iff[Transfer.transferred]$   
**lemmas**  $cempty-iff[simp] = empty-iff[Transfer.transferred]$   
**lemmas**  $cempty-fsubsetI[iff] = empty-subsetI[Transfer.transferred]$   
**lemmas**  $equals-cemptyI = equalsOI[Transfer.transferred]$   
**lemmas**  $equals-cemptyD = equalsOD[Transfer.transferred]$   
**lemmas**  $cBall-cempty[simp] = ball-empty[Transfer.transferred]$   
**lemmas**  $cBex-cempty[simp] = bex-empty[Transfer.transferred]$   
**lemmas**  $cInt-iff[simp] = Int-iff[Transfer.transferred]$   
**lemmas**  $cIntI[intro!] = IntI[Transfer.transferred]$   
**lemmas**  $cIntD1 = IntD1[Transfer.transferred]$   
**lemmas**  $cIntD2 = IntD2[Transfer.transferred]$   
**lemmas**  $cIntE[elim!] = IntE[Transfer.transferred]$   
**lemmas**  $cUn-iff[simp] = Un-iff[Transfer.transferred]$   
**lemmas**  $cUnI1[elim?] = UnI1[Transfer.transferred]$   
**lemmas**  $cUnI2[elim?] = UnI2[Transfer.transferred]$   
**lemmas**  $cUnCI[intro!] = UnCI[Transfer.transferred]$   
**lemmas**  $cuUnE[elim!] = UnE[Transfer.transferred]$   
**lemmas**  $cDiff-iff[simp] = Diff-iff[Transfer.transferred]$   
**lemmas**  $cDiffI[intro!] = DiffI[Transfer.transferred]$   
**lemmas**  $cDiffD1 = DiffD1[Transfer.transferred]$   
**lemmas**  $cDiffD2 = DiffD2[Transfer.transferred]$   
**lemmas**  $cDiffE[elim!] = DiffE[Transfer.transferred]$   
**lemmas**  $cinsert-iff[simp] = insert-iff[Transfer.transferred]$   
**lemmas**  $cinsertI1 = insertI1[Transfer.transferred]$   
**lemmas**  $cinsertI2 = insertI2[Transfer.transferred]$   
**lemmas**  $cinsertE[elim!] = insertE[Transfer.transferred]$   
**lemmas**  $cinsertCI[intro!] = insertCI[Transfer.transferred]$   
**lemmas**  $csubset-cinsert-iff = subset-insert-iff[Transfer.transferred]$   
**lemmas**  $cinsert-ident = insert-ident[Transfer.transferred]$   
**lemmas**  $csingletonI[intro!,no-atp] = singletonI[Transfer.transferred]$   
**lemmas**  $csingletonD[dest!,no-atp] = singletonD[Transfer.transferred]$   
**lemmas**  $fsingletonE = csingletonD [elim-format]$   
**lemmas**  $csingleton-iff = singleton-iff[Transfer.transferred]$

**lemmas** *csingleton-inject*[*dest!*] = *singleton-inject*[*Transfer.transferred*]  
**lemmas** *csingleton-finsert-inj-eq*[*iff,no-atp*] = *singleton-insert-inj-eq*[*Transfer.transferred*]  
**lemmas** *csingleton-finsert-inj-eq'*[*iff,no-atp*] = *singleton-insert-inj-eq'*[*Transfer.transferred*]  
**lemmas** *csubset-csingletonD* = *subset-singletonD*[*Transfer.transferred*]  
**lemmas** *cDiff-single-cinsert* = *Diff-single-insert*[*Transfer.transferred*]  
**lemmas** *cdoubleton-eq-iff* = *doubleton-eq-iff*[*Transfer.transferred*]  
**lemmas** *cUn-csingleton-iff* = *Un-singleton-iff*[*Transfer.transferred*]  
**lemmas** *csingleton-cUn-iff* = *singleton-Un-iff*[*Transfer.transferred*]  
**lemmas** *cimage-eqI*[*simp, intro*] = *image-eqI*[*Transfer.transferred*]  
**lemmas** *cimageI* = *imageI*[*Transfer.transferred*]  
**lemmas** *rev-cimage-eqI* = *rev-image-eqI*[*Transfer.transferred*]  
**lemmas** *cimageE*[*elim!*] = *imageE*[*Transfer.transferred*]  
**lemmas** *Compr-cimage-eq* = *Compr-image-eq*[*Transfer.transferred*]  
**lemmas** *cimage-cUn* = *image-Un*[*Transfer.transferred*]  
**lemmas** *cimage-iff* = *image-iff*[*Transfer.transferred*]  
**lemmas** *cimage-csubset-iff*[*no-atp*] = *image-subset-iff*[*Transfer.transferred*]  
**lemmas** *cimage-csubsetI* = *image-subsetI*[*Transfer.transferred*]  
**lemmas** *cimage-ident*[*simp*] = *image-ident*[*Transfer.transferred*]  
**lemmas** *if-split-cin1* = *if-split-mem1*[*Transfer.transferred*]  
**lemmas** *if-split-cin2* = *if-split-mem2*[*Transfer.transferred*]  
**lemmas** *cpsubsetI*[*intro!,no-atp*] = *psubsetI*[*Transfer.transferred*]  
**lemmas** *cpsubsetE*[*elim!,no-atp*] = *psubsetE*[*Transfer.transferred*]  
**lemmas** *cpsubset-finsert-iff* = *psubset-insert-iff*[*Transfer.transferred*]  
**lemmas** *cpsubset-eq* = *psubset-eq*[*Transfer.transferred*]  
**lemmas** *cpsubset-imp-fsubset* = *psubset-imp-subset*[*Transfer.transferred*]  
**lemmas** *cpsubset-trans* = *psubset-trans*[*Transfer.transferred*]  
**lemmas** *cpsubsetD* = *psubsetD*[*Transfer.transferred*]  
**lemmas** *cpsubset-csubset-trans* = *psubset-subset-trans*[*Transfer.transferred*]  
**lemmas** *csubset-cpsubset-trans* = *subset-psubset-trans*[*Transfer.transferred*]  
**lemmas** *cpsubset-imp-ex-fmem* = *psubset-imp-ex-mem*[*Transfer.transferred*]  
**lemmas** *csubset-cinsertI* = *subset-insertI*[*Transfer.transferred*]  
**lemmas** *csubset-cinsertI2* = *subset-insertI2*[*Transfer.transferred*]  
**lemmas** *csubset-cinsert* = *subset-insert*[*Transfer.transferred*]  
**lemmas** *cUn-upper1* = *Un-upper1*[*Transfer.transferred*]  
**lemmas** *cUn-upper2* = *Un-upper2*[*Transfer.transferred*]  
**lemmas** *cUn-least* = *Un-least*[*Transfer.transferred*]  
**lemmas** *cInt-lower1* = *Int-lower1*[*Transfer.transferred*]  
**lemmas** *cInt-lower2* = *Int-lower2*[*Transfer.transferred*]  
**lemmas** *cInt-greatest* = *Int-greatest*[*Transfer.transferred*]  
**lemmas** *cDiff-csubset* = *Diff-subset*[*Transfer.transferred*]  
**lemmas** *cDiff-csubset-conv* = *Diff-subset-conv*[*Transfer.transferred*]  
**lemmas** *csubset-cempty*[*simp*] = *subset-empty*[*Transfer.transferred*]  
**lemmas** *not-cpsubset-cempty*[*iff*] = *not-psubset-empty*[*Transfer.transferred*]  
**lemmas** *cinsert-is-cUn* = *insert-is-Un*[*Transfer.transferred*]  
**lemmas** *cinsert-not-cempty*[*simp*] = *insert-not-empty*[*Transfer.transferred*]  
**lemmas** *cempty-not-cinsert* = *empty-not-insert*[*Transfer.transferred*]  
**lemmas** *cinsert-absorb* = *insert-absorb*[*Transfer.transferred*]  
**lemmas** *cinsert-absorb2*[*simp*] = *insert-absorb2*[*Transfer.transferred*]  
**lemmas** *cinsert-commute* = *insert-commute*[*Transfer.transferred*]

**lemmas**  $cinsert-csubset[simp] = insert-subset[Transfer.transferred]$   
**lemmas**  $cinsert-cinter-cinsert[simp] = insert-inter-insert[Transfer.transferred]$   
**lemmas**  $cinsert-disjoint[simp,no-atp] = insert-disjoint[Transfer.transferred]$   
**lemmas**  $disjoint-cinsert[simp,no-atp] = disjoint-insert[Transfer.transferred]$   
**lemmas**  $cimage-cempty[simp] = image-empty[Transfer.transferred]$   
**lemmas**  $cimage-cinsert[simp] = image-insert[Transfer.transferred]$   
**lemmas**  $cimage-constant = image-constant[Transfer.transferred]$   
**lemmas**  $cimage-constant-conv = image-constant-conv[Transfer.transferred]$   
**lemmas**  $cimage-cimage = image-image[Transfer.transferred]$   
**lemmas**  $cinsert-cimage[simp] = insert-image[Transfer.transferred]$   
**lemmas**  $cimage-is-cempty[iff] = image-is-empty[Transfer.transferred]$   
**lemmas**  $cempty-is-cimage[iff] = empty-is-image[Transfer.transferred]$   
**lemmas**  $cimage-cong = image-cong[Transfer.transferred]$   
**lemmas**  $cimage-cInt-csubset = image-Int-subset[Transfer.transferred]$   
**lemmas**  $cimage-cDiff-csubset = image-diff-subset[Transfer.transferred]$   
**lemmas**  $cInt-absorb = Int-absorb[Transfer.transferred]$   
**lemmas**  $cInt-left-absorb = Int-left-absorb[Transfer.transferred]$   
**lemmas**  $cInt-commute = Int-commute[Transfer.transferred]$   
**lemmas**  $cInt-left-commute = Int-left-commute[Transfer.transferred]$   
**lemmas**  $cInt-assoc = Int-assoc[Transfer.transferred]$   
**lemmas**  $cInt-ac = Int-ac[Transfer.transferred]$   
**lemmas**  $cInt-absorb1 = Int-absorb1[Transfer.transferred]$   
**lemmas**  $cInt-absorb2 = Int-absorb2[Transfer.transferred]$   
**lemmas**  $cInt-cempty-left = Int-empty-left[Transfer.transferred]$   
**lemmas**  $cInt-cempty-right = Int-empty-right[Transfer.transferred]$   
**lemmas**  $disjoint-iff-cnot-equal = disjoint-iff-not-equal[Transfer.transferred]$   
**lemmas**  $cInt-cUn-distrib = Int-Un-distrib[Transfer.transferred]$   
**lemmas**  $cInt-cUn-distrib2 = Int-Un-distrib2[Transfer.transferred]$   
**lemmas**  $cInt-csubset-iff[no-atp, simp] = Int-subset-iff[Transfer.transferred]$   
**lemmas**  $cUn-absorb = Un-absorb[Transfer.transferred]$   
**lemmas**  $cUn-left-absorb = Un-left-absorb[Transfer.transferred]$   
**lemmas**  $cUn-commute = Un-commute[Transfer.transferred]$   
**lemmas**  $cUn-left-commute = Un-left-commute[Transfer.transferred]$   
**lemmas**  $cUn-assoc = Un-assoc[Transfer.transferred]$   
**lemmas**  $cUn-ac = Un-ac[Transfer.transferred]$   
**lemmas**  $cUn-absorb1 = Un-absorb1[Transfer.transferred]$   
**lemmas**  $cUn-absorb2 = Un-absorb2[Transfer.transferred]$   
**lemmas**  $cUn-cempty-left = Un-empty-left[Transfer.transferred]$   
**lemmas**  $cUn-cempty-right = Un-empty-right[Transfer.transferred]$   
**lemmas**  $cUn-cinsert-left[simp] = Un-insert-left[Transfer.transferred]$   
**lemmas**  $cUn-cinsert-right[simp] = Un-insert-right[Transfer.transferred]$   
**lemmas**  $cInt-cinsert-left = Int-insert-left[Transfer.transferred]$   
**lemmas**  $cInt-cinsert-left-if0[simp] = Int-insert-left-if0[Transfer.transferred]$   
**lemmas**  $cInt-cinsert-left-if1[simp] = Int-insert-left-if1[Transfer.transferred]$   
**lemmas**  $cInt-cinsert-right = Int-insert-right[Transfer.transferred]$   
**lemmas**  $cInt-cinsert-right-if0[simp] = Int-insert-right-if0[Transfer.transferred]$   
**lemmas**  $cInt-cinsert-right-if1[simp] = Int-insert-right-if1[Transfer.transferred]$   
**lemmas**  $cUn-cInt-distrib = Un-Int-distrib[Transfer.transferred]$   
**lemmas**  $cUn-cInt-distrib2 = Un-Int-distrib2[Transfer.transferred]$

**lemmas**  $cUn-cInt-crazy = Un-Int-crazy$ [Transfer.transferred]  
**lemmas**  $csubset-cUn-eq = subset-Un-eq$ [Transfer.transferred]  
**lemmas**  $cUn-cempty$ [iff] =  $Un-empty$ [Transfer.transferred]  
**lemmas**  $cUn-csubset-iff$ [no-atp, simp] =  $Un-subset-iff$ [Transfer.transferred]  
**lemmas**  $cUn-cDiff-cInt = Un-Diff-Int$ [Transfer.transferred]  
**lemmas**  $cDiff-cInt2 = Diff-Int2$ [Transfer.transferred]  
**lemmas**  $cUn-cInt-assoc-eq = Un-Int-assoc-eq$ [Transfer.transferred]  
**lemmas**  $cBall-cUn = ball-Un$ [Transfer.transferred]  
**lemmas**  $cBex-cUn = bex-Un$ [Transfer.transferred]  
**lemmas**  $cDiff-eq-cempty-iff$ [simp,no-atp] =  $Diff-eq-empty-iff$ [Transfer.transferred]  
**lemmas**  $cDiff-cancel$ [simp] =  $Diff-cancel$ [Transfer.transferred]  
**lemmas**  $cDiff-idemp$ [simp] =  $Diff-idemp$ [Transfer.transferred]  
**lemmas**  $cDiff-triv = Diff-triv$ [Transfer.transferred]  
**lemmas**  $cempty-cDiff$ [simp] =  $empty-Diff$ [Transfer.transferred]  
**lemmas**  $cDiff-cempty$ [simp] =  $Diff-empty$ [Transfer.transferred]  
**lemmas**  $cDiff-cinsert0$ [simp,no-atp] =  $Diff-insert0$ [Transfer.transferred]  
**lemmas**  $cDiff-cinsert = Diff-insert$ [Transfer.transferred]  
**lemmas**  $cDiff-cinsert2 = Diff-insert2$ [Transfer.transferred]  
**lemmas**  $cinsert-cDiff-if = insert-Diff-if$ [Transfer.transferred]  
**lemmas**  $cinsert-cDiff1$ [simp] =  $insert-Diff1$ [Transfer.transferred]  
**lemmas**  $cinsert-cDiff-single$ [simp] =  $insert-Diff-single$ [Transfer.transferred]  
**lemmas**  $cinsert-cDiff = insert-Diff$ [Transfer.transferred]  
**lemmas**  $cDiff-cinsert-absorb = Diff-insert-absorb$ [Transfer.transferred]  
**lemmas**  $cDiff-disjoint$ [simp] =  $Diff-disjoint$ [Transfer.transferred]  
**lemmas**  $cDiff-partition = Diff-partition$ [Transfer.transferred]  
**lemmas**  $double-cDiff = double-diff$ [Transfer.transferred]  
**lemmas**  $cUn-cDiff-cancel$ [simp] =  $Un-Diff-cancel$ [Transfer.transferred]  
**lemmas**  $cUn-cDiff-cancel2$ [simp] =  $Un-Diff-cancel2$ [Transfer.transferred]  
**lemmas**  $cDiff-cUn = Diff-Un$ [Transfer.transferred]  
**lemmas**  $cDiff-cInt = Diff-Int$ [Transfer.transferred]  
**lemmas**  $cUn-cDiff = Un-Diff$ [Transfer.transferred]  
**lemmas**  $cInt-cDiff = Int-Diff$ [Transfer.transferred]  
**lemmas**  $cDiff-cInt-distrib = Diff-Int-distrib$ [Transfer.transferred]  
**lemmas**  $cDiff-cInt-distrib2 = Diff-Int-distrib2$ [Transfer.transferred]  
**lemmas**  $cset-eq-csubset = set-eq-subset$ [Transfer.transferred]  
**lemmas**  $csubset-iff$ [no-atp] =  $subset-iff$ [Transfer.transferred]  
**lemmas**  $csubset-iff-psubset-eq = subset-iff-psubset-eq$ [Transfer.transferred]  
**lemmas**  $all-not-cin-conv$ [simp] =  $all-not-in-conv$ [Transfer.transferred]  
**lemmas**  $ex-cin-conv = ex-in-conv$ [Transfer.transferred]  
**lemmas**  $cimage-mono = image-mono$ [Transfer.transferred]  
**lemmas**  $cinsert-mono = insert-mono$ [Transfer.transferred]  
**lemmas**  $cunion-mono = Un-mono$ [Transfer.transferred]  
**lemmas**  $cinter-mono = Int-mono$ [Transfer.transferred]  
**lemmas**  $cminus-mono = Diff-mono$ [Transfer.transferred]  
**lemmas**  $cin-mono = in-mono$ [Transfer.transferred]  
**lemmas**  $cLeast-mono = Least-mono$ [Transfer.transferred]  
**lemmas**  $cequalityI = equalityI$ [Transfer.transferred]  
**lemmas**  $cUN-iff$  [simp] =  $UN-iff$ [Transfer.transferred]  
**lemmas**  $cUN-I$  [intro] =  $UN-I$ [Transfer.transferred]

**lemmas**  $cUN-E$  [elim!] =  $UN-E$ [Transfer.transferred]  
**lemmas**  $cUN-upper$  =  $UN-upper$ [Transfer.transferred]  
**lemmas**  $cUN-least$  =  $UN-least$ [Transfer.transferred]  
**lemmas**  $cUN-cinsert-distrib$  =  $UN-insert-distrib$ [Transfer.transferred]  
**lemmas**  $cUN-empty$  [simp] =  $UN-empty$ [Transfer.transferred]  
**lemmas**  $cUN-empty2$  [simp] =  $UN-empty2$ [Transfer.transferred]  
**lemmas**  $cUN-absorb$  =  $UN-absorb$ [Transfer.transferred]  
**lemmas**  $cUN-cinsert$  [simp] =  $UN-insert$ [Transfer.transferred]  
**lemmas**  $cUN-cUn$  [simp] =  $UN-Un$ [Transfer.transferred]  
**lemmas**  $cUN-cUN-flatten$  =  $UN-UN-flatten$ [Transfer.transferred]  
**lemmas**  $cUN-csubset-iff$  =  $UN-subset-iff$ [Transfer.transferred]  
**lemmas**  $cUN-constant$  [simp] =  $UN-constant$ [Transfer.transferred]  
**lemmas**  $cimage-cUnion$  =  $image-Union$ [Transfer.transferred]  
**lemmas**  $cUNION-cempty-conv$  [simp] =  $UNION-empty-conv$ [Transfer.transferred]  
**lemmas**  $cBall-cUN$  =  $ball-UN$ [Transfer.transferred]  
**lemmas**  $cBex-cUN$  =  $bex-UN$ [Transfer.transferred]  
**lemmas**  $cUn-eq-cUN$  =  $Un-eq-UN$ [Transfer.transferred]  
**lemmas**  $cUN-mono$  =  $UN-mono$ [Transfer.transferred]  
**lemmas**  $cimage-cUN$  =  $image-UN$ [Transfer.transferred]  
**lemmas**  $cUN-csingleton$  [simp] =  $UN-singleton$ [Transfer.transferred]

### 23.3 Additional lemmas

#### 23.3.1 empty

**lemma**  $emptyE$  [elim!]:  $cin\ a\ empty \implies P$  ⟨proof⟩

#### 23.3.2 cinsert

**lemma**  $countable-insert-iff$ :  $countable\ (insert\ x\ A) \longleftrightarrow countable\ A$   
 ⟨proof⟩

**lemma**  $set-cinsert$ :

**assumes**  $cin\ x\ A$

**obtains**  $B$  **where**  $A = cinsert\ x\ B$  **and**  $\neg\ cin\ x\ B$

⟨proof⟩

**lemma**  $mk-disjoint-cinsert$ :  $cin\ a\ A \implies \exists B. A = cinsert\ a\ B \wedge \neg\ cin\ a\ B$   
 ⟨proof⟩

#### 23.3.3 cimage

**lemma**  $subset-cimage-iff$ :  $csubset-eq\ B\ (cimage\ f\ A) \longleftrightarrow (\exists AA. csubset-eq\ AA\ A \wedge B = cimage\ f\ AA)$   
 ⟨proof⟩

#### 23.3.4 bounded quantification

**lemma**  $cBex-simps$  [simp, no-atp]:

$\bigwedge A\ P\ Q. cBex\ A\ (\lambda x. P\ x \wedge Q) = (cBex\ A\ P \wedge Q)$

$$\begin{aligned}
& \bigwedge A P Q. cBex A (\lambda x. P \wedge Q x) = (P \wedge cBex A Q) \\
& \bigwedge P. cBex empty P = False \\
& \bigwedge a B P. cBex (cinsert a B) P = (P a \vee cBex B P) \\
& \bigwedge A P f. cBex (cimage f A) P = cBex A (\lambda x. P (f x)) \\
& \bigwedge A P. (\neg cBex A P) = cBall A (\lambda x. \neg P x) \\
\langle proof \rangle
\end{aligned}$$

**lemma** *cBall-simps* [*simp, no-atp*]:

$$\begin{aligned}
& \bigwedge A P Q. cBall A (\lambda x. P x \vee Q) = (cBall A P \vee Q) \\
& \bigwedge A P Q. cBall A (\lambda x. P \vee Q x) = (P \vee cBall A Q) \\
& \bigwedge A P Q. cBall A (\lambda x. P \longrightarrow Q x) = (P \longrightarrow cBall A Q) \\
& \bigwedge A P Q. cBall A (\lambda x. P x \longrightarrow Q) = (cBex A P \longrightarrow Q) \\
& \bigwedge P. cBall empty P = True \\
& \bigwedge a B P. cBall (cinsert a B) P = (P a \wedge cBall B P) \\
& \bigwedge A P f. cBall (cimage f A) P = cBall A (\lambda x. P (f x)) \\
& \bigwedge A P. (\neg cBall A P) = cBex A (\lambda x. \neg P x) \\
\langle proof \rangle
\end{aligned}$$

**lemma** *atomize-cBall*:

$$(\bigwedge x. cin x A \implies P x) == Trueprop (cBall A (\lambda x. P x))$$

*<proof>*

### 23.3.5 *cUnion*

**lemma** *cUNION-cimage*:  $cUNION (cimage f A) g = cUNION A (g \circ f)$   
*<proof>*

## 23.4 Setup for Lifting/Transfer

### 23.4.1 Relator and predicator properties

**lift-definition** *rel-cset* ::  $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \text{ cset} \Rightarrow 'b \text{ cset} \Rightarrow bool$   
 is *rel-set parametric rel-set-transfer* *<proof>*

**lemma** *rel-cset-alt-def*:

$$\begin{aligned}
& rel-cset R a b \iff \\
& (\forall t \in rcset a. \exists u \in rcset b. R t u) \wedge \\
& (\forall t \in rcset b. \exists u \in rcset a. R u t) \\
\langle proof \rangle
\end{aligned}$$

**lemma** *rel-cset-iff*:

$$\begin{aligned}
& rel-cset R a b \iff \\
& (\forall t. cin t a \longrightarrow (\exists u. cin u b \wedge R t u)) \wedge \\
& (\forall t. cin t b \longrightarrow (\exists u. cin u a \wedge R u t)) \\
\langle proof \rangle
\end{aligned}$$

**lemma** *rel-cset-cUNION*:

$$\begin{aligned}
& \llbracket rel-cset Q A B; rel-fun Q (rel-cset R) f g \rrbracket \\
& \implies rel-cset R (cUnion (cimage f A)) (cUnion (cimage g B)) \\
\langle proof \rangle
\end{aligned}$$



**lemma** *rel-cset-csingle-iff* [*simp*]:  $rel-cset\ R\ (csingle\ x)\ (csingle\ y) \longleftrightarrow R\ x\ y$   
 ⟨*proof*⟩

### 23.4.2 Transfer rules for the Transfer package

Unconditional transfer rules

**context includes** *lifting-syntax*  
**begin**

**lemmas** *cempty-parametric* [*transfer-rule*] = *empty-transfer*[*Transfer.transferred*]

**lemma** *cinsert-parametric* [*transfer-rule*]:  
 $(A\ ==>\ rel-cset\ A\ ==>\ rel-cset\ A)\ cinsert\ cinsert$   
 ⟨*proof*⟩

**lemma** *cUn-parametric* [*transfer-rule*]:  
 $(rel-cset\ A\ ==>\ rel-cset\ A\ ==>\ rel-cset\ A)\ cUn\ cUn$   
 ⟨*proof*⟩

**lemma** *cUnion-parametric* [*transfer-rule*]:  
 $(rel-cset\ (rel-cset\ A)\ ==>\ rel-cset\ A)\ cUnion\ cUnion$   
 ⟨*proof*⟩

**lemma** *cimage-parametric* [*transfer-rule*]:  
 $((A\ ==>\ B)\ ==>\ rel-cset\ A\ ==>\ rel-cset\ B)\ cimage\ cimage$   
 ⟨*proof*⟩

**lemma** *cBall-parametric* [*transfer-rule*]:  
 $(rel-cset\ A\ ==>\ (A\ ==>\ (=))\ ==>\ (=))\ cBall\ cBall$   
 ⟨*proof*⟩

**lemma** *cBex-parametric* [*transfer-rule*]:  
 $(rel-cset\ A\ ==>\ (A\ ==>\ (=))\ ==>\ (=))\ cBex\ cBex$   
 ⟨*proof*⟩

**lemma** *rel-cset-parametric* [*transfer-rule*]:  
 $((A\ ==>\ B\ ==>\ (=))\ ==>\ rel-cset\ A\ ==>\ rel-cset\ B\ ==>\ (=))$   
*rel-cset rel-cset*  
 ⟨*proof*⟩

Rules requiring bi-unique, bi-total or right-total relations

**lemma** *cin-parametric* [*transfer-rule*]:  
 $bi-unique\ A\ ==>\ (A\ ==>\ rel-cset\ A\ ==>\ (=))\ cin\ cin$   
 ⟨*proof*⟩

**lemma** *cInt-parametric* [*transfer-rule*]:  
 $bi-unique\ A\ ==>\ (rel-cset\ A\ ==>\ rel-cset\ A\ ==>\ rel-cset\ A)\ cInt\ cInt$   
 ⟨*proof*⟩

**lemma** *cDiff-parametric* [transfer-rule]:

*bi-unique*  $A \implies (\text{rel-cset } A \implies \text{rel-cset } A \implies \text{rel-cset } A) \text{ cDiff cDiff}$   
 ⟨proof⟩

**lemma** *csubset-parametric* [transfer-rule]:

*bi-unique*  $A \implies (\text{rel-cset } A \implies \text{rel-cset } A \implies (=)) \text{ csubset-eq csubset-eq}$   
 ⟨proof⟩

**end**

**lifting-update** *cset.lifting*

**lifting-forget** *cset.lifting*

### 23.5 Registration as BNF

**context**

**includes** *cardinal-syntax*

**begin**

**lemma** *card-of-countable-sets-range*:

**fixes**  $A :: 'a \text{ set}$

**shows**  $|\{X. X \subseteq A \wedge \text{countable } X \wedge X \neq \{\}\}| \leq_o |\{f::\text{nat} \Rightarrow 'a. \text{range } f \subseteq A\}|$   
 ⟨proof⟩

**lemma** *card-of-countable-sets-Func*:

$|\{X. X \subseteq A \wedge \text{countable } X \wedge X \neq \{\}\}| \leq_o |A| \hat{c} \text{ natLeq}$

⟨proof⟩

**lemma** *ordLeq-countable-subsets*:

$|A| \leq_o |\{X. X \subseteq A \wedge \text{countable } X\}|$

⟨proof⟩

**end**

**lemma** *finite-countable-subset*:

*finite*  $\{X. X \subseteq A \wedge \text{countable } X\} \longleftrightarrow \text{finite } A$

⟨proof⟩

**lemma** *rcset-to-rcset*: *countable*  $A \implies \text{rcset } (\text{the-inv } \text{rcset } A) = A$

**including** *cset.lifting*

⟨proof⟩

**lemma** *Collect-Int-Times*:  $\{(x, y). R \ x \ y\} \cap A \times B = \{(x, y). R \ x \ y \wedge x \in A \wedge y \in B\}$

⟨proof⟩

**lemma** *rel-cset-aux*:

$$(\forall t \in \text{rcset } a. \exists u \in \text{rcset } b. R \ t \ u) \wedge (\forall t \in \text{rcset } b. \exists u \in \text{rcset } a. R \ u \ t) \longleftrightarrow$$

$$((\text{BNF-Def.Grp } \{x. \text{rcset } x \subseteq \{(a, b). R \ a \ b\}\} (\text{cimage fst}))^{-1-1} \text{ OO}$$

$$\text{BNF-Def.Grp } \{x. \text{rcset } x \subseteq \{(a, b). R \ a \ b\}\} (\text{cimage snd}) \ a \ b \ (\text{is } ?L = ?R)$$

*<proof>* **including** *cset.lifting*  
*<proof>*

**context****includes** *cardinal-syntax***begin****bnf** *'a cset**map: cimage**sets: rcset**bd: card-suc natLeq**wits: empty**rel: rel-cset**<proof>* **including** *cset.lifting* *<proof>* **including** *cset.lifting* *<proof>* **including** *cset.lifting* *<proof>***end****end**

## 24 Debugging facilities for code generated towards Isabelle/ML

**theory** *Debug***imports** *Main***begin****context****begin****qualified definition** *trace* :: *String.literal*  $\Rightarrow$  *unit* **where***[simp]: trace s = ()***qualified definition** *tracing* :: *String.literal*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a* **where***[simp]: tracing s = id***lemma** *[code]:**tracing s = (let u = trace s in id)**<proof>* **definition** *flush* :: *'a*  $\Rightarrow$  *unit* **where***[simp]: flush x = ()***qualified definition** *flushing* :: *'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *'b* **where***[simp]: flushing x = id***lemma** *[code, code-unfold]:*

```

flushing x = (let u = flush x in id)
⟨proof⟩ definition timing :: String.literal ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b where
[simp]: timing s f x = f x

end

code-printing
  constant Debug.trace ↪ (Eval) Output.tracing
| constant Debug.flush ↪ (Eval) Output.tracing/ (@{make'-string} -) — note
indirection via antiquotation
| constant Debug.timing ↪ (Eval) Timing.timeap'-msg

code-reserved (Eval) Output Timing

end

```

## 25 Sequence of Properties on Subsequences

```

theory Diagonal-Subsequence
imports Complex-Main
begin

locale subseqs =
  fixes P::nat⇒(nat⇒nat)⇒bool
  assumes ex-subseq:  $\bigwedge n s. \text{strict-mono } (s::\text{nat}\Rightarrow\text{nat}) \Longrightarrow \exists r'. \text{strict-mono } r' \wedge P n (s \circ r')$ 
begin

definition reduce where reduce s n = (SOME r'::nat⇒nat. strict-mono r'  $\wedge$  P n (s  $\circ$  r'))

lemma subseq-reduce[intro, simp]:
  strict-mono s  $\Longrightarrow$  strict-mono (reduce s n)
  ⟨proof⟩

lemma reduce-holds:
  strict-mono s  $\Longrightarrow$  P n (s  $\circ$  reduce s n)
  ⟨proof⟩

primrec seqseq :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat where
  seqseq 0 = id
| seqseq (Suc n) = seqseq n  $\circ$  reduce (seqseq n) n

lemma subseq-seqseq[intro, simp]: strict-mono (seqseq n)
  ⟨proof⟩

lemma seqseq-holds:
  P n (seqseq (Suc n))
  ⟨proof⟩

```

**definition**  $diagseq :: nat \Rightarrow nat$  **where**  $diagseq\ i = seqseq\ i\ i$

**lemma**  $diagseq\ mono: diagseq\ n < diagseq\ (Suc\ n)$   
 $\langle proof \rangle$

**lemma**  $subseq\ diagseq: strict\ mono\ diagseq$   
 $\langle proof \rangle$

**primrec**  $fold\ reduce$  **where**

$fold\ reduce\ n\ 0 = id$   
 $| fold\ reduce\ n\ (Suc\ k) = fold\ reduce\ n\ k \circ reduce\ (seqseq\ (n + k))\ (n + k)$

**lemma**  $subseq\ fold\ reduce[intro, simp]: strict\ mono\ (fold\ reduce\ n\ k)$   
 $\langle proof \rangle$

**lemma**  $ex\ subseq\ reduce\ index: seqseq\ (n + k) = seqseq\ n \circ fold\ reduce\ n\ k$   
 $\langle proof \rangle$

**lemma**  $seqseq\ fold\ reduce: seqseq\ n = fold\ reduce\ 0\ n$   
 $\langle proof \rangle$

**lemma**  $diagseq\ fold\ reduce: diagseq\ n = fold\ reduce\ 0\ n\ n$   
 $\langle proof \rangle$

**lemma**  $fold\ reduce\ add: fold\ reduce\ 0\ (m + n) = fold\ reduce\ 0\ m \circ fold\ reduce\ m\ n$   
 $\langle proof \rangle$

**lemma**  $diagseq\ add: diagseq\ (k + n) = (seqseq\ k \circ (fold\ reduce\ k\ n))\ (k + n)$   
 $\langle proof \rangle$

**lemma**  $diagseq\ sub:$   
**assumes**  $m \leq n$  **shows**  $diagseq\ n = (seqseq\ m \circ (fold\ reduce\ m\ (n - m)))\ n$   
 $\langle proof \rangle$

**lemma**  $subseq\ diagonal\ rest: strict\ mono\ (\lambda x. fold\ reduce\ k\ x\ (k + x))$   
 $\langle proof \rangle$

**lemma**  $diagseq\ seqseq: diagseq \circ ((+) k) = (seqseq\ k \circ (\lambda x. fold\ reduce\ k\ x\ (k + x)))$   
 $\langle proof \rangle$

**lemma**  $diagseq\ holds:$   
**assumes**  $subseq\ stable: \bigwedge r\ s\ n. strict\ mono\ r \Longrightarrow P\ n\ s \Longrightarrow P\ n\ (s \circ r)$   
**shows**  $P\ k\ (diagseq \circ ((+) (Suc\ k)))$   
 $\langle proof \rangle$

**end**

end

## 26 Common discrete functions

```
theory Discrete-Functions
imports Complex-Main
begin
```

### 26.1 Discrete logarithm

```
fun floor-log :: nat ⇒ nat
  where [simp del]: floor-log n = (if n < 2 then 0 else Suc (floor-log (n div 2)))
```

```
lemma floor-log-induct [consumes 1, case-names one double]:
```

```
  fixes n :: nat
```

```
  assumes n > 0
```

```
  assumes one: P 1
```

```
  assumes double:  $\bigwedge n. n \geq 2 \implies P (n \text{ div } 2) \implies P n$ 
```

```
  shows P n
```

```
⟨proof⟩
```

```
lemma floor-log-zero [simp]: floor-log 0 = 0
```

```
⟨proof⟩
```

```
lemma floor-log-one [simp]: floor-log 1 = 0
```

```
⟨proof⟩
```

```
lemma floor-log-Suc-zero [simp]: floor-log (Suc 0) = 0
```

```
⟨proof⟩
```

```
lemma floor-log-rec:  $n \geq 2 \implies \text{floor-log } n = \text{Suc } (\text{floor-log } (n \text{ div } 2))$ 
```

```
⟨proof⟩
```

```
lemma floor-log-twice [simp]:  $n \neq 0 \implies \text{floor-log } (2 * n) = \text{Suc } (\text{floor-log } n)$ 
```

```
⟨proof⟩
```

```
lemma floor-log-half [simp]:  $\text{floor-log } (n \text{ div } 2) = \text{floor-log } n - 1$ 
```

```
⟨proof⟩
```

```
lemma floor-log-power [simp]:  $\text{floor-log } (2 \wedge n) = n$ 
```

```
⟨proof⟩
```

```
lemma floor-log-mono: mono floor-log
```

```
⟨proof⟩
```

```
lemma floor-log-exp2-le:
```

```
  assumes n > 0
```

```
  shows  $2 \wedge \text{floor-log } n \leq n$ 
```

*<proof>*

**lemma** *floor-log-exp2-gt*:  $2 * 2^{\text{floor-log } n} > n$   
*<proof>*

**lemma** *floor-log-exp2-ge*:  $2 * 2^{\text{floor-log } n} \geq n$   
*<proof>*

**lemma** *floor-log-le-iff*:  $m \leq n \implies \text{floor-log } m \leq \text{floor-log } n$   
*<proof>*

**lemma** *floor-log-eqI*:  
**assumes**  $n > 0 \ 2^k \leq n \ n < 2 * 2^k$   
**shows**  $\text{floor-log } n = k$   
*<proof>*

**lemma** *floor-log-altdef*:  $\text{floor-log } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \text{nat } \lfloor \log 2 (\text{real-of-nat } n) \rfloor)$   
*<proof>*

## 26.2 Discrete square root

**definition** *floor-sqrt* ::  $\text{nat} \Rightarrow \text{nat}$   
**where**  $\text{floor-sqrt } n = \text{Max } \{m. m^2 \leq n\}$

**lemma** *floor-sqrt-aux*:  
**fixes**  $n :: \text{nat}$   
**shows**  $\text{finite } \{m. m^2 \leq n\}$  **and**  $\{m. m^2 \leq n\} \neq \{\}$   
*<proof>*

**lemma** *floor-sqrt-unique*:  
**assumes**  $m^2 \leq n \ n < (\text{Suc } m)^2$   
**shows**  $\text{floor-sqrt } n = m$   
*<proof>*

**lemma** *floor-sqrt-code*[code]:  $\text{floor-sqrt } n = \text{Max } (\text{Set.filter } (\lambda m. m^2 \leq n) \{0..n\})$   
*<proof>*

**lemma** *floor-sqrt-inverse-power2* [simp]:  $\text{floor-sqrt } (n^2) = n$   
*<proof>*

**lemma** *floor-sqrt-zero* [simp]:  $\text{floor-sqrt } 0 = 0$   
*<proof>*

**lemma** *floor-sqrt-one* [simp]:  $\text{floor-sqrt } 1 = 1$   
*<proof>*

**lemma** *mono-floor-sqrt*: *mono floor-sqrt*

*<proof>*

**lemma** *mono-floor-sqrt'*:  $m \leq n \implies \text{floor-sqrt } m \leq \text{floor-sqrt } n$   
*<proof>*

**lemma** *floor-sqrt-greater-zero-iff* [*simp*]:  $\text{floor-sqrt } n > 0 \iff n > 0$   
*<proof>*

**lemma** *floor-sqrt-power2-le* [*simp*]:  $(\text{floor-sqrt } n)^2 \leq n$   
*<proof>*

**lemma** *floor-sqrt-le*:  $\text{floor-sqrt } n \leq n$   
*<proof>*

Additional facts about the discrete square root, thanks to Julian Bien-darra, Manuel Eberl

**lemma** *Suc-floor-sqrt-power2-gt*:  $n < (\text{Suc } (\text{floor-sqrt } n))^2$   
*<proof>*

**lemma** *le-floor-sqrt-iff*:  $x \leq \text{floor-sqrt } y \iff x^2 \leq y$   
*<proof>*

**lemma** *le-floor-sqrtI*:  $x^2 \leq y \implies x \leq \text{floor-sqrt } y$   
*<proof>*

**lemma** *floor-sqrt-le-iff*:  $\text{floor-sqrt } y \leq x \iff (\forall z. z^2 \leq y \implies z \leq x)$   
*<proof>*

**lemma** *floor-sqrt-leI*:  
 $(\bigwedge z. z^2 \leq y \implies z \leq x) \implies \text{floor-sqrt } y \leq x$   
*<proof>*

**lemma** *floor-sqrt-Suc*:  
 $\text{floor-sqrt } (\text{Suc } n) = (\text{if } \exists m. \text{Suc } n = m^2 \text{ then } \text{Suc } (\text{floor-sqrt } n) \text{ else } \text{floor-sqrt } n)$   
*<proof>*

**end**

## 27 Pi and Function Sets

**theory** *FuncSet*  
**imports** *Main*  
**abbrevs**  $PiE = Pi_E$   
**and**  $PIE = \Pi_E$   
**begin**

**definition** *Pi* ::  $'a \text{ set} \Rightarrow ('a \Rightarrow 'b \text{ set}) \Rightarrow ('a \Rightarrow 'b) \text{ set}$   
**where**  $Pi A B = \{f. \forall x. x \in A \longrightarrow f x \in B\}$



**definition** *extensional* :: 'a set  $\Rightarrow$  ('a  $\Rightarrow$  'b) set  
**where** *extensional* A = {f.  $\forall x. x \notin A \longrightarrow f x = \text{undefined}$ }

**definition** *restrict* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a set  $\Rightarrow$  'a  $\Rightarrow$  'b  
**where** *restrict* f A = ( $\lambda x. \text{if } x \in A \text{ then } f x \text{ else undefined}$ )

**abbreviation** *funcset* :: 'a set  $\Rightarrow$  'b set  $\Rightarrow$  ('a  $\Rightarrow$  'b) set  
**where** *funcset* A B  $\equiv$  Pi A ( $\lambda \cdot$ . B)

**open-bundle** *funcset-syntax*

**begin**

**notation** *funcset* (**infixr**  $\langle \rightarrow \rangle$  60)

**end**

**syntax**

-Pi :: pptrn  $\Rightarrow$  'a set  $\Rightarrow$  'b set  $\Rightarrow$  ('a  $\Rightarrow$  'b) set  
 ( $\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \Pi \in \rangle \Pi \text{ -} \in \cdot / \text{ -} \rangle \rangle$  10)  
 -lam :: pptrn  $\Rightarrow$  'a set  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a  $\Rightarrow$  'b)  
 ( $\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \lambda \in \rangle \lambda \text{ -} \in \cdot / \text{ -} \rangle \rangle$  [0, 0, 3] 3)

**syntax-consts**

-Pi  $\equiv$  Pi **and**  
 -lam  $\equiv$  restrict

**translations**

$\Pi x \in A. B \equiv \text{CONST Pi A } (\lambda x. B)$   
 $\lambda x \in A. f \equiv \text{CONST restrict } (\lambda x. f) A$

**definition** *compose* :: 'a set  $\Rightarrow$  ('b  $\Rightarrow$  'c)  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a  $\Rightarrow$  'c)  
**where** *compose* A g f = ( $\lambda x \in A. g (f x)$ )

## 27.1 Basic Properties of Pi

**lemma** *Pi-I[intro!]*: ( $\bigwedge x. x \in A \Longrightarrow f x \in B x$ )  $\Longrightarrow$   $f \in \text{Pi } A B$   
 $\langle \text{proof} \rangle$

**lemma** *Pi-I[simp]*: ( $\bigwedge x. x \in A \longrightarrow f x \in B x$ )  $\Longrightarrow$   $f \in \text{Pi } A B$   
 $\langle \text{proof} \rangle$

**lemma** *funcsetI*: ( $\bigwedge x. x \in A \Longrightarrow f x \in B$ )  $\Longrightarrow$   $f \in A \rightarrow B$   
 $\langle \text{proof} \rangle$

**lemma** *Pi-mem*:  $f \in \text{Pi } A B \Longrightarrow x \in A \Longrightarrow f x \in B x$   
 $\langle \text{proof} \rangle$

**lemma** *Pi-iff*:  $f \in \text{Pi } I X \longleftrightarrow (\forall i \in I. f i \in X i)$   
 $\langle \text{proof} \rangle$

**lemma** *PiE [elim]*:  $f \in \text{Pi } A B \Longrightarrow (f x \in B x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$   
 $\langle \text{proof} \rangle$

**lemma** *Pi-cong*:  $(\bigwedge w. w \in A \implies f w = g w) \implies f \in \text{Pi } A \ B \longleftrightarrow g \in \text{Pi } A \ B$   
 ⟨proof⟩

**lemma** *funcset-id* [*simp*]:  $(\lambda x. x) \in A \rightarrow A$   
 ⟨proof⟩

**lemma** *funcset-mem*:  $f \in A \rightarrow B \implies x \in A \implies f x \in B$   
 ⟨proof⟩

**lemma** *funcset-image*:  $f \in A \rightarrow B \implies f ' A \subseteq B$   
 ⟨proof⟩

**lemma** *image-subset-iff-funcset*:  $F ' A \subseteq B \longleftrightarrow F \in A \rightarrow B$   
 ⟨proof⟩

**lemma** *funcset-to-empty-iff*:  $A \rightarrow \{\} = (\text{if } A = \{\} \text{ then } \text{UNIV} \text{ else } \{\})$   
 ⟨proof⟩

**lemma** *Pi-eq-empty*[*simp*]:  $(\prod x \in A. B x) = \{\} \longleftrightarrow (\exists x \in A. B x = \{\})$   
 ⟨proof⟩

**lemma** *Pi-empty* [*simp*]:  $\text{Pi } \{\} \ B = \text{UNIV}$   
 ⟨proof⟩

**lemma** *Pi-Int*:  $\text{Pi } I \ E \cap \text{Pi } I \ F = (\prod i \in I. E i \cap F i)$   
 ⟨proof⟩

**lemma** *Pi-UN*:  
 fixes  $A :: \text{nat} \Rightarrow 'a \text{ set}$   
 assumes *finite*  $I$   
 and *mono*:  $\bigwedge i \ n \ m. i \in I \implies n \leq m \implies A \ n \ i \subseteq A \ m \ i$   
 shows  $(\bigcup n. \text{Pi } I \ (A \ n)) = (\prod i \in I. \bigcup n. A \ n \ i)$   
 ⟨proof⟩

**lemma** *Pi-UNIV* [*simp*]:  $A \rightarrow \text{UNIV} = \text{UNIV}$   
 ⟨proof⟩

Covariance of Pi-sets in their second argument

**lemma** *Pi-mono*:  $(\bigwedge x. x \in A \implies B x \subseteq C x) \implies \text{Pi } A \ B \subseteq \text{Pi } A \ C$   
 ⟨proof⟩

Contravariance of Pi-sets in their first argument

**lemma** *Pi-anti-mono*:  $A' \subseteq A \implies \text{Pi } A \ B \subseteq \text{Pi } A' \ B$   
 ⟨proof⟩

**lemma** *prod-final*:  
 assumes  $1: \text{fst} \circ f \in \text{Pi } A \ B$   
 and  $2: \text{snd} \circ f \in \text{Pi } A \ C$

**shows**  $f \in (\Pi z \in A. B z \times C z)$   
 ⟨proof⟩

**lemma** *Pi-split-domain[simp]*:  $x \in \text{Pi } (I \cup J) X \longleftrightarrow x \in \text{Pi } I X \wedge x \in \text{Pi } J X$   
 ⟨proof⟩

**lemma** *Pi-split-insert-domain[simp]*:  $x \in \text{Pi } (\text{insert } i I) X \longleftrightarrow x \in \text{Pi } I X \wedge x i \in X i$   
 ⟨proof⟩

**lemma** *Pi-cancel-fupd-range[simp]*:  $i \notin I \implies x \in \text{Pi } I (B(i := b)) \longleftrightarrow x \in \text{Pi } I B$   
 ⟨proof⟩

**lemma** *Pi-cancel-fupd[simp]*:  $i \notin I \implies x(i := a) \in \text{Pi } I B \longleftrightarrow x \in \text{Pi } I B$   
 ⟨proof⟩

**lemma** *Pi-fupd-iff*:  $i \in I \implies f \in \text{Pi } I (B(i := A)) \longleftrightarrow f \in \text{Pi } (I - \{i\}) B \wedge f i \in A$   
 ⟨proof⟩

**lemma** *fst-Pi*:  $\text{fst} \in A \times B \rightarrow A$  **and** *snd-Pi*:  $\text{snd} \in A \times B \rightarrow B$   
 ⟨proof⟩

## 27.2 Composition With a Restricted Domain: *compose*

**lemma** *funcset-compose*:  $f \in A \rightarrow B \implies g \in B \rightarrow C \implies \text{compose } A g f \in A \rightarrow C$   
 ⟨proof⟩

**lemma** *compose-assoc*:  
**assumes**  $f \in A \rightarrow B$   
**shows**  $\text{compose } A h (\text{compose } A g f) = \text{compose } A (\text{compose } B h g) f$   
 ⟨proof⟩

**lemma** *compose-eq*:  $x \in A \implies \text{compose } A g f x = g (f x)$   
 ⟨proof⟩

**lemma** *surj-compose*:  $f \text{ ' } A = B \implies g \text{ ' } B = C \implies \text{compose } A g f \text{ ' } A = C$   
 ⟨proof⟩

## 27.3 Bounded Abstraction: *restrict*

**lemma** *restrict-cong*:  $I = J \implies (\bigwedge i. i \in J \implies f i = g i) \implies \text{restrict } f I = \text{restrict } g J$   
 ⟨proof⟩

**lemma** *restrictI[intro!]*:  $(\bigwedge x. x \in A \implies f x \in B x) \implies (\lambda x \in A. f x) \in \text{Pi } A B$   
 ⟨proof⟩

**lemma** *restrict-apply[simp]*:  $(\lambda y \in A. f y) x = (\text{if } x \in A \text{ then } f x \text{ else undefined})$   
 ⟨proof⟩

**lemma** *restrict-apply'*:  $x \in A \implies (\lambda y \in A. f y) x = f x$   
 ⟨proof⟩

**lemma** *restrict-ext*:  $(\bigwedge x. x \in A \implies f x = g x) \implies (\lambda x \in A. f x) = (\lambda x \in A. g x)$   
 ⟨proof⟩

**lemma** *restrict-UNIV*:  $\text{restrict } f \text{ UNIV} = f$   
 ⟨proof⟩

**lemma** *inj-on-restrict-eq [simp]*:  $\text{inj-on } (\text{restrict } f A) A \longleftrightarrow \text{inj-on } f A$   
 ⟨proof⟩

**lemma** *inj-on-restrict-iff*:  $A \subseteq B \implies \text{inj-on } (\text{restrict } f B) A \longleftrightarrow \text{inj-on } f A$   
 ⟨proof⟩

**lemma** *Id-compose*:  $f \in A \rightarrow B \implies f \in \text{extensional } A \implies \text{compose } A (\lambda y \in B. y)$   
 $f = f$   
 ⟨proof⟩

**lemma** *compose-Id*:  $g \in A \rightarrow B \implies g \in \text{extensional } A \implies \text{compose } A g (\lambda x \in A. x) = g$   
 ⟨proof⟩

**lemma** *image-restrict-eq [simp]*:  $(\text{restrict } f A) \text{ ` } A = f \text{ ` } A$   
 ⟨proof⟩

**lemma** *restrict-restrict[simp]*:  $\text{restrict } (\text{restrict } f A) B = \text{restrict } f (A \cap B)$   
 ⟨proof⟩

**lemma** *restrict-fupd[simp]*:  $i \notin I \implies \text{restrict } (f (i := x)) I = \text{restrict } f I$   
 ⟨proof⟩

**lemma** *restrict-upd[simp]*:  $i \notin I \implies (\text{restrict } f I)(i := y) = \text{restrict } (f(i := y))$   
 (insert  $i$   $I$ )  
 ⟨proof⟩

**lemma** *restrict-Pi-cancel*:  $\text{restrict } x I \in \text{Pi } I A \longleftrightarrow x \in \text{Pi } I A$   
 ⟨proof⟩

**lemma** *sum-restrict' [simp]*:  $\text{sum}' (\lambda i \in I. g i) I = \text{sum}' (\lambda i. g i) I$   
 ⟨proof⟩

**lemma** *prod-restrict' [simp]*:  $\text{prod}' (\lambda i \in I. g i) I = \text{prod}' (\lambda i. g i) I$   
 ⟨proof⟩

## 27.4 Bijections Between Sets

The definition of *bij-betw* is in *Fun.thy*, but most of the theorems belong here, or need at least *Hilbert-Choice*.

**lemma** *bij-betwI*:

**assumes**  $f \in A \rightarrow B$   
**and**  $g \in B \rightarrow A$   
**and**  $g \cdot f: \bigwedge x. x \in A \implies g (f x) = x$   
**and**  $f \cdot g: \bigwedge y. y \in B \implies f (g y) = y$   
**shows** *bij-betw*  $f A B$   
 $\langle$ *proof* $\rangle$

**lemma** *bij-betw-imp-funcset*: *bij-betw*  $f A B \implies f \in A \rightarrow B$

$\langle$ *proof* $\rangle$

**lemma** *inj-on-compose*: *bij-betw*  $f A B \implies$  *inj-on*  $g B \implies$  *inj-on*  $(\text{compose } A g f)$

$\langle$ *proof* $\rangle$

**lemma** *bij-betw-compose*: *bij-betw*  $f A B \implies$  *bij-betw*  $g B C \implies$  *bij-betw*  $(\text{compose } A g f) A C$

$\langle$ *proof* $\rangle$

**lemma** *bij-betw-restrict-eq* [*simp*]: *bij-betw*  $(\text{restrict } f A) A B =$  *bij-betw*  $f A B$

$\langle$ *proof* $\rangle$

## 27.5 Extensionality

**lemma** *extensional-empty*[*simp*]: *extensional*  $\{\}$  =  $\{\lambda x. \text{undefined}\}$

$\langle$ *proof* $\rangle$

**lemma** *extensional-arb*:  $f \in \text{extensional } A \implies x \notin A \implies f x = \text{undefined}$

$\langle$ *proof* $\rangle$

**lemma** *restrict-extensional* [*simp*]: *restrict*  $f A \in \text{extensional } A$

$\langle$ *proof* $\rangle$

**lemma** *compose-extensional* [*simp*]: *compose*  $A f g \in \text{extensional } A$

$\langle$ *proof* $\rangle$

**lemma** *extensionalityI*:

**assumes**  $f \in \text{extensional } A$   
**and**  $g \in \text{extensional } A$   
**and**  $\bigwedge x. x \in A \implies f x = g x$   
**shows**  $f = g$   
 $\langle$ *proof* $\rangle$

**lemma** *extensional-restrict*:  $f \in \text{extensional } A \implies$  *restrict*  $f A = f$

$\langle$ *proof* $\rangle$

**lemma** *extensional-subset*:  $f \in \text{extensional } A \implies A \subseteq B \implies f \in \text{extensional } B$   
 ⟨proof⟩

**lemma** *inv-into-funcset*:  $f \text{ ‘ } A = B \implies (\lambda x \in B. \text{inv-into } A \ f \ x) \in B \rightarrow A$   
 ⟨proof⟩

**lemma** *compose-inv-into-id*:  $\text{bij-betw } f \ A \ B \implies \text{compose } A \ (\lambda y \in B. \text{inv-into } A \ f \ y)$   
 $f = (\lambda x \in A. x)$   
 ⟨proof⟩

**lemma** *compose-id-inv-into*:  $f \text{ ‘ } A = B \implies \text{compose } B \ f \ (\lambda y \in B. \text{inv-into } A \ f \ y)$   
 $= (\lambda x \in B. x)$   
 ⟨proof⟩

**lemma** *extensional-insert*[*intro, simp*]:  
**assumes**  $a \in \text{extensional } (\text{insert } i \ I)$   
**shows**  $a(i := b) \in \text{extensional } (\text{insert } i \ I)$   
 ⟨proof⟩

**lemma** *extensional-Int*[*simp*]:  $\text{extensional } I \cap \text{extensional } I' = \text{extensional } (I \cap I')$   
 ⟨proof⟩

**lemma** *extensional-UNIV*[*simp*]:  $\text{extensional } UNIV = UNIV$   
 ⟨proof⟩

**lemma** *restrict-extensional-sub*[*intro*]:  $A \subseteq B \implies \text{restrict } f \ A \in \text{extensional } B$   
 ⟨proof⟩

**lemma** *extensional-insert-undefined*[*intro, simp*]:  
 $a \in \text{extensional } (\text{insert } i \ I) \implies a(i := \text{undefined}) \in \text{extensional } I$   
 ⟨proof⟩

**lemma** *extensional-insert-cancel*[*intro, simp*]:  
 $a \in \text{extensional } I \implies a \in \text{extensional } (\text{insert } i \ I)$   
 ⟨proof⟩

## 27.6 Cardinality

**lemma** *card-inj*:  $f \in A \rightarrow B \implies \text{inj-on } f \ A \implies \text{finite } B \implies \text{card } A \leq \text{card } B$   
 ⟨proof⟩

**lemma** *card-bij*:  
**assumes**  $f \in A \rightarrow B \text{ inj-on } f \ A$   
**and**  $g \in B \rightarrow A \text{ inj-on } g \ B$   
**and**  $\text{finite } A \ \text{finite } B$   
**shows**  $\text{card } A = \text{card } B$   
 ⟨proof⟩

## 27.7 Extensional Function Spaces

**definition**  $PiE :: 'a \text{ set} \Rightarrow ('a \Rightarrow 'b \text{ set}) \Rightarrow ('a \Rightarrow 'b) \text{ set}$   
**where**  $PiE \ S \ T = Pi \ S \ T \cap \text{extensional } S$

**abbreviation**  $Pi_E \ A \ B \equiv PiE \ A \ B$

**syntax**

$-PiE :: p\text{trn} \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow ('a \Rightarrow 'b) \text{ set}$   
 $(\langle \langle \text{indent}=3 \ \text{notation}=\langle \text{binder } \Pi_E \rangle \rangle \Pi_E \ -\in- / - \rangle \ 10)$

**syntax-consts**

$-PiE \equiv Pi_E$

**translations**

$\Pi_E \ x \in A. B \equiv CONST \ Pi_E \ A \ (\lambda x. B)$

**abbreviation**  $\text{extensional-funcset} :: 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow ('a \Rightarrow 'b) \text{ set}$  (**infixr**  $\langle \rightarrow_E \rangle$  60)

**where**  $A \rightarrow_E B \equiv (\Pi_E \ i \in A. B)$

**lemma**  $\text{extensional-funcset-def}$ :  $\text{extensional-funcset } S \ T = (S \rightarrow T) \cap \text{extensional } S$

$\langle \text{proof} \rangle$

**lemma**  $PiE\text{-empty-domain}[simp]$ :  $Pi_E \ \{\} \ T = \{\lambda x. \text{undefined}\}$

$\langle \text{proof} \rangle$

**lemma**  $PiE\text{-UNIV-domain}$ :  $Pi_E \ UNIV \ T = Pi \ UNIV \ T$

$\langle \text{proof} \rangle$

**lemma**  $PiE\text{-empty-range}[simp]$ :  $i \in I \Longrightarrow F \ i = \{\} \Longrightarrow (\Pi_E \ i \in I. F \ i) = \{\}$

$\langle \text{proof} \rangle$

**lemma**  $PiE\text{-eq-empty-iff}$ :  $Pi_E \ I \ F = \{\} \longleftrightarrow (\exists i \in I. F \ i = \{\})$

$\langle \text{proof} \rangle$

**lemma**  $PiE\text{-arb}$ :  $f \in Pi_E \ S \ T \Longrightarrow x \notin S \Longrightarrow f \ x = \text{undefined}$

$\langle \text{proof} \rangle$

**lemma**  $PiE\text{-mem}$ :  $f \in Pi_E \ S \ T \Longrightarrow x \in S \Longrightarrow f \ x \in T \ x$

$\langle \text{proof} \rangle$

**lemma**  $PiE\text{-fun-upd}$ :  $y \in T \ x \Longrightarrow f \in Pi_E \ S \ T \Longrightarrow f(x := y) \in Pi_E \ (\text{insert } x \ S)$

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

**lemma**  $\text{fun-upd-in-PiE}$ :  $x \notin S \Longrightarrow f \in Pi_E \ (\text{insert } x \ S) \ T \Longrightarrow f(x := \text{undefined}) \in Pi_E \ S \ T$

$\langle \text{proof} \rangle$

**lemma**  $PiE\text{-insert-eq}$ :  $Pi_E \ (\text{insert } x \ S) \ T = (\lambda(y, g). g(x := y)) \ ` \ (T \ x \times Pi_E \ S$

$T$ )  
 $\langle proof \rangle$

**lemma** *PiE-Int*:  $Pi_E I A \cap Pi_E I B = Pi_E I (\lambda x. A x \cap B x)$   
 $\langle proof \rangle$

**lemma** *PiE-cong*:  $(\bigwedge i. i \in I \implies A i = B i) \implies Pi_E I A = Pi_E I B$   
 $\langle proof \rangle$

**lemma** *PiE-E [elim]*:  
**assumes**  $f \in Pi_E A B$   
**obtains**  $x \in A$  **and**  $f x \in B x$   
 |  $x \notin A$  **and**  $f x = undefined$   
 $\langle proof \rangle$

**lemma** *PiE-I[intro!]*:  
 $(\bigwedge x. x \in A \implies f x \in B x) \implies (\bigwedge x. x \notin A \implies f x = undefined) \implies f \in Pi_E A B$   
 $\langle proof \rangle$

**lemma** *PiE-mono*:  $(\bigwedge x. x \in A \implies B x \subseteq C x) \implies Pi_E A B \subseteq Pi_E A C$   
 $\langle proof \rangle$

**lemma** *PiE-iff*:  $f \in Pi_E I X \longleftrightarrow (\forall i \in I. f i \in X i) \wedge f \in \text{extensional } I$   
 $\langle proof \rangle$

**lemma** *restrict-PiE-iff*:  $\text{restrict } f I \in Pi_E I X \longleftrightarrow (\forall i \in I. f i \in X i)$   
 $\langle proof \rangle$

**lemma** *ext-funcset-to-sing-iff [simp]*:  $A \rightarrow_E \{a\} = \{\lambda x \in A. a\}$   
 $\langle proof \rangle$

**lemma** *PiE-restrict[simp]*:  $f \in Pi_E A B \implies \text{restrict } f A = f$   
 $\langle proof \rangle$

**lemma** *restrict-PiE[simp]*:  $\text{restrict } f I \in Pi_E I S \longleftrightarrow f \in Pi I S$   
 $\langle proof \rangle$

**lemma** *PiE-eq-subset*:  
**assumes**  $ne: \bigwedge i. i \in I \implies F i \neq \{\}$   $\wedge \bigwedge i. i \in I \implies F' i \neq \{\}$   
**and**  $eq: Pi_E I F = Pi_E I F'$   
**and**  $i \in I$   
**shows**  $F i \subseteq F' i$   
 $\langle proof \rangle$

**lemma** *PiE-eq-iff-not-empty*:  
**assumes**  $ne: \bigwedge i. i \in I \implies F i \neq \{\}$   $\wedge \bigwedge i. i \in I \implies F' i \neq \{\}$   
**shows**  $Pi_E I F = Pi_E I F' \longleftrightarrow (\forall i \in I. F i = F' i)$   
 $\langle proof \rangle$



**lemma** *PiE-eq-iff*:  $PiE\ I\ F = PiE\ I\ F' \longleftrightarrow (\forall i \in I. F\ i = F'\ i) \vee ((\exists i \in I. F\ i = \{\}) \wedge (\exists i \in I. F'\ i = \{\}))$   
 ⟨proof⟩

**lemma** *extensional-funcset-fun-upd-restricts-rangeI*:  
 $\forall y \in S. f\ x \neq f\ y \implies f \in (insert\ x\ S) \rightarrow_E\ T \implies f(x := undefined) \in S \rightarrow_E (T - \{f\ x\})$   
 ⟨proof⟩

**lemma** *extensional-funcset-fun-upd-extends-rangeI*:  
**assumes**  $a \in T\ f \in S \rightarrow_E (T - \{a\})$   
**shows**  $f(x := a) \in insert\ x\ S \rightarrow_E\ T$   
 ⟨proof⟩

**lemma** *subset-PiE*:  
 $PiE\ I\ S \subseteq PiE\ I\ T \longleftrightarrow PiE\ I\ S = \{\} \vee (\forall i \in I. S\ i \subseteq T\ i)$  (**is** ?lhs  $\longleftrightarrow$  -  $\vee$  ?rhs)  
 ⟨proof⟩

**lemma** *PiE-eq*:  $PiE\ I\ S = PiE\ I\ T \longleftrightarrow PiE\ I\ S = \{\} \wedge PiE\ I\ T = \{\} \vee (\forall i \in I. S\ i = T\ i)$   
 ⟨proof⟩

**lemma** *PiE-UNIV [simp]*:  $PiE\ UNIV\ (\lambda i. UNIV) = UNIV$   
 ⟨proof⟩

**lemma** *image-projection-PiE*:  
 $(\lambda f. f\ i) \cdot (PiE\ I\ S) = (if\ PiE\ I\ S = \{\} then\ \{\} else\ if\ i \in I then\ S\ i else\ \{undefined\})$   
 ⟨proof⟩

**lemma** *PiE-singleton*:  
**assumes**  $f \in extensional\ A$   
**shows**  $PiE\ A\ (\lambda x. \{f\ x\}) = \{f\}$   
 ⟨proof⟩

**lemma** *PiE-eq-singleton*:  $(\Pi_E\ i \in I. S\ i) = \{\lambda i \in I. f\ i\} \longleftrightarrow (\forall i \in I. S\ i = \{f\ i\})$   
 ⟨proof⟩

**lemma** *PiE-over-singleton-iff*:  $(\Pi_E\ x \in \{a\}. B\ x) = (\bigcup b \in B\ a. \{\lambda x \in \{a\}. b\})$   
 ⟨proof⟩

**lemma** *all-PiE-elements*:  
 $(\forall z \in PiE\ I\ S. \forall i \in I. P\ i\ (z\ i)) \longleftrightarrow PiE\ I\ S = \{\} \vee (\forall i \in I. \forall x \in S\ i. P\ i\ x)$   
 (**is** ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *PiE-ext*:  $\llbracket x \in PiE\ k\ s; y \in PiE\ k\ s; \bigwedge i. i \in k \implies x\ i = y\ i \rrbracket \implies x = y$

*<proof>*

### 27.7.1 Injective Extensional Function Spaces

**lemma** *extensional-funcset-fun-upd-inj-onI*:

**assumes**  $f \in S \rightarrow_E (T - \{a\})$

**and** *inj-on*  $f S$

**shows** *inj-on*  $(f(x := a)) S$

*<proof>*

**lemma** *extensional-funcset-extend-domain-inj-on-eq*:

**assumes**  $x \notin S$

**shows**  $\{f. f \in (\text{insert } x S) \rightarrow_E T \wedge \text{inj-on } f (\text{insert } x S)\} =$

$(\lambda(y, g). g(x:=y)) \text{ ‘ } \{(y, g). y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge \text{inj-on } g S\}$

*<proof>*

**lemma** *extensional-funcset-extend-domain-inj-onI*:

**assumes**  $x \notin S$

**shows** *inj-on*  $(\lambda(y, g). g(x := y)) \{(y, g). y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge \text{inj-on } g S\}$

*<proof>*

### 27.7.2 Misc properties of functions, composition and restriction from HOL Light

**lemma** *function-factors-left-gen*:

$(\forall x y. P x \wedge P y \wedge g x = g y \longrightarrow f x = f y) \longleftrightarrow (\exists h. \forall x. P x \longrightarrow f x = h(g x))$

**(is ?lhs = ?rhs)**

*<proof>*

**lemma** *function-factors-left*:  $(\forall x y. (g x = g y) \longrightarrow (f x = f y)) \longleftrightarrow (\exists h. f = h \circ g)$

*<proof>*

**lemma** *function-factors-right-gen*:  $(\forall x. P x \longrightarrow (\exists y. g y = f x)) \longleftrightarrow (\exists h. \forall x. P x \longrightarrow f x = g(h x))$

*<proof>*

**lemma** *function-factors-right*:  $(\forall x. \exists y. g y = f x) \longleftrightarrow (\exists h. f = g \circ h)$

*<proof>*

**lemma** *restrict-compose-right*:  $\text{restrict } (g \circ \text{restrict } f S) S = \text{restrict } (g \circ f) S$

*<proof>*

**lemma** *restrict-compose-left*:  $f \text{ ‘ } S \subseteq T \implies \text{restrict } (\text{restrict } g T \circ f) S = \text{restrict } (g \circ f) S$

*<proof>*

**27.7.3 Cardinality**

**lemma** *finite-PiE*:  $\text{finite } S \implies (\bigwedge i. i \in S \implies \text{finite } (T i)) \implies \text{finite } (\Pi_E i \in S. T i)$   
 ⟨proof⟩

**lemma** *inj-combinator*:  $x \notin S \implies \text{inj-on } (\lambda(y, g). g(x := y)) (T x \times \text{Pi}_E S T)$   
 ⟨proof⟩

**lemma** *card-PiE*:  $\text{finite } S \implies \text{card } (\Pi_E i \in S. T i) = (\prod_{i \in S}. \text{card } (T i))$   
 ⟨proof⟩

**lemma** *card-funcsetE*:  $\text{finite } A \implies \text{card } (A \rightarrow_E B) = \text{card } B \wedge \text{card } A$   
 ⟨proof⟩

**lemma** *card-inj-on-subset-funcset*:

**assumes** *finB*:  $\text{finite } B$

**and** *finC*:  $\text{finite } C$

**and** *AB*:  $A \subseteq B$

**shows**  $\text{card } \{f \in B \rightarrow_E C. \text{inj-on } f A\} =$

$\text{card } C \wedge (\text{card } B - \text{card } A) * \text{prod } ((-) (\text{card } C)) \{0 ..< \text{card } A\}$

⟨proof⟩

**27.8 The pigeonhole principle**

An alternative formulation of this is that for a function mapping a finite set  $A$  of cardinality  $m$  to a finite set  $B$  of cardinality  $n$ , there exists an element  $y \in B$  that is hit at least  $\lceil \frac{m}{n} \rceil$  times. However, since we do not have real numbers or rounding yet, we state it in the following equivalent form:

**lemma** *pigeonhole-card*:

**assumes**  $f \in A \rightarrow B$  *finite A* *finite B*  $B \neq \{\}$

**shows**  $\exists y \in B. \text{card } (f^{-1} \{y\} \cap A) * \text{card } B \geq \text{card } A$

⟨proof⟩

**end**

**28 Partitions and Disjoint Sets**

**theory** *Disjoint-Sets*

**imports** *FuncSet*

**begin**

**lemma** *mono-imp-UN-eq-last*:  $\text{mono } A \implies (\bigcup_{i \leq n}. A i) = A n$   
 ⟨proof⟩

**28.1 Set of Disjoint Sets**

**abbreviation** *disjoint* :: 'a set set  $\Rightarrow$  bool **where** *disjoint*  $\equiv$  *pairwise disjnt*

**lemma disjoint-def:**  $disjoint\ A \iff (\forall a \in A. \forall b \in A. a \neq b \implies a \cap b = \{\})$   
 ⟨proof⟩

**lemma disjointI:**  
 $(\bigwedge a\ b. a \in A \implies b \in A \implies a \neq b \implies a \cap b = \{\}) \implies disjoint\ A$   
 ⟨proof⟩

**lemma disjointD:**  
 $disjoint\ A \implies a \in A \implies b \in A \implies a \neq b \implies a \cap b = \{\}$   
 ⟨proof⟩

**lemma disjoint-image:**  $inj\text{-on}\ f\ (\bigcup A) \implies disjoint\ A \implies disjoint\ ((\cdot) f\ 'A)$   
 ⟨proof⟩

**lemma assumes disjoint (A ∪ B)**  
**shows disjoint-unionD1:**  $disjoint\ A$  **and disjoint-unionD2:**  $disjoint\ B$   
 ⟨proof⟩

**lemma disjoint-INT:**  
**assumes**  $*$ :  $\bigwedge i. i \in I \implies disjoint\ (F\ i)$   
**shows**  $disjoint\ \{\bigcap i \in I. X\ i \mid X. \forall i \in I. X\ i \in F\ i\}$   
 ⟨proof⟩

**lemma diff-Union-pairwise-disjoint:**  
**assumes**  $pairwise\ disjnt\ \mathcal{A}\ \mathcal{B} \subseteq \mathcal{A}$   
**shows**  $\bigcup \mathcal{A} - \bigcup \mathcal{B} = \bigcup (\mathcal{A} - \mathcal{B})$   
 ⟨proof⟩

**lemma Int-Union-pairwise-disjoint:**  
**assumes**  $pairwise\ disjnt\ (\mathcal{A} \cup \mathcal{B})$   
**shows**  $\bigcup \mathcal{A} \cap \bigcup \mathcal{B} = \bigcup (\mathcal{A} \cap \mathcal{B})$   
 ⟨proof⟩

**lemma psubset-Union-pairwise-disjoint:**  
**assumes**  $\mathcal{B}$ :  $pairwise\ disjnt\ \mathcal{B}$  **and**  $\mathcal{A} \subseteq \mathcal{B} - \{\{\}\}$   
**shows**  $\bigcup \mathcal{A} \subseteq \bigcup \mathcal{B}$   
 ⟨proof⟩

### 28.1.1 Family of Disjoint Sets

**definition disjoint-family-on** ::  $(i \Rightarrow 'a\ set) \Rightarrow 'i\ set \Rightarrow bool$  **where**  
 $disjoint\ family\ on\ A\ S \iff (\forall m \in S. \forall n \in S. m \neq n \implies A\ m \cap A\ n = \{\})$

**abbreviation disjoint-family**  $A \equiv disjoint\ family\ on\ A\ UNIV$

**lemma disjoint-family-elim-disjnt:**  
**assumes**  $infinite\ A$   $finite\ C$   
**and**  $df$ :  $disjoint\ family\ on\ B\ A$   
**obtains**  $x$  **where**  $x \in A$   $disjnt\ C\ (B\ x)$

*<proof>*

**lemma** *disjoint-family-onD:*

*disjoint-family-on*  $A\ I \implies i \in I \implies j \in I \implies i \neq j \implies A\ i \cap A\ j = \{\}$

*<proof>*

**lemma** *disjoint-family-subset:* *disjoint-family*  $A \implies (\bigwedge x. B\ x \subseteq A\ x) \implies$  *disjoint-family*  $B$

*<proof>*

**lemma** *disjoint-family-on-insert:*

$i \notin I \implies$  *disjoint-family-on*  $A\ (\text{insert } i\ I) \iff A\ i \cap (\bigcup_{i \in I}. A\ i) = \{\} \wedge$  *disjoint-family-on*  $A\ I$

*<proof>*

**lemma** *disjoint-family-on-bisimulation:*

**assumes** *disjoint-family-on*  $f\ S$

**and**  $\bigwedge n\ m. n \in S \implies m \in S \implies n \neq m \implies f\ n \cap f\ m = \{\} \implies g\ n \cap g\ m = \{\}$

**shows** *disjoint-family-on*  $g\ S$

*<proof>*

**lemma** *disjoint-family-on-mono:*

$A \subseteq B \implies$  *disjoint-family-on*  $f\ B \implies$  *disjoint-family-on*  $f\ A$

*<proof>*

**lemma** *disjoint-family-Suc:*

$(\bigwedge n. A\ n \subseteq A\ (\text{Suc } n)) \implies$  *disjoint-family*  $(\lambda i. A\ (\text{Suc } i) - A\ i)$

*<proof>*

**lemma** *disjoint-family-on-disjoint-image:*

*disjoint-family-on*  $A\ I \implies$  *disjoint*  $(A\ \text{' } I)$

*<proof>*

**lemma** *disjoint-family-on-vimageI:* *disjoint-family-on*  $F\ I \implies$  *disjoint-family-on*  $(\lambda i. f\ \text{' } F\ i)\ I$

*<proof>*

**lemma** *disjoint-image-disjoint-family-on:*

**assumes**  $d: \text{disjoint } (A\ \text{' } I)$  **and**  $i: \text{inj-on } A\ I$

**shows** *disjoint-family-on*  $A\ I$

*<proof>*

**lemma** *disjoint-family-on-iff-disjoint-image:*

**assumes**  $\bigwedge i. i \in I \implies A\ i \neq \{\}$

**shows** *disjoint-family-on*  $A\ I \iff \text{disjoint } (A\ \text{' } I) \wedge \text{inj-on } A\ I$

*<proof>*

**lemma** *card-UN-disjoint':*

**assumes** *disjoint-family-on*  $A I \wedge i. i \in I \implies \text{finite } (A i) \text{ finite } I$   
**shows**  $\text{card } (\bigcup_{i \in I}. A i) = (\sum_{i \in I}. \text{card } (A i))$   
 ⟨*proof*⟩

**lemma** *disjoint-UN*:

**assumes**  $F: \bigwedge i. i \in I \implies \text{disjoint } (F i)$  **and**  $*$ : *disjoint-family-on*  $(\lambda i. \bigcup (F i)) I$   
**shows**  $\text{disjoint } (\bigcup_{i \in I}. F i)$   
 ⟨*proof*⟩

**lemma** *distinct-list-bind*:

**assumes**  $\text{distinct } xs \wedge x. x \in \text{set } xs \implies \text{distinct } (f x)$   
*disjoint-family-on*  $(\text{set } \circ f) (\text{set } xs)$   
**shows**  $\text{distinct } (\text{List.bind } xs f)$   
 ⟨*proof*⟩

**lemma** *bij-betw-UNION-disjoint*:

**assumes** *disj*: *disjoint-family-on*  $A' I$   
**assumes** *bij*:  $\bigwedge i. i \in I \implies \text{bij-betw } f (A i) (A' i)$   
**shows**  $\text{bij-betw } f (\bigcup_{i \in I}. A i) (\bigcup_{i \in I}. A' i)$   
 ⟨*proof*⟩

**lemma** *disjoint-union*:  $\text{disjoint } C \implies \text{disjoint } B \implies \bigcup C \cap \bigcup B = \{\} \implies \text{disjoint } (C \cup B)$   
 ⟨*proof*⟩

Sum/product of the union of a finite disjoint family

**context** *comm-monoid-set*

**begin**

**lemma** *UNION-disjoint-family*:

**assumes** *finite*  $I$  **and**  $\forall i \in I. \text{finite } (A i)$   
**and** *disjoint-family-on*  $A I$   
**shows**  $F g (\bigcup (A ' I)) = F (\lambda x. F g (A x)) I$   
 ⟨*proof*⟩

**lemma** *Union-disjoint-sets*:

**assumes**  $\forall A \in C. \text{finite } A$  **and** *disjoint*  $C$   
**shows**  $F g (\bigcup C) = (F \circ F) g C$   
 ⟨*proof*⟩

**end**

The union of an infinite disjoint family of non-empty sets is infinite.

**lemma** *infinite-disjoint-family-imp-infinite-UNION*:

**assumes**  $\neg \text{finite } A \wedge x. x \in A \implies f x \neq \{\}$  *disjoint-family-on*  $f A$   
**shows**  $\neg \text{finite } (\bigcup (f ' A))$   
 ⟨*proof*⟩

## 28.2 Construct Disjoint Sequences

**definition** *disjointed* :: (nat  $\Rightarrow$  'a set)  $\Rightarrow$  nat  $\Rightarrow$  'a set **where**  
*disjointed* A n = A n - ( $\bigcup_{i \in \{0..<n\}}$ . A i)

**lemma** *finite-UN-disjointed-eq*: ( $\bigcup_{i \in \{0..<n\}}$ . disjointed A i) = ( $\bigcup_{i \in \{0..<n\}}$ . A i)  
 <proof>

**lemma** *UN-disjointed-eq*: ( $\bigcup i$ . disjointed A i) = ( $\bigcup i$ . A i)  
 <proof>

**lemma** *less-disjoint-disjointed*:  $m < n \implies$  disjointed A m  $\cap$  disjointed A n = {}  
 <proof>

**lemma** *disjoint-family-disjointed*: disjoint-family (disjointed A)  
 <proof>

**lemma** *disjointed-subset*: disjointed A n  $\subseteq$  A n  
 <proof>

**lemma** *disjointed-0[simp]*: disjointed A 0 = A 0  
 <proof>

**lemma** *disjointed-mono*: mono A  $\implies$  disjointed A (Suc n) = A (Suc n) - A n  
 <proof>

## 28.3 Partitions

Partitions  $P$  of a set  $A$ . We explicitly disallow empty sets.

**definition** *partition-on* :: 'a set  $\Rightarrow$  'a set set  $\Rightarrow$  bool  
**where**  
*partition-on* A P  $\longleftrightarrow$   $\bigcup P = A \wedge$  disjoint P  $\wedge$  {}  $\notin$  P

**lemma** *partition-onI*:  
 $\bigcup P = A \implies (\bigwedge p q. p \in P \implies q \in P \implies p \neq q \implies$  disjnt p q)  $\implies$  {}  $\notin$  P  
 $\implies$  *partition-on* A P  
 <proof>

**lemma** *partition-onD1*: *partition-on* A P  $\implies$  A =  $\bigcup P$   
 <proof>

**lemma** *partition-onD2*: *partition-on* A P  $\implies$  disjoint P  
 <proof>

**lemma** *partition-onD3*: *partition-on* A P  $\implies$  {}  $\notin$  P  
 <proof>

## 28.4 Constructions of partitions

**lemma** *partition-on-empty*: *partition-on*  $\{\}$   $P \longleftrightarrow P = \{\}$   
 ⟨*proof*⟩

**lemma** *partition-on-space*:  $A \neq \{\} \implies \text{partition-on } A \{A\}$   
 ⟨*proof*⟩

**lemma** *partition-on-singletons*: *partition-on*  $A ((\lambda x. \{x\}) \text{ ‘ } A)$   
 ⟨*proof*⟩

**lemma** *partition-on-transform*:

**assumes**  $P$ : *partition-on*  $A P$

**assumes**  $F$ -UN:  $\bigcup (F \text{ ‘ } P) = F (\bigcup P)$  **and**  $F$ -disjnt:  $\bigwedge p q. p \in P \implies q \in P$   
 $\implies \text{disjnt } p q \implies \text{disjnt } (F p) (F q)$

**shows** *partition-on*  $(F A) (F \text{ ‘ } P - \{\{\}\})$

⟨*proof*⟩

**lemma** *partition-on-restrict*: *partition-on*  $A P \implies \text{partition-on } (B \cap A) ((\cap) B \text{ ‘ } P - \{\{\}\})$   
 ⟨*proof*⟩

**lemma** *partition-on-vimage*: *partition-on*  $A P \implies \text{partition-on } (f \text{ ‘ } A) ((\text{‘}) f \text{ ‘ } P - \{\{\}\})$   
 ⟨*proof*⟩

**lemma** *partition-on-inj-image*:

**assumes**  $P$ : *partition-on*  $A P$  **and**  $f$ : *inj-on*  $f A$

**shows** *partition-on*  $(f \text{ ‘ } A) ((\text{‘}) f \text{ ‘ } P - \{\{\}\})$

⟨*proof*⟩

**lemma** *partition-on-insert*:

**assumes**  $\text{disjnt } p (\bigcup P)$

**shows** *partition-on*  $A (\text{insert } p P) \longleftrightarrow \text{partition-on } (A - p) P \wedge p \subseteq A \wedge p \neq \{\}$

⟨*proof*⟩

## 28.5 Finiteness of partitions

**lemma** *finitely-many-partition-on*:

**assumes** *finite*  $A$

**shows** *finite*  $\{P. \text{partition-on } A P\}$

⟨*proof*⟩

**lemma** *finite-elements*: *finite*  $A \implies \text{partition-on } A P \implies \text{finite } P$   
 ⟨*proof*⟩

**lemma** *product-partition*:

**assumes** *partition-on*  $A P$  **and**  $\bigwedge p. p \in P \implies \text{finite } p$

**shows**  $\text{card } A = (\sum p \in P. \text{card } p)$

⟨*proof*⟩



## 28.6 Equivalence of partitions and equivalence classes

**lemma** *partition-on-quotient*:

**assumes**  $r$ : *equiv*  $A$   $r$

**shows** *partition-on*  $A$  ( $A // r$ )

*<proof>*

**lemma** *equiv-partition-on*:

**assumes**  $P$ : *partition-on*  $A$   $P$

**shows** *equiv*  $A$   $\{(x, y). \exists p \in P. x \in p \wedge y \in p\}$

*<proof>*

**lemma** *partition-on-eq-quotient*:

**assumes**  $P$ : *partition-on*  $A$   $P$

**shows**  $A // \{(x, y). \exists p \in P. x \in p \wedge y \in p\} = P$

*<proof>*

**lemma** *partition-on-alt*: *partition-on*  $A$   $P \longleftrightarrow (\exists r. \text{equiv } A \ r \wedge P = A // r)$

*<proof>*

## 28.7 Refinement of partitions

**definition** *refines* :: 'a set  $\Rightarrow$  'a set set  $\Rightarrow$  'a set set  $\Rightarrow$  bool

**where** *refines*  $A$   $P$   $Q \equiv$

*partition-on*  $A$   $P \wedge \text{partition-on } A$   $Q \wedge (\forall X \in P. \exists Y \in Q. X \subseteq Y)$

**lemma** *refines-refl*: *partition-on*  $A$   $P \Longrightarrow \text{refines } A$   $P$   $P$

*<proof>*

**lemma** *refines-asym1*:

**assumes** *refines*  $A$   $P$   $Q$  *refines*  $A$   $Q$   $P$

**shows**  $P \subseteq Q$

*<proof>*

**lemma** *refines-asym*:  $\llbracket \text{refines } A$   $P$   $Q; \text{refines } A$   $Q$   $P \rrbracket \Longrightarrow P=Q$

*<proof>*

**lemma** *refines-trans*:  $\llbracket \text{refines } A$   $P$   $Q; \text{refines } A$   $Q$   $R \rrbracket \Longrightarrow \text{refines } A$   $P$   $R$

*<proof>*

**lemma** *refines-obtains-subset*:

**assumes** *refines*  $A$   $P$   $Q$   $q \in Q$

**shows** *partition-on*  $q$   $\{p \in P. p \subseteq q\}$

*<proof>*

## 28.8 The coarsest common refinement of a set of partitions

**definition** *common-refinement* :: 'a set set set  $\Rightarrow$  'a set set

**where** *common-refinement*  $\mathcal{P} \equiv (\bigcup f \in (\Pi_E P \in \mathcal{P}. P). \{\bigcap (f \text{ ' } \mathcal{P})\}) - \{\{\}\}$

With non-extensional function space

**lemma** *common-refinement*:  $\text{common-refinement } \mathcal{P} = (\bigcup f \in (\prod P \in \mathcal{P}. P). \{\bigcap (f \text{ ‘ } \mathcal{P})\}) - \{\{\}\}$   
*(is ?lhs = ?rhs)*  
 ⟨proof⟩

**lemma** *common-refinement-exists*:  $[[X \in \text{common-refinement } \mathcal{P}; P \in \mathcal{P}] \implies \exists R \in \mathcal{P}. X \subseteq R$   
 ⟨proof⟩

**lemma** *Union-common-refinement*:  $\bigcup (\text{common-refinement } \mathcal{P}) = (\bigcap P \in \mathcal{P}. \bigcup P)$   
 ⟨proof⟩

**lemma** *partition-on-common-refinement*:  
*assumes*  $A: \bigwedge P. P \in \mathcal{P} \implies \text{partition-on } A P$  **and**  $\mathcal{P} \neq \{\}$   
*shows*  $\text{partition-on } A (\text{common-refinement } \mathcal{P})$   
 ⟨proof⟩

**lemma** *refines-common-refinement*:  
*assumes*  $\bigwedge P. P \in \mathcal{P} \implies \text{partition-on } A P$   $P \in \mathcal{P}$   
*shows*  $\text{refines } A (\text{common-refinement } \mathcal{P}) P$   
 ⟨proof⟩

The common refinement is itself refined by any other

**lemma** *common-refinement-coarsest*:  
*assumes*  $\bigwedge P. P \in \mathcal{P} \implies \text{partition-on } A P$   $\text{partition-on } A R$   $\bigwedge P. P \in \mathcal{P} \implies \text{refines } A R P$   $\mathcal{P} \neq \{\}$   
*shows*  $\text{refines } A R (\text{common-refinement } \mathcal{P})$   
 ⟨proof⟩

**lemma** *finite-common-refinement*:  
*assumes*  $\text{finite } \mathcal{P}$   $\bigwedge P. P \in \mathcal{P} \implies \text{finite } P$   
*shows*  $\text{finite } (\text{common-refinement } \mathcal{P})$   
 ⟨proof⟩

**lemma** *card-common-refinement*:  
*assumes*  $\text{finite } \mathcal{P}$   $\bigwedge P. P \in \mathcal{P} \implies \text{finite } P$   
*shows*  $\text{card } (\text{common-refinement } \mathcal{P}) \leq (\prod P \in \mathcal{P}. \text{card } P)$   
 ⟨proof⟩

end

## 29 Type of finite sets defined as a subtype of sets

```
theory FSet
imports Main Countable
begin
```

## 29.1 Definition of the type

```
typedef 'a fset = {A :: 'a set. finite A} morphisms fset Abs-fset
⟨proof⟩
```

```
setup-lifting type-definition-fset
```

## 29.2 Basic operations and type class instantiations

```
instantiation fset :: (finite) finite
```

```
begin
```

```
instance ⟨proof⟩
```

```
end
```

```
instantiation fset :: (type) {bounded-lattice-bot, distrib-lattice, minus}
```

```
begin
```

```
lift-definition bot-fset :: 'a fset is {} parametric empty-transfer ⟨proof⟩
```

```
lift-definition less-eq-fset :: 'a fset ⇒ 'a fset ⇒ bool is subset-eq parametric
```

```
subset-transfer
```

```
⟨proof⟩
```

```
definition less-fset :: 'a fset ⇒ 'a fset ⇒ bool where xs < ys ≡ xs ≤ ys ∧ xs ≠
```

```
(ys :: 'a fset)
```

```
lemma less-fset-transfer[transfer-rule]:
```

```
includes lifting-syntax
```

```
assumes [transfer-rule]: bi-unique A
```

```
shows ((pcr-fset A) ==> (pcr-fset A) ==> (=)) (C) (<)
```

```
⟨proof⟩
```

```
lift-definition sup-fset :: 'a fset ⇒ 'a fset ⇒ 'a fset is union parametric union-transfer
```

```
⟨proof⟩
```

```
lift-definition inf-fset :: 'a fset ⇒ 'a fset ⇒ 'a fset is inter parametric inter-transfer
```

```
⟨proof⟩
```

```
lift-definition minus-fset :: 'a fset ⇒ 'a fset ⇒ 'a fset is minus parametric
```

```
Diff-transfer
```

```
⟨proof⟩
```

```
instance
```

```
⟨proof⟩
```

```
end
```

```
abbreviation fempty :: 'a fset (⟨{||}⟩) where {||} ≡ bot
```

**abbreviation**  $fsubset\text{-}eq :: 'a\ fset \Rightarrow 'a\ fset \Rightarrow bool$  (**infix**  $\langle | \subseteq | \rangle 50$ ) **where**  $xs\ | \subseteq | ys \equiv xs \leq ys$

**abbreviation**  $fsubset :: 'a\ fset \Rightarrow 'a\ fset \Rightarrow bool$  (**infix**  $\langle | \subset | \rangle 50$ ) **where**  $xs\ | \subset | ys \equiv xs < ys$

**abbreviation**  $funion :: 'a\ fset \Rightarrow 'a\ fset \Rightarrow 'a\ fset$  (**infixl**  $\langle | \cup | \rangle 65$ ) **where**  $xs\ | \cup | ys \equiv sup\ xs\ ys$

**abbreviation**  $finter :: 'a\ fset \Rightarrow 'a\ fset \Rightarrow 'a\ fset$  (**infixl**  $\langle | \cap | \rangle 65$ ) **where**  $xs\ | \cap | ys \equiv inf\ xs\ ys$

**abbreviation**  $fminus :: 'a\ fset \Rightarrow 'a\ fset \Rightarrow 'a\ fset$  (**infixl**  $\langle | - | \rangle 65$ ) **where**  $xs\ | - | ys \equiv minus\ xs\ ys$

**instantiation**  $fset :: (equal)\ equal$

**begin**

**definition**  $HOL.equal\ A\ B \longleftrightarrow A\ | \subseteq | B \wedge B\ | \subseteq | A$

**instance**  $\langle proof \rangle$

**end**

**instantiation**  $fset :: (type)\ conditionally\text{-}complete\text{-}lattice$

**begin**

**context includes**  $lifting\text{-}syntax$

**begin**

**lemma**  $right\text{-}total\text{-}Inf\text{-}fset\text{-}transfer:$

**assumes**  $[transfer\text{-}rule]: bi\text{-}unique\ A$  **and**  $[transfer\text{-}rule]: right\text{-}total\ A$

**shows**  $(rel\text{-}set\ (rel\text{-}set\ A) \implies rel\text{-}set\ A)$

$(\lambda S. \text{if finite } (\bigcap S \cap Collect\ (Domainp\ A)) \text{ then } \bigcap S \cap Collect\ (Domainp\ A)$   
 $\text{else } \{\})$

$(\lambda S. \text{if finite } (Inf\ S) \text{ then } Inf\ S \text{ else } \{\})$

$\langle proof \rangle$

**lemma**  $Inf\text{-}fset\text{-}transfer:$

**assumes**  $[transfer\text{-}rule]: bi\text{-}unique\ A$  **and**  $[transfer\text{-}rule]: bi\text{-}total\ A$

**shows**  $(rel\text{-}set\ (rel\text{-}set\ A) \implies rel\text{-}set\ A)$   $(\lambda A. \text{if finite } (Inf\ A) \text{ then } Inf\ A \text{ else } \{\})$

$(\lambda A. \text{if finite } (Inf\ A) \text{ then } Inf\ A \text{ else } \{\})$

$\langle proof \rangle$

**lift-definition**  $Inf\text{-}fset :: 'a\ fset\ set \Rightarrow 'a\ fset$  **is**  $\lambda A. \text{if finite } (Inf\ A) \text{ then } Inf\ A$   
 $\text{else } \{\}$

**parametric**  $right\text{-}total\text{-}Inf\text{-}fset\text{-}transfer\ Inf\text{-}fset\text{-}transfer\ \langle proof \rangle$

**lemma**  $Sup\text{-}fset\text{-}transfer:$

**assumes**  $[transfer\text{-}rule]: bi\text{-}unique\ A$

**shows**  $(rel\text{-}set\ (rel\text{-}set\ A) \implies rel\text{-}set\ A)$   $(\lambda A. \text{if finite } (Sup\ A) \text{ then } Sup\ A$   
 $\text{else } \{\})$

$(\lambda A. \text{if finite } (Sup\ A) \text{ then } Sup\ A \text{ else } \{\})\ \langle proof \rangle$

**lift-definition**  $Sup\text{-}fset :: 'a\ fset\ set \Rightarrow 'a\ fset$  **is**  $\lambda A. \text{if finite } (Sup\ A) \text{ then } Sup\ A$

*else* {}

**parametric** *Sup-fset-transfer* ⟨*proof*⟩

**lemma** *finite-Sup*:  $\exists z. \text{finite } z \wedge (\forall a. a \in X \longrightarrow a \leq z) \implies \text{finite } (\text{Sup } X)$   
 ⟨*proof*⟩

**lemma** *transfer-bdd-below*[*transfer-rule*]:  $(\text{rel-set } (\text{pcr-fset } (=)) \implies (=)) \text{ bdd-below}$   
*bdd-below*  
 ⟨*proof*⟩

**end**

**instance**

⟨*proof*⟩

**end**

**instantiation** *fset* :: (*finite*) *complete-lattice*

**begin**

**lift-definition** *top-fset* :: 'a *fset* **is** *UNIV* **parametric** *right-total-UNIV-transfer*  
*UNIV-transfer*  
 ⟨*proof*⟩

**instance**

⟨*proof*⟩

**end**

**instantiation** *fset* :: (*finite*) *complete-boolean-algebra*

**begin**

**lift-definition** *uminus-fset* :: 'a *fset*  $\Rightarrow$  'a *fset* **is** *uminus*  
**parametric** *right-total-Compl-transfer* *Compl-transfer* ⟨*proof*⟩

**instance**

⟨*proof*⟩

**end**

**abbreviation** *fUNIV* :: 'a::*finite* *fset* **where** *fUNIV*  $\equiv$  *top*

**abbreviation** *fuminus* :: 'a::*finite* *fset*  $\Rightarrow$  'a *fset* ( $\langle$ |-|  $\rightarrow$  [81] 80) **where** |-|  $x \equiv$   
*uminus*  $x$

**declare** *top-fset.rep-eq*[*simp*]

### 29.3 Other operations

**lift-definition** *finsert* :: 'a  $\Rightarrow$  'a *fset*  $\Rightarrow$  'a *fset* **is** *insert* **parametric** *Lifting-Set.insert-transfer*  
 ⟨*proof*⟩

**syntax**

$-fset :: args \Rightarrow 'a fset \ (\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix finite set enumeration} \rangle \rangle \{|\cdot|\}) \rangle$

**syntax-consts**

$-fset \Rightarrow finsert$

**translations**

$\{x, xs\} == CONST\ finsert\ x\ \{xs\}$

$\{x\} == CONST\ finsert\ x\ \{\}$

**abbreviation**  $fmember :: 'a \Rightarrow 'a fset \Rightarrow bool \ (\text{infix } \langle |\in| \rangle 50) \ \text{where}$

$x \ |\in| \ X \equiv x \in fset\ X$

**abbreviation**  $not-fmember :: 'a \Rightarrow 'a fset \Rightarrow bool \ (\text{infix } \langle |\notin| \rangle 50) \ \text{where}$

$x \ |\notin| \ X \equiv x \notin fset\ X$

**context****begin**

**qualified abbreviation**  $Ball :: 'a fset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool \ \text{where}$

$Ball\ X \equiv Set.Ball\ (fset\ X)$

**alias**  $fBall = FSet.Ball$

**qualified abbreviation**  $Bex :: 'a fset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool \ \text{where}$

$Bex\ X \equiv Set.Bex\ (fset\ X)$

**alias**  $fBex = FSet.Bex$

**end****syntax** (*input*)

$-fBall :: pptrn \Rightarrow 'a fset \Rightarrow bool \Rightarrow bool \ (\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite !} \rangle \rangle \langle -/|\cdot| \rangle \langle - \rangle \rangle [0, 0, 10] 10)$

$-fBex :: pptrn \Rightarrow 'a fset \Rightarrow bool \Rightarrow bool \ (\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite } ? \rangle \rangle \langle -/|\cdot| \rangle \langle - \rangle \rangle [0, 0, 10] 10)$

$-fBex1 :: pptrn \Rightarrow 'a fset \Rightarrow bool \Rightarrow bool \ (\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite } ?! \rangle \rangle \langle -/|\cdot| \rangle \langle - \rangle \rangle [0, 0, 10] 10)$

**syntax**

$-fBall :: pptrn \Rightarrow 'a fset \Rightarrow bool \Rightarrow bool \ (\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite } \forall \rangle \rangle \langle -/|\in| \rangle \langle - \rangle \rangle [0, 0, 10] 10)$

$-fBex :: pptrn \Rightarrow 'a fset \Rightarrow bool \Rightarrow bool \ (\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite } \exists \rangle \rangle \langle -/|\in| \rangle \langle - \rangle \rangle [0, 0, 10] 10)$

$-fBnex :: pptrn \Rightarrow 'a fset \Rightarrow bool \Rightarrow bool \ (\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite } \# \rangle \rangle \langle -/|\in| \rangle \langle - \rangle \rangle [0, 0, 10] 10)$

$-fBex1 :: pptrn \Rightarrow 'a fset \Rightarrow bool \Rightarrow bool \ (\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite } \exists! \rangle \rangle \langle -/|\in| \rangle \langle - \rangle \rangle [0, 0, 10] 10)$

**syntax-consts**

$-fBall\ -fBnex \Rightarrow fBall \ \text{and}$

$\neg fBex \equiv fBex$  **and**  
 $\neg fBex1 \equiv Ex1$

**translations**

$\forall x | \in | A. P \equiv \text{CONST } FSet.Ball \ A \ (\lambda x. P)$   
 $\exists x | \in | A. P \equiv \text{CONST } FSet.Bex \ A \ (\lambda x. P)$   
 $\nexists x | \in | A. P \equiv \text{CONST } fBall \ A \ (\lambda x. \neg P)$   
 $\exists ! x | \in | A. P \rightarrow \exists ! x. x | \in | A \wedge P$

$\langle ML \rangle$

**syntax**

$\text{-setlessfAll} :: [idt, 'a, bool] \Rightarrow bool \ (\langle \langle indent=3 \ notation=\langle binder \ finite \ \forall \ \rangle \ \forall \ -|C| \ - / \ - \rangle \ [0, 0, 10] \ 10)$   
 $\text{-setlessfEx} :: [idt, 'a, bool] \Rightarrow bool \ (\langle \langle indent=3 \ notation=\langle binder \ finite \ \exists \ \rangle \ \exists \ -|C| \ - / \ - \rangle \ [0, 0, 10] \ 10)$   
 $\text{-setlefAll} :: [idt, 'a, bool] \Rightarrow bool \ (\langle \langle indent=3 \ notation=\langle binder \ finite \ \forall \ \rangle \ \forall \ -| \subseteq | \ - / \ - \rangle \ [0, 0, 10] \ 10)$   
 $\text{-setlefEx} :: [idt, 'a, bool] \Rightarrow bool \ (\langle \langle indent=3 \ notation=\langle binder \ finite \ \exists \ \rangle \ \exists \ -| \subseteq | \ - / \ - \rangle \ [0, 0, 10] \ 10)$

**syntax-consts**

$\text{-setlessfAll} \ \text{-setlefAll} \equiv All$  **and**  
 $\text{-setlessfEx} \ \text{-setlefEx} \equiv Ex$

**translations**

$\forall A | \subseteq | B. P \rightarrow \forall A. A | \subseteq | B \rightarrow P$   
 $\exists A | \subseteq | B. P \rightarrow \exists A. A | \subseteq | B \wedge P$   
 $\forall A | \subseteq | B. P \rightarrow \forall A. A | \subseteq | B \rightarrow P$   
 $\exists A | \subseteq | B. P \rightarrow \exists A. A | \subseteq | B \wedge P$

**context includes** *lifting-syntax*

**begin**

**lemma** *fmember-transfer0*[*transfer-rule*]:

**assumes** [*transfer-rule*]: *bi-unique A*  
**shows** ( $A \equiv \equiv \equiv \rightarrow pcr\text{-fset } A \equiv \equiv \equiv \rightarrow (=)$ ) ( $\in$ ) ( $| \in |$ )  
 $\langle proof \rangle$

**lemma** *fBall-transfer0*[*transfer-rule*]:

**assumes** [*transfer-rule*]: *bi-unique A*  
**shows** ( $pcr\text{-fset } A \equiv \equiv \equiv \rightarrow (A \equiv \equiv \equiv \rightarrow (=)) \equiv \equiv \equiv \rightarrow (=)$ ) (*Ball*) (*fBall*)  
 $\langle proof \rangle$

**lemma** *fBex-transfer0*[*transfer-rule*]:

**assumes** [*transfer-rule*]: *bi-unique A*  
**shows** ( $pcr\text{-fset } A \equiv \equiv \equiv \rightarrow (A \equiv \equiv \equiv \rightarrow (=)) \equiv \equiv \equiv \rightarrow (=)$ ) (*Bex*) (*fBex*)  
 $\langle proof \rangle$

**lift-definition**  $ffilter :: ('a \Rightarrow bool) \Rightarrow 'a \text{ fset} \Rightarrow 'a \text{ fset}$  **is** *Set.filter*  
**parametric** *Lifting-Set.filter-transfer*  $\langle proof \rangle$

**lift-definition**  $fPow :: 'a \text{ fset} \Rightarrow 'a \text{ fset fset}$  **is** *Pow* **parametric** *Pow-transfer*  
 $\langle proof \rangle$

**lift-definition**  $fcard :: 'a \text{ fset} \Rightarrow nat$  **is** *card* **parametric** *card-transfer*  $\langle proof \rangle$

**lift-definition**  $fimage :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ fset} \Rightarrow 'b \text{ fset}$  (**infixr**  $\langle | \rangle$  90) **is** *image*  
**parametric** *image-transfer*  $\langle proof \rangle$

**lift-definition**  $fthe-elem :: 'a \text{ fset} \Rightarrow 'a$  **is** *the-elem*  $\langle proof \rangle$

**lift-definition**  $fbind :: 'a \text{ fset} \Rightarrow ('a \Rightarrow 'b \text{ fset}) \Rightarrow 'b \text{ fset}$  **is** *Set.bind* **parametric**  
*bind-transfer*  
 $\langle proof \rangle$

**lift-definition**  $ffUnion :: 'a \text{ fset fset} \Rightarrow 'a \text{ fset}$  **is** *Union* **parametric** *Union-transfer*  
 $\langle proof \rangle$

**lift-definition**  $ffold :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ fset} \Rightarrow 'b$  **is** *Finite-Set.fold*  $\langle proof \rangle$

**lift-definition**  $fset-of-list :: 'a \text{ list} \Rightarrow 'a \text{ fset}$  **is** *set*  $\langle proof \rangle$

**lift-definition**  $sorted-list-of-fset :: 'a::linorder \text{ fset} \Rightarrow 'a \text{ list}$  **is** *sorted-list-of-set*  
 $\langle proof \rangle$

## 29.4 Transferred lemmas from Set.thy

**lemma**  $fset-eqI: (\bigwedge x. (x \in A) = (x \in B)) \Longrightarrow A = B$   
 $\langle proof \rangle$

**lemma**  $fset-eq-iff[no-atp]: (A = B) = (\bigvee x. (x \in A) = (x \in B))$   
 $\langle proof \rangle$

**lemma**  $fBallI[no-atp]: (\bigwedge x. x \in A \Longrightarrow P x) \Longrightarrow fBall A P$   
 $\langle proof \rangle$

**lemma**  $fbspec[no-atp]: fBall A P \Longrightarrow x \in A \Longrightarrow P x$   
 $\langle proof \rangle$

**lemma**  $fBallE[no-atp]: fBall A P \Longrightarrow (P x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$   
 $\langle proof \rangle$

**lemma**  $fBexI[no-atp]: P x \Longrightarrow x \in A \Longrightarrow fBex A P$   
 $\langle proof \rangle$

**lemma**  $rev-fBexI[no-atp]: x \in A \Longrightarrow P x \Longrightarrow fBex A P$



*<proof>*

**lemma** *fBexCI[no-atp]*:  $(fBall\ A\ (\lambda x. \neg P\ x) \implies P\ a) \implies a\ |\in|\ A \implies fBex\ A\ P$   
*<proof>*

**lemma** *fBexE[no-atp]*:  $fBex\ A\ P \implies (\bigwedge x. x\ |\in|\ A \implies P\ x \implies Q) \implies Q$   
*<proof>*

**lemma** *fBall-triv[no-atp]*:  $fBall\ A\ (\lambda x. P) = ((\exists x. x\ |\in|\ A) \longrightarrow P)$   
*<proof>*

**lemma** *fBex-triv[no-atp]*:  $fBex\ A\ (\lambda x. P) = ((\exists x. x\ |\in|\ A) \wedge P)$   
*<proof>*

**lemma** *fBex-triv-one-point1[no-atp]*:  $fBex\ A\ (\lambda x. x = a) = (a\ |\in|\ A)$   
*<proof>*

**lemma** *fBex-triv-one-point2[no-atp]*:  $fBex\ A\ ((=)\ a) = (a\ |\in|\ A)$   
*<proof>*

**lemma** *fBex-one-point1[no-atp]*:  $fBex\ A\ (\lambda x. x = a \wedge P\ x) = (a\ |\in|\ A \wedge P\ a)$   
*<proof>*

**lemma** *fBex-one-point2[no-atp]*:  $fBex\ A\ (\lambda x. a = x \wedge P\ x) = (a\ |\in|\ A \wedge P\ a)$   
*<proof>*

**lemma** *fBall-one-point1[no-atp]*:  $fBall\ A\ (\lambda x. x = a \longrightarrow P\ x) = (a\ |\in|\ A \longrightarrow P\ a)$   
*<proof>*

**lemma** *fBall-one-point2[no-atp]*:  $fBall\ A\ (\lambda x. a = x \longrightarrow P\ x) = (a\ |\in|\ A \longrightarrow P\ a)$   
*<proof>*

**lemma** *fBall-conj-distrib*:  $fBall\ A\ (\lambda x. P\ x \wedge Q\ x) = (fBall\ A\ P \wedge fBall\ A\ Q)$   
*<proof>*

**lemma** *fBex-disj-distrib*:  $fBex\ A\ (\lambda x. P\ x \vee Q\ x) = (fBex\ A\ P \vee fBex\ A\ Q)$   
*<proof>*

**lemma** *fBall-cong[fundef-cong]*:  $A = B \implies (\bigwedge x. x\ |\in|\ B \implies P\ x = Q\ x) \implies fBall\ A\ P = fBall\ B\ Q$   
*<proof>*

**lemma** *fBex-cong[fundef-cong]*:  $A = B \implies (\bigwedge x. x\ |\in|\ B \implies P\ x = Q\ x) \implies fBex\ A\ P = fBex\ B\ Q$   
*<proof>*

**lemma** *fsubsetI[introl]*:  $(\bigwedge x. x\ |\in|\ A \implies x\ |\in|\ B) \implies A\ |\subseteq|\ B$

*<proof>*

**lemma** *fsubsetD[elim, intro?]*:  $A \mid\subseteq\mid B \implies c \mid\in\mid A \implies c \mid\in\mid B$   
*<proof>*

**lemma** *rev-fsubsetD[no-atp, intro?]*:  $c \mid\in\mid A \implies A \mid\subseteq\mid B \implies c \mid\in\mid B$   
*<proof>*

**lemma** *fsubsetCE[no-atp, elim]*:  $A \mid\subseteq\mid B \implies (c \mid\notin\mid A \implies P) \implies (c \mid\in\mid B \implies P) \implies P$   
*<proof>*

**lemma** *fsubset-eq[no-atp]*:  $(A \mid\subseteq\mid B) = fBall\ A\ (\lambda x. x \mid\in\mid B)$   
*<proof>*

**lemma** *contra-fsubsetD[no-atp]*:  $A \mid\subseteq\mid B \implies c \mid\notin\mid B \implies c \mid\notin\mid A$   
*<proof>*

**lemma** *fsubset-refl*:  $A \mid\subseteq\mid A$   
*<proof>*

**lemma** *fsubset-trans*:  $A \mid\subseteq\mid B \implies B \mid\subseteq\mid C \implies A \mid\subseteq\mid C$   
*<proof>*

**lemma** *fset-rev-mp*:  $c \mid\in\mid A \implies A \mid\subseteq\mid B \implies c \mid\in\mid B$   
*<proof>*

**lemma** *fset-mp*:  $A \mid\subseteq\mid B \implies c \mid\in\mid A \implies c \mid\in\mid B$   
*<proof>*

**lemma** *fsubset-not-fsubset-eq[code]*:  $(A \mid\subset\mid B) = (A \mid\subseteq\mid B \wedge \neg B \mid\subseteq\mid A)$   
*<proof>*

**lemma** *eq-fmem-trans*:  $a = b \implies b \mid\in\mid A \implies a \mid\in\mid A$   
*<proof>*

**lemma** *fsubset-antisym[intro!]*:  $A \mid\subseteq\mid B \implies B \mid\subseteq\mid A \implies A = B$   
*<proof>*

**lemma** *fequalityD1*:  $A = B \implies A \mid\subseteq\mid B$   
*<proof>*

**lemma** *fequalityD2*:  $A = B \implies B \mid\subseteq\mid A$   
*<proof>*

**lemma** *fequalityE*:  $A = B \implies (A \mid\subseteq\mid B \implies B \mid\subseteq\mid A \implies P) \implies P$   
*<proof>*

**lemma** *fequalityCE[elim]*:

$A = B \implies (c \in A \implies c \in B \implies P) \implies (c \notin A \implies c \notin B \implies P) \implies P$   
 ⟨proof⟩

**lemma** *eqfset-imp-iff*:  $A = B \implies (x \in A) = (x \in B)$   
 ⟨proof⟩

**lemma** *eqfelem-imp-iff*:  $x = y \implies (x \in A) = (y \in A)$   
 ⟨proof⟩

**lemma** *fempty-iff[simp]*:  $(c \in \{\}) = False$   
 ⟨proof⟩

**lemma** *fempty-fsubsetI[iff]*:  $\{\} \subseteq x$   
 ⟨proof⟩

**lemma** *equalsffemptyI*:  $(\bigwedge y. y \in A \implies False) \implies A = \{\}$   
 ⟨proof⟩

**lemma** *equalsffemptyD*:  $A = \{\} \implies a \notin A$   
 ⟨proof⟩

**lemma** *fBall-fempty[simp]*:  $fBall \{\} P = True$   
 ⟨proof⟩

**lemma** *fBex-fempty[simp]*:  $fBex \{\} P = False$   
 ⟨proof⟩

**lemma** *fPow-iff[iff]*:  $(A \in fPow B) = (A \subseteq B)$   
 ⟨proof⟩

**lemma** *fPowI*:  $A \subseteq B \implies A \in fPow B$   
 ⟨proof⟩

**lemma** *fPowD*:  $A \in fPow B \implies A \subseteq B$   
 ⟨proof⟩

**lemma** *fPow-bottom*:  $\{\} \in fPow B$   
 ⟨proof⟩

**lemma** *fPow-top*:  $A \in fPow A$   
 ⟨proof⟩

**lemma** *fPow-not-fempty*:  $fPow A \neq \{\}$   
 ⟨proof⟩

**lemma** *finter-iff[simp]*:  $(c \in A \cap B) = (c \in A \wedge c \in B)$   
 ⟨proof⟩

**lemma** *finterI[intro!]*:  $c \in A \implies c \in B \implies c \in A \cap B$

*<proof>*

**lemma** *finterD1*:  $c \in A \cap B \implies c \in A$   
*<proof>*

**lemma** *finterD2*:  $c \in A \cap B \implies c \in B$   
*<proof>*

**lemma** *finterE[elim!]*:  $c \in A \cap B \implies (c \in A \implies c \in B \implies P) \implies P$   
*<proof>*

**lemma** *funion-iff[simp]*:  $(c \in A \cup B) = (c \in A \vee c \in B)$   
*<proof>*

**lemma** *funionI1[elim?]*:  $c \in A \implies c \in A \cup B$   
*<proof>*

**lemma** *funionI2[elim?]*:  $c \in B \implies c \in A \cup B$   
*<proof>*

**lemma** *funionCI[intro!]*:  $(c \notin B \implies c \in A) \implies c \in A \cup B$   
*<proof>*

**lemma** *funionE[elim!]*:  $c \in A \cup B \implies (c \in A \implies P) \implies (c \in B \implies P) \implies P$   
*<proof>*

**lemma** *fminus-iff[simp]*:  $(c \in A - B) = (c \in A \wedge c \notin B)$   
*<proof>*

**lemma** *fminusI[intro!]*:  $c \in A \implies c \notin B \implies c \in A - B$   
*<proof>*

**lemma** *fminusD1*:  $c \in A - B \implies c \in A$   
*<proof>*

**lemma** *fminusD2*:  $c \in A - B \implies c \in B \implies P$   
*<proof>*

**lemma** *fminusE[elim!]*:  $c \in A - B \implies (c \in A \implies c \notin B \implies P) \implies P$   
*<proof>*

**lemma** *finsert-iff[simp]*:  $(a \in \text{finsert } b \ A) = (a = b \vee a \in A)$   
*<proof>*

**lemma** *finsertI1*:  $a \in \text{finsert } a \ B$   
*<proof>*

**lemma** *finsertI2*:  $a \in B \implies a \in \text{finsert } b \ B$

*<proof>*

**lemma** *finsertE[elim!]*:  $a \in | \text{finsert } b \ A \implies (a = b \implies P) \implies (a \in | \ A \implies P) \implies P$   
*<proof>*

**lemma** *finsertCI[intro!]*:  $(a \notin | \ B \implies a = b) \implies a \in | \ \text{finsert } b \ B$   
*<proof>*

**lemma** *fsubset-finsert-iff*:  
 $(A \subseteq | \ \text{finsert } x \ B) = (if \ x \ \in | \ A \ \text{then } A \ |- \ \{|x|\} \ \subseteq | \ B \ \text{else } A \ \subseteq | \ B)$   
*<proof>*

**lemma** *finsert-ident*:  $x \notin | \ A \implies x \notin | \ B \implies (\text{finsert } x \ A = \text{finsert } x \ B) = (A = B)$   
*<proof>*

**lemma** *fsingletonI[intro!,no-atp]*:  $a \in | \ \{|a|\}$   
*<proof>*

**lemma** *fsingletonD[dest!,no-atp]*:  $b \in | \ \{|a|\} \implies b = a$   
*<proof>*

**lemma** *fsingleton-iff*:  $(b \in | \ \{|a|\}) = (b = a)$   
*<proof>*

**lemma** *fsingleton-inject[dest!]*:  $\{|a|\} = \{|b|\} \implies a = b$   
*<proof>*

**lemma** *fsingleton-finsert-inj-eq[iff,no-atp]*:  $(\{|b|\} = \text{finsert } a \ A) = (a = b \wedge A \subseteq | \ \{|b|\})$   
*<proof>*

**lemma** *fsingleton-finsert-inj-eq'[iff,no-atp]*:  $(\text{finsert } a \ A = \{|b|\}) = (a = b \wedge A \subseteq | \ \{|b|\})$   
*<proof>*

**lemma** *fsubset-fsingletonD*:  $A \subseteq | \ \{|x|\} \implies A = \{|\} \vee A = \{|x|\}$   
*<proof>*

**lemma** *fminus-single-finsert*:  $A \ |- \ \{|x|\} \ \subseteq | \ B \implies A \ \subseteq | \ \text{finsert } x \ B$   
*<proof>*

**lemma** *fdoubleton-eq-iff*:  $(\{|a, b|\} = \{|c, d|\}) = (a = c \wedge b = d \vee a = d \wedge b = c)$   
*<proof>*

**lemma** *funion-fsingleton-iff*:  
 $(A \cup | \ B = \{|x|\}) = (A = \{|\} \wedge B = \{|x|\} \vee A = \{|x|\} \wedge B = \{|\} \vee A = \{|x|\})$

$$\wedge B = \{|x|\}$$

*<proof>*

**lemma** *fsingleton-funion-iff*:

$$(\{|x|\} = A \mid \cup \mid B) = (A = \{|\}\ \wedge B = \{|x|\} \vee A = \{|x|\} \wedge B = \{|\}\ \vee A = \{|x|\} \wedge B = \{|x|\})$$

*<proof>*

**lemma** *fimage-eqI[simp, intro]*:  $b = f x \implies x \mid \in \mid A \implies b \mid \in \mid f \mid \uparrow \mid A$

*<proof>*

**lemma** *fimageI*:  $x \mid \in \mid A \implies f x \mid \in \mid f \mid \uparrow \mid A$

*<proof>*

**lemma** *rev-fimage-eqI*:  $x \mid \in \mid A \implies b = f x \implies b \mid \in \mid f \mid \uparrow \mid A$

*<proof>*

**lemma** *fimageE[elim!]*:  $b \mid \in \mid f \mid \uparrow \mid A \implies (\wedge x. b = f x \implies x \mid \in \mid A \implies \textit{thesis}) \implies \textit{thesis}$

*<proof>*

**lemma** *Compr-fimage-eq*:  $\{x. x \mid \in \mid f \mid \uparrow \mid A \wedge P x\} = f \textit{' } \{x. x \mid \in \mid A \wedge P (f x)\}$

*<proof>*

**lemma** *fimage-funion*:  $f \mid \uparrow \mid (A \mid \cup \mid B) = f \mid \uparrow \mid A \mid \cup \mid f \mid \uparrow \mid B$

*<proof>*

**lemma** *fimage-iff*:  $(z \mid \in \mid f \mid \uparrow \mid A) = fBex A (\lambda x. z = f x)$

*<proof>*

**lemma** *fimage-fsubset-iff[no-atp]*:  $(f \mid \uparrow \mid A \mid \subseteq \mid B) = fBall A (\lambda x. f x \mid \in \mid B)$

*<proof>*

**lemma** *fimage-fsubsetI*:  $(\wedge x. x \mid \in \mid A \implies f x \mid \in \mid B) \implies f \mid \uparrow \mid A \mid \subseteq \mid B$

*<proof>*

**lemma** *fimage-ident[simp]*:  $(\lambda x. x) \mid \uparrow \mid Y = Y$

*<proof>*

**lemma** *if-split-fmem1*:  $((\textit{if } Q \textit{ then } x \textit{ else } y) \mid \in \mid b) = ((Q \longrightarrow x \mid \in \mid b) \wedge (\neg Q \longrightarrow y \mid \in \mid b))$

*<proof>*

**lemma** *if-split-fmem2*:  $(a \mid \in \mid (\textit{if } Q \textit{ then } x \textit{ else } y)) = ((Q \longrightarrow a \mid \in \mid x) \wedge (\neg Q \longrightarrow a \mid \in \mid y))$

*<proof>*

**lemma** *pfsubsetI[intro!,no-atp]*:  $A \mid \subseteq \mid B \implies A \neq B \implies A \mid \subset \mid B$

*<proof>*

**lemma** *pfsubsetE[elim!,no-atp]*:  $A \mid\subset\mid B \implies (A \mid\subseteq\mid B \implies \neg B \mid\subseteq\mid A \implies R) \implies R$   
 ⟨proof⟩

**lemma** *pfsubset-finsert-iff*:  
 $(A \mid\subset\mid \text{finsert } x \ B) =$   
 (if  $x \mid\in\mid B$  then  $A \mid\subset\mid B$  else if  $x \mid\in\mid A$  then  $A \mid\mid\mid \{x\} \mid\subset\mid B$  else  $A \mid\subseteq\mid B$ )  
 ⟨proof⟩

**lemma** *pfsubset-eq*:  $(A \mid\subset\mid B) = (A \mid\subseteq\mid B \wedge A \neq B)$   
 ⟨proof⟩

**lemma** *pfsubset-imp-fsubset*:  $A \mid\subset\mid B \implies A \mid\subseteq\mid B$   
 ⟨proof⟩

**lemma** *pfsubset-trans*:  $A \mid\subset\mid B \implies B \mid\subset\mid C \implies A \mid\subset\mid C$   
 ⟨proof⟩

**lemma** *pfsubsetD*:  $A \mid\subset\mid B \implies c \mid\in\mid A \implies c \mid\in\mid B$   
 ⟨proof⟩

**lemma** *pfsubset-fsubset-trans*:  $A \mid\subset\mid B \implies B \mid\subseteq\mid C \implies A \mid\subset\mid C$   
 ⟨proof⟩

**lemma** *fsubset-pfsubset-trans*:  $A \mid\subseteq\mid B \implies B \mid\subset\mid C \implies A \mid\subset\mid C$   
 ⟨proof⟩

**lemma** *pfsubset-imp-ex-fmem*:  $A \mid\subset\mid B \implies \exists b. b \mid\in\mid B \mid\mid\mid A$   
 ⟨proof⟩

**lemma** *fimage-fPow-mono*:  $f \mid\uparrow\mid A \mid\subseteq\mid B \implies (\mid\uparrow\mid) f \mid\uparrow\mid \text{fPow } A \mid\subseteq\mid \text{fPow } B$   
 ⟨proof⟩

**lemma** *fimage-fPow-surj*:  $f \mid\uparrow\mid A = B \implies (\mid\uparrow\mid) f \mid\uparrow\mid \text{fPow } A = \text{fPow } B$   
 ⟨proof⟩

**lemma** *fsubset-finsertI*:  $B \mid\subseteq\mid \text{finsert } a \ B$   
 ⟨proof⟩

**lemma** *fsubset-finsertI2*:  $A \mid\subseteq\mid B \implies A \mid\subseteq\mid \text{finsert } b \ B$   
 ⟨proof⟩

**lemma** *fsubset-finsert*:  $x \mid\notin\mid A \implies (A \mid\subseteq\mid \text{finsert } x \ B) = (A \mid\subseteq\mid B)$   
 ⟨proof⟩

**lemma** *funion-upper1*:  $A \mid\subseteq\mid A \mid\cup\mid B$   
 ⟨proof⟩

**lemma** *funion-upper2*:  $B \subseteq A \cup B$   
 ⟨proof⟩

**lemma** *funion-least*:  $A \subseteq C \implies B \subseteq C \implies A \cup B \subseteq C$   
 ⟨proof⟩

**lemma** *finter-lower1*:  $A \cap B \subseteq A$   
 ⟨proof⟩

**lemma** *finter-lower2*:  $A \cap B \subseteq B$   
 ⟨proof⟩

**lemma** *finter-greatest*:  $C \subseteq A \implies C \subseteq B \implies C \subseteq A \cap B$   
 ⟨proof⟩

**lemma** *fminus-fsubset*:  $A \setminus B \subseteq A$   
 ⟨proof⟩

**lemma** *fminus-fsubset-conv*:  $(A \setminus B \subseteq C) = (A \subseteq B \cup C)$   
 ⟨proof⟩

**lemma** *fsubset-fempty[simp]*:  $(A \subseteq \{\}) = (A = \{\})$   
 ⟨proof⟩

**lemma** *not-pfsubset-fempty[iff]*:  $\neg A \subset \{\}$   
 ⟨proof⟩

**lemma** *finsert-is-funion*:  $finsert\ a\ A = \{a\} \cup A$   
 ⟨proof⟩

**lemma** *finsert-not-fempty[simp]*:  $finsert\ a\ A \neq \{\}$   
 ⟨proof⟩

**lemma** *fempty-not-finsert*:  $\{\} \neq finsert\ a\ A$   
 ⟨proof⟩

**lemma** *finsert-absorb*:  $a \in A \implies finsert\ a\ A = A$   
 ⟨proof⟩

**lemma** *finsert-absorb2[simp]*:  $finsert\ x\ (finsert\ x\ A) = finsert\ x\ A$   
 ⟨proof⟩

**lemma** *finsert-commute*:  $finsert\ x\ (finsert\ y\ A) = finsert\ y\ (finsert\ x\ A)$   
 ⟨proof⟩

**lemma** *finsert-fsubset[simp]*:  $(finsert\ x\ A \subseteq B) = (x \in B \wedge A \subseteq B)$   
 ⟨proof⟩

**lemma** *finsert-inter-finsert[simp]*:  $finsert\ a\ A \cap finsert\ a\ B = finsert\ a\ (A \cap B)$



*<proof>*

**lemma** *finsert-disjoint[simp,no-atp]*:

$(finsert\ a\ A\ |\cap| B = \{\|\}) = (a\ |\notin| B \wedge A\ |\cap| B = \{\|\})$   
 $(\{\|\} = finsert\ a\ A\ |\cap| B) = (a\ |\notin| B \wedge \{\|\} = A\ |\cap| B)$   
*<proof>*

**lemma** *disjoint-finsert[simp,no-atp]*:

$(B\ |\cap| finsert\ a\ A = \{\|\}) = (a\ |\notin| B \wedge B\ |\cap| A = \{\|\})$   
 $(\{\|\} = A\ |\cap| finsert\ b\ B) = (b\ |\notin| A \wedge \{\|\} = A\ |\cap| B)$   
*<proof>*

**lemma** *fimage-fempty[simp]*:  $f\ |\uparrow| \{\|\} = \{\|\}$

*<proof>*

**lemma** *fimage-finsert[simp]*:  $f\ |\uparrow| finsert\ a\ B = finsert\ (f\ a)\ (f\ |\uparrow| B)$

*<proof>*

**lemma** *fimage-constant*:  $x\ |\in| A \implies (\lambda x. c)\ |\uparrow| A = \{c\}$

*<proof>*

**lemma** *fimage-constant-conv*:  $(\lambda x. c)\ |\uparrow| A = (if\ A = \{\|\}\ then\ \{\|\}\ else\ \{c\})$

*<proof>*

**lemma** *fimage-fimage*:  $f\ |\uparrow| g\ |\uparrow| A = (\lambda x. f\ (g\ x))\ |\uparrow| A$

*<proof>*

**lemma** *finsert-fimage[simp]*:  $x\ |\in| A \implies finsert\ (f\ x)\ (f\ |\uparrow| A) = f\ |\uparrow| A$

*<proof>*

**lemma** *fimage-is-fempty[iff]*:  $(f\ |\uparrow| A = \{\|\}) = (A = \{\|\})$

*<proof>*

**lemma** *fempty-is-fimage[iff]*:  $(\{\|\} = f\ |\uparrow| A) = (A = \{\|\})$

*<proof>*

**lemma** *fimage-cong*:  $M = N \implies (\bigwedge x. x\ |\in| N \implies f\ x = g\ x) \implies f\ |\uparrow| M = g\ |\uparrow| N$

*<proof>*

**lemma** *fimage-finter-fsubset*:  $f\ |\uparrow| (A\ |\cap| B) \subseteq f\ |\uparrow| A\ |\cap| f\ |\uparrow| B$

*<proof>*

**lemma** *fimage-fminus-fsubset*:  $f\ |\uparrow| A\ |\minus| f\ |\uparrow| B \subseteq f\ |\uparrow| (A\ |\minus| B)$

*<proof>*

**lemma** *finter-absorb*:  $A\ |\cap| A = A$

*<proof>*

**lemma** *finter-left-absorb*:  $A \mid \cap \mid (A \mid \cap \mid B) = A \mid \cap \mid B$   
 ⟨proof⟩

**lemma** *finter-commute*:  $A \mid \cap \mid B = B \mid \cap \mid A$   
 ⟨proof⟩

**lemma** *finter-left-commute*:  $A \mid \cap \mid (B \mid \cap \mid C) = B \mid \cap \mid (A \mid \cap \mid C)$   
 ⟨proof⟩

**lemma** *finter-assoc*:  $A \mid \cap \mid B \mid \cap \mid C = A \mid \cap \mid (B \mid \cap \mid C)$   
 ⟨proof⟩

**lemma** *finter-ac*:  
 $A \mid \cap \mid B \mid \cap \mid C = A \mid \cap \mid (B \mid \cap \mid C)$   
 $A \mid \cap \mid (A \mid \cap \mid B) = A \mid \cap \mid B$   
 $A \mid \cap \mid B = B \mid \cap \mid A$   
 $A \mid \cap \mid (B \mid \cap \mid C) = B \mid \cap \mid (A \mid \cap \mid C)$   
 ⟨proof⟩

**lemma** *finter-absorb1*:  $B \mid \subseteq \mid A \implies A \mid \cap \mid B = B$   
 ⟨proof⟩

**lemma** *finter-absorb2*:  $A \mid \subseteq \mid B \implies A \mid \cap \mid B = A$   
 ⟨proof⟩

**lemma** *finter-fempty-left*:  $\{\mid\} \mid \cap \mid B = \{\mid\}$   
 ⟨proof⟩

**lemma** *finter-fempty-right*:  $A \mid \cap \mid \{\mid\} = \{\mid\}$   
 ⟨proof⟩

**lemma** *disjoint-iff-fnot-equal*:  $(A \mid \cap \mid B = \{\mid\}) = fBall A (\lambda x. fBall B ((\neq) x))$   
 ⟨proof⟩

**lemma** *finter-funion-distrib*:  $A \mid \cap \mid (B \mid \cup \mid C) = A \mid \cap \mid B \mid \cup \mid (A \mid \cap \mid C)$   
 ⟨proof⟩

**lemma** *finter-funion-distrib2*:  $B \mid \cup \mid C \mid \cap \mid A = B \mid \cap \mid A \mid \cup \mid (C \mid \cap \mid A)$   
 ⟨proof⟩

**lemma** *finter-fsubset-iff[no-atp, simp]*:  $(C \mid \subseteq \mid A \mid \cap \mid B) = (C \mid \subseteq \mid A \wedge C \mid \subseteq \mid B)$   
 ⟨proof⟩

**lemma** *funion-absorb*:  $A \mid \cup \mid A = A$   
 ⟨proof⟩

**lemma** *funion-left-absorb*:  $A \mid \cup \mid (A \mid \cup \mid B) = A \mid \cup \mid B$   
 ⟨proof⟩

**lemma** *funion-commute*:  $A \mid\cup\mid B = B \mid\cup\mid A$   
 ⟨proof⟩

**lemma** *funion-left-commute*:  $A \mid\cup\mid (B \mid\cup\mid C) = B \mid\cup\mid (A \mid\cup\mid C)$   
 ⟨proof⟩

**lemma** *funion-assoc*:  $A \mid\cup\mid B \mid\cup\mid C = A \mid\cup\mid (B \mid\cup\mid C)$   
 ⟨proof⟩

**lemma** *funion-ac*:  
 $A \mid\cup\mid B \mid\cup\mid C = A \mid\cup\mid (B \mid\cup\mid C)$   
 $A \mid\cup\mid (A \mid\cup\mid B) = A \mid\cup\mid B$   
 $A \mid\cup\mid B = B \mid\cup\mid A$   
 $A \mid\cup\mid (B \mid\cup\mid C) = B \mid\cup\mid (A \mid\cup\mid C)$   
 ⟨proof⟩

**lemma** *funion-absorb1*:  $A \mid\subseteq\mid B \implies A \mid\cup\mid B = B$   
 ⟨proof⟩

**lemma** *funion-absorb2*:  $B \mid\subseteq\mid A \implies A \mid\cup\mid B = A$   
 ⟨proof⟩

**lemma** *funion-fempty-left*:  $\{\mid\mid\} \mid\cup\mid B = B$   
 ⟨proof⟩

**lemma** *funion-fempty-right*:  $A \mid\cup\mid \{\mid\mid\} = A$   
 ⟨proof⟩

**lemma** *funion-finsert-left[simp]*:  $finsert\ a\ B \mid\cup\mid C = finsert\ a\ (B \mid\cup\mid C)$   
 ⟨proof⟩

**lemma** *funion-finsert-right[simp]*:  $A \mid\cup\mid finsert\ a\ B = finsert\ a\ (A \mid\cup\mid B)$   
 ⟨proof⟩

**lemma** *finter-finsert-left*:  $finsert\ a\ B \mid\cap\mid C = (if\ a \mid\in\mid C\ then\ finsert\ a\ (B \mid\cap\mid C)\ else\ B \mid\cap\mid C)$   
 ⟨proof⟩

**lemma** *finter-finsert-left-iffempty[simp]*:  $a \mid\notin\mid C \implies finsert\ a\ B \mid\cap\mid C = B \mid\cap\mid C$   
 ⟨proof⟩

**lemma** *finter-finsert-left-if1[simp]*:  $a \mid\in\mid C \implies finsert\ a\ B \mid\cap\mid C = finsert\ a\ (B \mid\cap\mid C)$   
 ⟨proof⟩

**lemma** *finter-finsert-right*:  
 $A \mid\cap\mid finsert\ a\ B = (if\ a \mid\in\mid A\ then\ finsert\ a\ (A \mid\cap\mid B)\ else\ A \mid\cap\mid B)$   
 ⟨proof⟩

**lemma** *finter-finsert-right-iffempty[simp]*:  $a \notin A \implies A \cap \text{finsert } a B = A \cap B$

*<proof>*

**lemma** *finter-finsert-right-if1[simp]*:  $a \in A \implies A \cap \text{finsert } a B = \text{finsert } a (A \cap B)$

*<proof>*

**lemma** *funion-finter-distrib*:  $A \cup (B \cap C) = A \cup B \cap (A \cup C)$

*<proof>*

**lemma** *funion-finter-distrib2*:  $B \cap C \cup A = B \cup A \cap (C \cup A)$

*<proof>*

**lemma** *funion-finter-crazy*:

$A \cap B \cup (B \cap C) \cup (C \cap A) = A \cup B \cap (B \cup C) \cap (C \cup A)$

*<proof>*

**lemma** *fsubset-funion-eq*:  $(A \subseteq B) = (A \cup B = B)$

*<proof>*

**lemma** *funion-fempty[iff]*:  $(A \cup B = \{\}) = (A = \{\} \wedge B = \{\})$

*<proof>*

**lemma** *funion-fsubset-iff[no-atp, simp]*:  $(A \cup B \subseteq C) = (A \subseteq C \wedge B \subseteq C)$

*<proof>*

**lemma** *funion-fminus-finter*:  $A - B \cup (A \cap B) = A$

*<proof>*

**lemma** *ffunion-empty[simp]*:  $\text{ffUnion } \{\} = \{\}$

*<proof>*

**lemma** *ffunion-mono*:  $A \subseteq B \implies \text{ffUnion } A \subseteq \text{ffUnion } B$

*<proof>*

**lemma** *ffunion-insert[simp]*:  $\text{ffUnion } (\text{finsert } a B) = a \cup \text{ffUnion } B$

*<proof>*

**lemma** *fminus-finter2*:  $A \cap C - (B \cap C) = A \cap C - B$

*<proof>*

**lemma** *funion-finter-assoc-eq*:  $(A \cap B \cup C = A \cap (B \cup C)) = (C \subseteq A)$

*<proof>*

**lemma** *fBall-funion*:  $\text{fBall } (A \cup B) P = (\text{fBall } A P \wedge \text{fBall } B P)$

*<proof>*

**lemma** *fBex-funion*:  $\text{fBex } (A \cup B) P = (\text{fBex } A P \vee \text{fBex } B P)$

*<proof>*

**lemma** *fminus-eq-fempty-iff*[simp,no-atp]:  $(A \text{ |-| } B = \{\}) = (A \text{ |}\subseteq\text{ } B)$   
*<proof>*

**lemma** *fminus-cancel*[simp]:  $A \text{ |-| } A = \{\}$   
*<proof>*

**lemma** *fminus-idemp*[simp]:  $A \text{ |-| } B \text{ |-| } B = A \text{ |-| } B$   
*<proof>*

**lemma** *fminus-triv*:  $A \text{ |}\cap\text{ } B = \{\} \implies A \text{ |-| } B = A$   
*<proof>*

**lemma** *fempty-fminus*[simp]:  $\{\} \text{ |-| } A = \{\}$   
*<proof>*

**lemma** *fminus-fempty*[simp]:  $A \text{ |-| } \{\} = A$   
*<proof>*

**lemma** *fminus-finsertffempty*[simp,no-atp]:  $x \notin A \implies A \text{ |-| } \text{finsert } x B = A \text{ |-| } B$   
*<proof>*

**lemma** *fminus-finsert*:  $A \text{ |-| } \text{finsert } a B = A \text{ |-| } B \text{ |-| } \{a\}$   
*<proof>*

**lemma** *fminus-finsert2*:  $A \text{ |-| } \text{finsert } a B = A \text{ |-| } \{a\} \text{ |-| } B$   
*<proof>*

**lemma** *finsert-fminus-if*:  $\text{finsert } x A \text{ |-| } B = (\text{if } x \in B \text{ then } A \text{ |-| } B \text{ else } \text{finsert } x (A \text{ |-| } B))$   
*<proof>*

**lemma** *finsert-fminus1*[simp]:  $x \in B \implies \text{finsert } x A \text{ |-| } B = A \text{ |-| } B$   
*<proof>*

**lemma** *finsert-fminus-single*[simp]:  $\text{finsert } a (A \text{ |-| } \{a\}) = \text{finsert } a A$   
*<proof>*

**lemma** *finsert-fminus*:  $a \in A \implies \text{finsert } a (A \text{ |-| } \{a\}) = A$   
*<proof>*

**lemma** *fminus-finsert-absorb*:  $x \notin A \implies \text{finsert } x A \text{ |-| } \{x\} = A$   
*<proof>*

**lemma** *fminus-disjoint*[simp]:  $A \text{ |}\cap\text{ } (B \text{ |-| } A) = \{\}$   
*<proof>*

**lemma** *fminus-partition*:  $A \sqsubseteq B \implies A \cup (B \dashv A) = B$   
 ⟨proof⟩

**lemma** *double-fminus*:  $A \sqsubseteq B \implies B \sqsubseteq C \implies B \dashv (C \dashv A) = A$   
 ⟨proof⟩

**lemma** *funion-fminus-cancel[simp]*:  $A \cup (B \dashv A) = A \cup B$   
 ⟨proof⟩

**lemma** *funion-fminus-cancel2[simp]*:  $B \dashv A \cup A = B \cup A$   
 ⟨proof⟩

**lemma** *fminus-funion*:  $A \dashv (B \cup C) = A \dashv B \cap (A \dashv C)$   
 ⟨proof⟩

**lemma** *fminus-finter*:  $A \dashv (B \cap C) = A \dashv B \cup (A \dashv C)$   
 ⟨proof⟩

**lemma** *funion-fminus*:  $A \cup B \dashv C = A \dashv C \cup (B \dashv C)$   
 ⟨proof⟩

**lemma** *finter-fminus*:  $A \cap B \dashv C = A \cap (B \dashv C)$   
 ⟨proof⟩

**lemma** *fminus-finter-distrib*:  $C \cap (A \dashv B) = C \cap A \dashv (C \cap B)$   
 ⟨proof⟩

**lemma** *fminus-finter-distrib2*:  $A \dashv B \cap C = A \cap C \dashv (B \cap C)$   
 ⟨proof⟩

**lemma** *fUNIV-bool[no-atp]*:  $fUNIV = \{False, True\}$   
 ⟨proof⟩

**lemma** *fPow-fempty[simp]*:  $fPow \{\}\} = \{\{\}\}$   
 ⟨proof⟩

**lemma** *fPow-finsert*:  $fPow (finsert a A) = fPow A \cup finsert a \uparrow fPow A$   
 ⟨proof⟩

**lemma** *funion-fPow-fsubset*:  $fPow A \cup fPow B \sqsubseteq fPow (A \cup B)$   
 ⟨proof⟩

**lemma** *fPow-finter-eq[simp]*:  $fPow (A \cap B) = fPow A \cap fPow B$   
 ⟨proof⟩

**lemma** *fset-eq-fsubset*:  $(A = B) = (A \sqsubseteq B \wedge B \sqsubseteq A)$   
 ⟨proof⟩

**lemma** *fsubset-iff[no-atp]*:  $(A \sqsubseteq B) = (\forall t. t \in A \longrightarrow t \in B)$

*<proof>*

**lemma** *fsubset-iff-pfsubset-eq*:  $(A \sqsubseteq B) = (A \subset B \vee A = B)$   
*<proof>*

**lemma** *all-not-fin-conv[simp]*:  $(\forall x. x \notin A) = (A = \{\})$   
*<proof>*

**lemma** *ex-fin-conv*:  $(\exists x. x \in A) = (A \neq \{\})$   
*<proof>*

**lemma** *fimage-mono*:  $A \sqsubseteq B \implies f \upharpoonright A \sqsubseteq f \upharpoonright B$   
*<proof>*

**lemma** *fPow-mono*:  $A \sqsubseteq B \implies fPow A \sqsubseteq fPow B$   
*<proof>*

**lemma** *finsert-mono*:  $C \sqsubseteq D \implies finsert a C \sqsubseteq finsert a D$   
*<proof>*

**lemma** *funion-mono*:  $A \sqsubseteq C \implies B \sqsubseteq D \implies A \cup B \sqsubseteq C \cup D$   
*<proof>*

**lemma** *finter-mono*:  $A \sqsubseteq C \implies B \sqsubseteq D \implies A \cap B \sqsubseteq C \cap D$   
*<proof>*

**lemma** *fminus-mono*:  $A \sqsubseteq C \implies D \sqsubseteq B \implies A - B \sqsubseteq C - D$   
*<proof>*

**lemma** *fin-mono*:  $A \sqsubseteq B \implies x \in A \longrightarrow x \in B$   
*<proof>*

**lemma** *fthe-felem-eq[simp]*:  $fthe-elem \{x\} = x$   
*<proof>*

**lemma** *fLeast-mono*:  
 $mono f \implies fBex S (\lambda x. fBall S ((\leq) x)) \implies (LEAST y. y \in f \upharpoonright S) = f$   
 $(LEAST x. x \in S)$   
*<proof>*

**lemma** *fbind-fbind*:  $fbind (fbind A B) C = fbind A (\lambda x. fbind (B x) C)$   
*<proof>*

**lemma** *fempty-fbind[simp]*:  $fbind \{\} f = \{\}$   
*<proof>*

**lemma** *nonempty-fbind-const*:  $A \neq \{\} \implies fbind A (\lambda-. B) = B$   
*<proof>*

**lemma** *fbind-const*:  $fbind\ A\ (\lambda\cdot.\ B) = (if\ A = \{\|\} \text{ then } \{\|\} \text{ else } B)$   
 ⟨proof⟩

**lemma** *ffmember-filter[simp]*:  $(x \in\| ffilter\ P\ A) = (x \in\| A \wedge P\ x)$   
 ⟨proof⟩

**lemma** *fequalityI*:  $A \subseteq\| B \implies B \subseteq\| A \implies A = B$   
 ⟨proof⟩

**lemma** *fset-of-list-simps[simp]*:  
 $fset\ of\ list\ [] = \{\|\}$   
 $fset\ of\ list\ (x21 \# x22) = finsert\ x21\ (fset\ of\ list\ x22)$   
 ⟨proof⟩

**lemma** *fset-of-list-append[simp]*:  $fset\ of\ list\ (xs \ @\ ys) = fset\ of\ list\ xs \ |\cup|\ fset\ of\ list\ ys$   
 ⟨proof⟩

**lemma** *fset-of-list-rev[simp]*:  $fset\ of\ list\ (rev\ xs) = fset\ of\ list\ xs$   
 ⟨proof⟩

**lemma** *fset-of-list-map[simp]*:  $fset\ of\ list\ (map\ f\ xs) = f \ |^|\ fset\ of\ list\ xs$   
 ⟨proof⟩

## 29.5 Additional lemmas

### 29.5.1 *ffUnion*

**lemma** *ffUnion-union-distrib[simp]*:  $ffUnion\ (A \ |\cup|\ B) = ffUnion\ A \ |\cup|\ ffUnion\ B$   
 ⟨proof⟩

### 29.5.2 *fbind*

**lemma** *fbind-cong[fundef-cong]*:  $A = B \implies (\wedge x. x \in\| B \implies f\ x = g\ x) \implies fbind\ A\ f = fbind\ B\ g$   
 ⟨proof⟩

### 29.5.3 *fsingleton*

**lemma** *fsingletonE*:  $b \in\| \{a\} \implies (b = a \implies thesis) \implies thesis$   
 ⟨proof⟩

### 29.5.4 *fempty*

**lemma** *fempty-ffilter[simp]*:  $ffilter\ (\lambda\cdot.\ False)\ A = \{\|\}$   
 ⟨proof⟩

**lemma** *femptyE [elim!]*:  $a \in\| \{\|\} \implies P$



$\langle proof \rangle$

### 29.5.5 *fset*

**lemma** *fset-simps*[*simp*]:

$fset \{\|\} = \{\}$

$fset (finsert\ x\ X) = insert\ x\ (fset\ X)$

$\langle proof \rangle$

**lemma** *finite-fset* [*simp*]:

**shows** *finite* (*fset* *S*)

$\langle proof \rangle$

**lemmas** *fset-cong* = *fset-inject*

**lemma** *filter-fset* [*simp*]:

**shows**  $fset\ (ffilter\ P\ xs) = Collect\ P\ \cap\ fset\ xs$

$\langle proof \rangle$

**lemma** *inter-fset*[*simp*]:  $fset\ (A\ |\cap|\ B) = fset\ A\ \cap\ fset\ B$

$\langle proof \rangle$

**lemma** *union-fset*[*simp*]:  $fset\ (A\ |\cup|\ B) = fset\ A\ \cup\ fset\ B$

$\langle proof \rangle$

**lemma** *minus-fset*[*simp*]:  $fset\ (A\ |-|\ B) = fset\ A\ -\ fset\ B$

$\langle proof \rangle$

### 29.5.6 *ffilter*

**lemma** *subset-ffilter*:

$ffilter\ P\ A\ |\subseteq|\ ffilter\ Q\ A = (\forall\ x.\ x\ |\in|\ A \longrightarrow P\ x \longrightarrow Q\ x)$

$\langle proof \rangle$

**lemma** *eq-ffilter*:

$(ffilter\ P\ A = ffilter\ Q\ A) = (\forall\ x.\ x\ |\in|\ A \longrightarrow P\ x = Q\ x)$

$\langle proof \rangle$

**lemma** *pfssubset-ffilter*:

$(\bigwedge x.\ x\ |\in|\ A \Longrightarrow P\ x \Longrightarrow Q\ x) \Longrightarrow (x\ |\in|\ A \wedge \neg P\ x \wedge Q\ x) \Longrightarrow$

$ffilter\ P\ A\ |\subset|\ ffilter\ Q\ A$

$\langle proof \rangle$

### 29.5.7 *fset-of-list*

**lemma** *fset-of-list-filter*[*simp*]:

$fset-of-list\ (filter\ P\ xs) = ffilter\ P\ (fset-of-list\ xs)$

$\langle proof \rangle$

**lemma** *fset-of-list-subset*[*intro*]:

$set\ xs \subseteq set\ ys \implies fset\text{-of-list}\ xs \mid\subseteq\ fset\text{-of-list}\ ys$   
 ⟨proof⟩

**lemma** *fset-of-list-elem*:  $(x \mid\in\ fset\text{-of-list}\ xs) \longleftrightarrow (x \in set\ xs)$   
 ⟨proof⟩

### 29.5.8 *finsert*

**lemma** *set-finsert*:

**assumes**  $x \mid\in\ A$

**obtains**  $B$  **where**  $A = finsert\ x\ B$  **and**  $x \mid\notin\ B$

⟨proof⟩

**lemma** *mk-disjoint-finsert*:  $a \mid\in\ A \implies \exists B. A = finsert\ a\ B \wedge a \mid\notin\ B$   
 ⟨proof⟩

**lemma** *finsert-eq-iff*:

**assumes**  $a \mid\notin\ A$  **and**  $b \mid\notin\ B$

**shows**  $(finsert\ a\ A = finsert\ b\ B) =$

$(if\ a = b\ then\ A = B\ else\ \exists C. A = finsert\ b\ C \wedge b \mid\notin\ C \wedge B = finsert\ a\ C \wedge a \mid\notin\ C)$

⟨proof⟩

### 29.5.9 *fimage*

**lemma** *subset-fimage-iff*:  $(B \mid\subseteq\ f\mid^{\uparrow}A) = (\exists AA. AA \mid\subseteq\ A \wedge B = f\mid^{\uparrow}AA)$   
 ⟨proof⟩

**lemma** *fimage-strict-mono*:

**assumes** *inj-on*  $f$  (*fset*  $B$ ) **and**  $A \mid\subset\ B$

**shows**  $f\mid^{\uparrow}A \mid\subset\ f\mid^{\uparrow}B$

— TODO: Configure transfer framework to lift  $\llbracket inj\text{-on}\ ?f\ ?B; ?A \subset ?B \rrbracket \implies ?f\mid^{\uparrow} ?A \subset ?f\mid^{\uparrow} ?B$ .

⟨proof⟩

### 29.5.10 bounded quantification

**lemma** *box-simps* [*simp*, *no-atp*]:

$\bigwedge A\ P\ Q. fBex\ A\ (\lambda x. P\ x \wedge Q) = (fBex\ A\ P \wedge Q)$

$\bigwedge A\ P\ Q. fBex\ A\ (\lambda x. P \wedge Q\ x) = (P \wedge fBex\ A\ Q)$

$\bigwedge P. fBex\ \{\mid\} P = False$

$\bigwedge a\ B\ P. fBex\ (finsert\ a\ B)\ P = (P\ a \vee fBex\ B\ P)$

$\bigwedge A\ P\ f. fBex\ (f\mid^{\uparrow}A)\ P = fBex\ A\ (\lambda x. P\ (f\ x))$

$\bigwedge A\ P. (\neg fBex\ A\ P) = fBall\ A\ (\lambda x. \neg P\ x)$

⟨proof⟩

**lemma** *ball-simps* [*simp*, *no-atp*]:

$\bigwedge A\ P\ Q. fBall\ A\ (\lambda x. P\ x \vee Q) = (fBall\ A\ P \vee Q)$

$\bigwedge A\ P\ Q. fBall\ A\ (\lambda x. P \vee Q\ x) = (P \vee fBall\ A\ Q)$

$\bigwedge A\ P\ Q. fBall\ A\ (\lambda x. P \longrightarrow Q\ x) = (P \longrightarrow fBall\ A\ Q)$

$$\begin{aligned}
& \bigwedge A P Q. fBall A (\lambda x. P x \longrightarrow Q) = (fBex A P \longrightarrow Q) \\
& \bigwedge P. fBall \{\|\} P = True \\
& \bigwedge a B P. fBall (finsert a B) P = (P a \wedge fBall B P) \\
& \bigwedge A P f. fBall (f \mid^{\dagger} A) P = fBall A (\lambda x. P (f x)) \\
& \bigwedge A P. (\neg fBall A P) = fBex A (\lambda x. \neg P x) \\
& \langle proof \rangle
\end{aligned}$$

**lemma** *atomize-fBall*:

$$(\bigwedge x. x \mid \in A \implies P x) \implies Trueprop (fBall A (\lambda x. P x))$$

*<proof>*

**lemma** *fBall-mono[mono]*:  $P \leq Q \implies fBall S P \leq fBall S Q$   
*<proof>*

**lemma** *fBex-mono[mono]*:  $P \leq Q \implies fBex S P \leq fBex S Q$   
*<proof>*

**end**

### 29.5.11 *fcard*

**lemma** *fcard-fempty*:

$$fcard \{\|\} = 0$$

*<proof>*

**lemma** *fcard-finsert-disjoint*:

$$x \mid \notin A \implies fcard (finsert x A) = Suc (fcard A)$$

*<proof>*

**lemma** *fcard-finsert-if*:

$$fcard (finsert x A) = (if x \mid \in A \text{ then } fcard A \text{ else } Suc (fcard A))$$

*<proof>*

**lemma** *fcard-0-eq [simp, no-atp]*:

$$fcard A = 0 \iff A = \{\|\}$$

*<proof>*

**lemma** *fcard-Suc-fminus1*:

$$x \mid \in A \implies Suc (fcard (A \mid - \{|x|\})) = fcard A$$

*<proof>*

**lemma** *fcard-fminus-fsingleton*:

$$x \mid \in A \implies fcard (A \mid - \{|x|\}) = fcard A - 1$$

*<proof>*

**lemma** *fcard-fminus-fsingleton-if*:

$$fcard (A \mid - \{|x|\}) = (if x \mid \in A \text{ then } fcard A - 1 \text{ else } fcard A)$$

*<proof>*

**lemma** *fcard-fminus-finsert[simp]*:

**assumes**  $a \in A$  **and**  $a \notin B$

**shows**  $\text{fcard } (A \text{ |-| } \text{finsert } a B) = \text{fcard } (A \text{ |-| } B) - 1$

*<proof>*

**lemma** *fcard-finsert*:  $\text{fcard } (\text{finsert } x A) = \text{Suc } (\text{fcard } (A \text{ |-| } \{|x\}))$

*<proof>*

**lemma** *fcard-finsert-le*:  $\text{fcard } A \leq \text{fcard } (\text{finsert } x A)$

*<proof>*

**lemma** *fcard-mono*:

$A \subseteq B \implies \text{fcard } A \leq \text{fcard } B$

*<proof>*

**lemma** *fcard-seteq*:  $A \subseteq B \implies \text{fcard } B \leq \text{fcard } A \implies A = B$

*<proof>*

**lemma** *pfssubset-fcard-mono*:  $A \subset B \implies \text{fcard } A < \text{fcard } B$

*<proof>*

**lemma** *fcard-funion-finter*:

$\text{fcard } A + \text{fcard } B = \text{fcard } (A \cup B) + \text{fcard } (A \cap B)$

*<proof>*

**lemma** *fcard-funion-disjoint*:

$A \cap B = \{\} \implies \text{fcard } (A \cup B) = \text{fcard } A + \text{fcard } B$

*<proof>*

**lemma** *fcard-funion-fsubset*:

$B \subseteq A \implies \text{fcard } (A \text{ |-| } B) = \text{fcard } A - \text{fcard } B$

*<proof>*

**lemma** *diff-fcard-le-fcard-fminus*:

$\text{fcard } A - \text{fcard } B \leq \text{fcard } (A \text{ |-| } B)$

*<proof>*

**lemma** *fcard-fminus1-less*:  $x \in A \implies \text{fcard } (A \text{ |-| } \{|x\}) < \text{fcard } A$

*<proof>*

**lemma** *fcard-fminus2-less*:

$x \in A \implies y \in A \implies \text{fcard } (A \text{ |-| } \{|x\} \text{ |-| } \{|y\}) < \text{fcard } A$

*<proof>*

**lemma** *fcard-fminus1-le*:  $\text{fcard } (A \text{ |-| } \{|x\}) \leq \text{fcard } A$

*<proof>*

**lemma** *fcard-pfssubset*:  $A \subseteq B \implies \text{fcard } A < \text{fcard } B \implies A < B$

*<proof>*

**29.5.12** *sorted-list-of-fset*

**lemma** *sorted-list-of-fset-simps*[simp]:  
 $set (sorted-list-of-fset S) = fset S$   
 $fset-of-list (sorted-list-of-fset S) = S$   
 ⟨proof⟩

**29.5.13** *ffold*

**context** *comp-fun-commute*

**begin**

**lemma** *ffold-empty*[simp]:  $ffold f z \{\|\} = z$   
 ⟨proof⟩

**lemma** *ffold-finsert* [simp]:  
**assumes**  $x \notin A$   
**shows**  $ffold f z (finsert x A) = f x (ffold f z A)$   
 ⟨proof⟩

**lemma** *ffold-fun-left-comm*:  
 $f x (ffold f z A) = ffold f (f x z) A$   
 ⟨proof⟩

**lemma** *ffold-finsert2*:  
 $x \notin A \implies ffold f z (finsert x A) = ffold f (f x z) A$   
 ⟨proof⟩

**lemma** *ffold-rec*:  
**assumes**  $x \in A$   
**shows**  $ffold f z A = f x (ffold f z (A -\{x\}))$   
 ⟨proof⟩

**lemma** *ffold-finsert-remove*:  
 $ffold f z (finsert x A) = f x (ffold f z (A -\{x\}))$   
 ⟨proof⟩

**end**

**lemma** *ffold-fimage*:  
**assumes** *inj-on*  $g (fset A)$   
**shows**  $ffold f z (g \upharpoonright A) = ffold (f \circ g) z A$   
 ⟨proof⟩

**lemma** *ffold-cong*:  
**assumes** *comp-fun-commute*  $f$  *comp-fun-commute*  $g$   
 $\bigwedge x. x \in A \implies f x = g x$   
**and**  $s = t$  **and**  $A = B$   
**shows**  $ffold f s A = ffold g t B$   
 ⟨proof⟩

**context** *comp-fun-idem*

**begin**

**lemma** *ffold-finsert-idem*:

$ffold\ f\ z\ (finsert\ x\ A) = f\ x\ (ffold\ f\ z\ A)$   
 ⟨*proof*⟩

**declare** *ffold-finsert* [*simp del*] *ffold-finsert-idem* [*simp*]

**lemma** *ffold-finsert-idem2*:

$ffold\ f\ z\ (finsert\ x\ A) = ffold\ f\ (f\ x\ z)\ A$   
 ⟨*proof*⟩

**end**

### 29.5.14 ( $|\subset|$ )

**lemma** *wfP-pfsubset*: *wfP* ( $|\subset|$ )

⟨*proof*⟩

### 29.5.15 Group operations

**locale** *comm-monoid-fset* = *comm-monoid*

**begin**

**sublocale** *set*: *comm-monoid-set* ⟨*proof*⟩

**lift-definition** *F* :: ( $'b \Rightarrow 'a$ )  $\Rightarrow$   $'b\ fset \Rightarrow 'a\ \mathbf{is}\ set.F$  ⟨*proof*⟩

**lemma** *cong[fundef-cong]*:  $A = B \Longrightarrow (\bigwedge x. x \in | B \Longrightarrow g\ x = h\ x) \Longrightarrow F\ g\ A = F\ h\ B$

⟨*proof*⟩

**lemma** *cong-simp[cong]*:

$\llbracket A = B; \bigwedge x. x \in | B = simp \Rightarrow g\ x = h\ x \rrbracket \Longrightarrow F\ g\ A = F\ h\ B$   
 ⟨*proof*⟩

**end**

**context** *comm-monoid-add* **begin**

**sublocale** *fsum*: *comm-monoid-fset plus 0*

**rewrites** *comm-monoid-set.F plus 0 = sum*

**defines** *fsum* = *fsum.F*

⟨*proof*⟩

**end**

### 29.5.16 Semilattice operations

**locale** *semilattice-fset* = *semilattice*

**begin**

**sublocale** *set*: *semilattice-set*  $\langle$ *proof* $\rangle$

**lift-definition**  $F :: 'a \text{ fset} \Rightarrow 'a \text{ is set.F}$   $\langle$ *proof* $\rangle$

**lemma** *eq-fold*:  $F (\text{finsert } x \ A) = \text{ffold } f \ x \ A$   
 $\langle$ *proof* $\rangle$

**lemma** *singleton* [*simp*]:  $F \{x\} = x$   
 $\langle$ *proof* $\rangle$

**lemma** *insert-not-elem*:  $x \notin A \Longrightarrow A \neq \{\}\Longrightarrow F (\text{finsert } x \ A) = x * F \ A$   
 $\langle$ *proof* $\rangle$

**lemma** *in-idem*:  $x \in A \Longrightarrow x * F \ A = F \ A$   
 $\langle$ *proof* $\rangle$

**lemma** *insert* [*simp*]:  $A \neq \{\}\Longrightarrow F (\text{finsert } x \ A) = x * F \ A$   
 $\langle$ *proof* $\rangle$

**end**

**locale** *semilattice-order-fset* = *binary?*: *semilattice-order* + *semilattice-fset*  
**begin**

**end**

**context** *linorder* **begin**

**sublocale** *fMin*: *semilattice-order-fset* *min* *less-eq* *less*  
**rewrites** *semilattice-set.F* *min* = *Min*  
**defines** *fMin* = *fMin.F*  
 $\langle$ *proof* $\rangle$

**sublocale** *fMax*: *semilattice-order-fset* *max* *greater-eq* *greater*  
**rewrites** *semilattice-set.F* *max* = *Max*  
**defines** *fMax* = *fMax.F*  
 $\langle$ *proof* $\rangle$

**end**

**lemma** *mono-fMax-commute*:  $\text{mono } f \Longrightarrow A \neq \{\}\Longrightarrow f (\text{fMax } A) = \text{fMax } (f \ |' \ A)$   
 $\langle$ *proof* $\rangle$

**lemma** *mono-fMin-commute*:  $\text{mono } f \Longrightarrow A \neq \{\}\Longrightarrow f (\text{fMin } A) = \text{fMin } (f \ |' \ A)$

*<proof>*

**lemma** *fMax-in[simp]*:  $A \neq \{\}\Longrightarrow fMax A \in A$   
*<proof>*

**lemma** *fMin-in[simp]*:  $A \neq \{\}\Longrightarrow fMin A \in A$   
*<proof>*

**lemma** *fMax-ge[simp]*:  $x \in A \Longrightarrow x \leq fMax A$   
*<proof>*

**lemma** *fMin-le[simp]*:  $x \in A \Longrightarrow fMin A \leq x$   
*<proof>*

**lemma** *fMax-eqI*:  $(\bigwedge y. y \in A \Longrightarrow y \leq x) \Longrightarrow x \in A \Longrightarrow fMax A = x$   
*<proof>*

**lemma** *fMin-eqI*:  $(\bigwedge y. y \in A \Longrightarrow x \leq y) \Longrightarrow x \in A \Longrightarrow fMin A = x$   
*<proof>*

**lemma** *fMax-finsert[simp]*:  $fMax (finsert x A) = (if A = \{\} then x else max x (fMax A))$   
*<proof>*

**lemma** *fMin-finsert[simp]*:  $fMin (finsert x A) = (if A = \{\} then x else min x (fMin A))$   
*<proof>*

**context** *linorder begin*

**lemma** *fset-linorder-max-induct[case-names fempty finsert]*:  
**assumes**  $P \{\}$   
**and**  $\bigwedge x S. [\forall y. y \in S \longrightarrow y < x; P S] \Longrightarrow P (finsert x S)$   
**shows**  $P S$   
*<proof>*

**lemma** *fset-linorder-min-induct[case-names fempty finsert]*:  
**assumes**  $P \{\}$   
**and**  $\bigwedge x S. [\forall y. y \in S \longrightarrow y > x; P S] \Longrightarrow P (finsert x S)$   
**shows**  $P S$   
*<proof>*

**end**

## 29.6 Choice in fsets

**lemma** *fset-choice*:  
**assumes**  $\forall x. x \in A \longrightarrow (\exists y. P x y)$   
**shows**  $\exists f. \forall x. x \in A \longrightarrow P x (f x)$



*<proof>*

## 29.7 Induction and Cases rules for fsets

**lemma** *fset-exhaust* [*case-names empty insert, cases type: fset*]:

**assumes** *fempty-case*:  $S = \{\{\}\} \implies P$   
**and** *finsert-case*:  $\bigwedge x S'. S = \text{finsert } x S' \implies P$   
**shows**  $P$   
*<proof>*

**lemma** *fset-induct* [*case-names empty insert*]:

**assumes** *fempty-case*:  $P \{\{\}\}$   
**and** *finsert-case*:  $\bigwedge x S. P S \implies P (\text{finsert } x S)$   
**shows**  $P S$   
*<proof>*

**lemma** *fset-induct-stronger* [*case-names empty insert, induct type: fset*]:

**assumes** *fempty-fset-case*:  $P \{\{\}\}$   
**and** *finsert-fset-case*:  $\bigwedge x S. \llbracket x \notin S; P S \rrbracket \implies P (\text{finsert } x S)$   
**shows**  $P S$   
*<proof>*

**lemma** *fset-card-induct*:

**assumes** *fempty-fset-case*:  $P \{\{\}\}$   
**and** *card-fset-Suc-case*:  $\bigwedge S T. \text{Suc } (\text{fcard } S) = (\text{fcard } T) \implies P S \implies P T$   
**shows**  $P S$   
*<proof>*

**lemma** *fset-strong-cases*:

**obtains**  $xs = \{\{\}\}$   
 |  $ys$   $x \notin ys$  **where**  $x \notin ys$  **and**  $xs = \text{finsert } x ys$   
*<proof>*

**lemma** *fset-induct2*:

$P \{\{\}\} \{\{\}\} \implies$   
 $(\bigwedge x xs. x \notin xs \implies P (\text{finsert } x xs) \{\{\}\}) \implies$   
 $(\bigwedge y ys. y \notin ys \implies P \{\{\}\} (\text{finsert } y ys)) \implies$   
 $(\bigwedge x xs y ys. \llbracket P xs ys; x \notin xs; y \notin ys \rrbracket \implies P (\text{finsert } x xs) (\text{finsert } y ys)) \implies$   
 $P xsa ysa$   
*<proof>*

## 29.8 Lemmas depending on induction

**lemma** *ffUnion-fsubset-iff*:  $\text{ffUnion } A \subseteq B \longleftrightarrow \text{fBall } A (\lambda x. x \subseteq B)$

*<proof>*

## 29.9 Setup for Lifting/Transfer

### 29.9.1 Relator and predicator properties

**lift-definition** *rel-fset* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a fset* ⇒ *'b fset* ⇒ *bool* **is rel-set**  
**parametric** *rel-set-transfer* ⟨*proof*⟩

**lemma** *rel-fset-alt-def*: *rel-fset* *R* = (λ*A B*. (∀*x*. ∃*y*. *x* ∈ *A* → *y* ∈ *B* ∧ *R* *x y*)  
 ∧ (∀*y*. ∃*x*. *y* ∈ *B* → *x* ∈ *A* ∧ *R* *x y*))  
 ⟨*proof*⟩

**lemma** *finite-rel-set*:

**assumes** *fin*: *finite* *X* *finite* *Z*

**assumes** *R-S*: *rel-set* (*R OO S*) *X Z*

**shows** ∃*Y*. *finite* *Y* ∧ *rel-set* *R X Y* ∧ *rel-set* *S Y Z*

⟨*proof*⟩

### 29.9.2 Transfer rules for the Transfer package

Unconditional transfer rules

**context includes** *lifting-syntax*

**begin**

**lemma** *fempty-transfer* [*transfer-rule*]:

*rel-fset* *A* {||} {||}

⟨*proof*⟩

**lemma** *finsert-transfer* [*transfer-rule*]:

(*A* ==> *rel-fset* *A* ==> *rel-fset* *A*) *finsert* *finsert*

⟨*proof*⟩

**lemma** *funion-transfer* [*transfer-rule*]:

(*rel-fset* *A* ==> *rel-fset* *A* ==> *rel-fset* *A*) *funion* *funion*

⟨*proof*⟩

**lemma** *ffUnion-transfer* [*transfer-rule*]:

(*rel-fset* (*rel-fset* *A*) ==> *rel-fset* *A*) *ffUnion* *ffUnion*

⟨*proof*⟩

**lemma** *fimage-transfer* [*transfer-rule*]:

((*A* ==> *B*) ==> *rel-fset* *A* ==> *rel-fset* *B*) *fimage* *fimage*

⟨*proof*⟩

**lemma** *fBall-transfer* [*transfer-rule*]:

(*rel-fset* *A* ==> (*A* ==> (=)) ==> (=)) *fBall* *fBall*

⟨*proof*⟩

**lemma** *fBex-transfer* [*transfer-rule*]:

(*rel-fset* *A* ==> (*A* ==> (=)) ==> (=)) *fBex* *fBex*

⟨*proof*⟩

**lemma** *fPow-transfer* [*transfer-rule*]:  
 $(rel\text{-}fset\ A\ ==\Rightarrow\ rel\text{-}fset\ (rel\text{-}fset\ A))\ fPow\ fPow$   
 $\langle proof \rangle$

**lemma** *rel-fset-transfer* [*transfer-rule*]:  
 $((A\ ==\Rightarrow\ B\ ==\Rightarrow\ (=))\ ==\Rightarrow\ rel\text{-}fset\ A\ ==\Rightarrow\ rel\text{-}fset\ B\ ==\Rightarrow\ (=))$   
 $rel\text{-}fset\ rel\text{-}fset$   
 $\langle proof \rangle$

**lemma** *bind-transfer* [*transfer-rule*]:  
 $(rel\text{-}fset\ A\ ==\Rightarrow\ (A\ ==\Rightarrow\ rel\text{-}fset\ B)\ ==\Rightarrow\ rel\text{-}fset\ B)\ fbind\ fbind$   
 $\langle proof \rangle$

Rules requiring bi-unique, bi-total or right-total relations

**lemma** *fmember-transfer* [*transfer-rule*]:  
**assumes** *bi-unique* *A*  
**shows**  $(A\ ==\Rightarrow\ rel\text{-}fset\ A\ ==\Rightarrow\ (=))\ (|\in|)\ (|\in|)$   
 $\langle proof \rangle$

**lemma** *finter-transfer* [*transfer-rule*]:  
**assumes** *bi-unique* *A*  
**shows**  $(rel\text{-}fset\ A\ ==\Rightarrow\ rel\text{-}fset\ A\ ==\Rightarrow\ rel\text{-}fset\ A)\ finter\ finter$   
 $\langle proof \rangle$

**lemma** *fminus-transfer* [*transfer-rule*]:  
**assumes** *bi-unique* *A*  
**shows**  $(rel\text{-}fset\ A\ ==\Rightarrow\ rel\text{-}fset\ A\ ==\Rightarrow\ rel\text{-}fset\ A)\ (|-)\ (|-)$   
 $\langle proof \rangle$

**lemma** *fsubset-transfer* [*transfer-rule*]:  
**assumes** *bi-unique* *A*  
**shows**  $(rel\text{-}fset\ A\ ==\Rightarrow\ rel\text{-}fset\ A\ ==\Rightarrow\ (=))\ (|\subseteq|)\ (|\subseteq|)$   
 $\langle proof \rangle$

**lemma** *fSup-transfer* [*transfer-rule*]:  
 $bi\text{-}unique\ A\ \Rightarrow\ (rel\text{-}set\ (rel\text{-}fset\ A)\ ==\Rightarrow\ rel\text{-}fset\ A)\ Sup\ Sup$   
 $\langle proof \rangle$

**lemma** *fInf-transfer* [*transfer-rule*]:  
**assumes** *bi-unique* *A* **and** *bi-total* *A*  
**shows**  $(rel\text{-}set\ (rel\text{-}fset\ A)\ ==\Rightarrow\ rel\text{-}fset\ A)\ Inf\ Inf$   
 $\langle proof \rangle$

**lemma** *ffilter-transfer* [*transfer-rule*]:  
**assumes** *bi-unique* *A*

**shows**  $((A \text{ ===== } (=)) \text{ ===== } \text{rel-fset } A \text{ ===== } \text{rel-fset } A) \text{ ffilter ffilter}$   
 ⟨proof⟩

**lemma** *card-transfer* [*transfer-rule*]:  
*bi-unique*  $A \implies (\text{rel-fset } A \text{ ===== } (=)) \text{ fcard fcard}$   
 ⟨proof⟩

**end**

**lifting-update** *fset.lifting*  
**lifting-forget** *fset.lifting*

## 29.10 BNF setup

**context**  
**includes** *fset.lifting*  
**begin**

**lemma** *rel-fset-alt*:  
 $\text{rel-fset } R \ a \ b \longleftrightarrow (\forall t \in \text{fset } a. \exists u \in \text{fset } b. R \ t \ u) \wedge (\forall t \in \text{fset } b. \exists u \in \text{fset } a. R \ u \ t)$   
 ⟨proof⟩

**lemma** *fset-to-fset*:  $\text{finite } A \implies \text{fset } (\text{the-inv fset } A) = A$   
 ⟨proof⟩

**lemma** *rel-fset-aux*:  
 $(\forall t \in \text{fset } a. \exists u \in \text{fset } b. R \ t \ u) \wedge (\forall u \in \text{fset } b. \exists t \in \text{fset } a. R \ t \ u) \longleftrightarrow$   
 $((\text{BNF-Def.Grp } \{a. \text{fset } a \subseteq \{(a, b). R \ a \ b\}\} (\text{fimage fst}))^{-1-1} \text{ OO}$   
 $\text{BNF-Def.Grp } \{a. \text{fset } a \subseteq \{(a, b). R \ a \ b\}\} (\text{fimage snd})) \ a \ b \ (\text{is ?L} = \text{?R})$   
 ⟨proof⟩

**bnf** *'a fset*  
*map*: *fimage*  
*sets*: *fset*  
*bd*: *natLeq*  
*wits*:  $\{\|\}$   
*rel*: *rel-fset*  
 ⟨proof⟩

**lemma** *rel-fset-fset*:  $\text{rel-set } \chi \ (\text{fset } A1) \ (\text{fset } A2) = \text{rel-fset } \chi \ A1 \ A2$   
 ⟨proof⟩

**end**

**declare**  
*fset.map-comp*[*simp*]  
*fset.map-id*[*simp*]  
*fset.set-map*[*simp*]

### 29.11 Size setup

**context includes** *fset.lifting*

**begin**

**lift-definition** *size-fset* :: ('a ⇒ nat) ⇒ 'a fset ⇒ nat **is** λf. sum (Suc ∘ f) ⟨proof⟩

**end**

**instantiation** *fset* :: (type) size

**begin**

**definition** *size-fset* **where**

*size-fset-overloaded-def*: *size-fset* = FSet.size-fset (λ-. 0)

**instance** ⟨proof⟩

**end**

**lemma** *size-fset-simps*[*simp*]: *size-fset* f X = (∑ x ∈ fset X. Suc (f x))  
 ⟨proof⟩

**lemma** *size-fset-overloaded-simps*[*simp*]: *size* X = (∑ x ∈ fset X. Suc 0)  
 ⟨proof⟩

**lemma** *fset-size-o-map*: *inj* f ⇒ *size-fset* g ∘ *fimage* f = *size-fset* (g ∘ f)  
 ⟨proof⟩

⟨ML⟩

**lifting-update** *fset.lifting*

**lifting-forget** *fset.lifting*

### 29.12 Advanced relator customization

Set vs. sum relators:

**lemma** *rel-set-rel-sum*[*simp*]:

*rel-set* (rel-sum χ φ) A1 A2 ↔

*rel-set* χ (Inl -' A1) (Inl -' A2) ∧ *rel-set* φ (Inr -' A1) (Inr -' A2)

(**is** ?L ↔ ?Rl ∧ ?Rr)

⟨proof⟩

#### 29.12.1 Countability

**lemma** *exists-fset-of-list*: ∃ xs. *fset-of-list* xs = S

**including** *fset.lifting*

⟨proof⟩

**lemma** *fset-of-list-surj*[*simp*, *intro*]: *surj* *fset-of-list*

⟨proof⟩

**instance** *fset* :: (countable) countable

⟨proof⟩

**29.13 Quickcheck setup**

Setup adapted from sets.

**notation** *Quickcheck-Exhaustive.orelse* (**infixr**  $\langle orelse \rangle$  55)

**context**

**includes** *term-syntax*

**begin**

**definition** [*code-unfold*]:

*valterm-femptyset* = *Code-Evaluation.valtermify* ( $\{\|\}$  :: ('a :: typerep) fset)

**definition** [*code-unfold*]:

*valtermify-finsert* x s = *Code-Evaluation.valtermify finsert*  $\{\cdot\}$  (x :: ('a :: typerep \* -))  $\{\cdot\}$  s

**end**

**instantiation** *fset* :: (*exhaustive*) *exhaustive*

**begin**

**fun** *exhaustive-fset* **where**

*exhaustive-fset* f i = (if i = 0 then None else (f  $\{\|\}$  orelse *exhaustive-fset* ( $\lambda A. f A$  orelse *Quickcheck-Exhaustive.exhaustive* ( $\lambda x. \text{if } x \in A \text{ then None else } f (\text{finsert } x A)$ ) (i - 1)) (i - 1))))

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

**end**

**instantiation** *fset* :: (*full-exhaustive*) *full-exhaustive*

**begin**

**fun** *full-exhaustive-fset* **where**

*full-exhaustive-fset* f i = (if i = 0 then None else (f *valterm-femptyset* orelse *full-exhaustive-fset* ( $\lambda A. f A$  orelse *Quickcheck-Exhaustive.full-exhaustive* ( $\lambda x. \text{if } x \in A \text{ then None else } f (\text{valtermify-finsert } x A)$ ) (i - 1)) (i - 1))))

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

**end**

**no-notation** *Quickcheck-Exhaustive.orelse* (**infixr**  $\langle orelse \rangle$  55)

**instantiation** *fset* :: (*random*) *random*

**begin**

**context**

**includes** *state-combinator-syntax*

**begin**

```
fun random-aux-fset :: natural  $\Rightarrow$  natural  $\Rightarrow$  natural  $\times$  natural  $\Rightarrow$  ('a fset  $\times$  (unit
 $\Rightarrow$  term))  $\times$  natural  $\times$  natural where
random-aux-fset 0 j = Quickcheck-Random.collapse (Random.select-weight [(1, Pair
valterm-femptyset)]) |
random-aux-fset (Code-Numeral.Suc i) j =
  Quickcheck-Random.collapse (Random.select-weight
    [(1, Pair valterm-femptyset),
     (Code-Numeral.Suc i,
      Quickcheck-Random.random j  $\circ\rightarrow$  ( $\lambda x$ . random-aux-fset i j  $\circ\rightarrow$  ( $\lambda s$ . Pair
(valtermify-finsert x s))))))])
```

**lemma** [code]:

```
random-aux-fset i j =
  Quickcheck-Random.collapse (Random.select-weight [(1, Pair valterm-femptyset),
    (i, Quickcheck-Random.random j  $\circ\rightarrow$  ( $\lambda x$ . random-aux-fset (i - 1) j  $\circ\rightarrow$  ( $\lambda s$ .
Pair (valtermify-finsert x s)))))])
<proof>
```

**definition** random-fset i = random-aux-fset i i

**instance** <proof>

**end**

**end**

## 29.14 Code Generation Setup

The following *code-unfold* lemmas are so the pre-processor of the code generator will perform conversions like, e.g.,  $(x \in | f | \uparrow \text{fset-of-list } xs) = (x \in f \text{ 'set } xs)$ .

**declare**

```
ffilter.rep-eq[code-unfold]
fimage.rep-eq[code-unfold]
finsert.rep-eq[code-unfold]
fset-of-list.rep-eq[code-unfold]
inf-fset.rep-eq[code-unfold]
minus-fset.rep-eq[code-unfold]
sup-fset.rep-eq[code-unfold]
uminus-fset.rep-eq[code-unfold]
```

**end**

### 30 Type of finite maps defined as a subtype of maps

```

theory Finite-Map
  imports FSet AList Conditional-Parametricity
  abbrevs (= =  $\subseteq_f$ )
begin

```

#### 30.1 Auxiliary constants and lemmas over *map*

```

parametric-constant map-add-transfer[transfer-rule]: map-add-def
parametric-constant map-of-transfer[transfer-rule]: map-of-def

```

```

context includes lifting-syntax begin

```

```

abbreviation rel-map :: ('b  $\Rightarrow$  'c  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'c)  $\Rightarrow$  bool where
rel-map f  $\equiv$  (=)  $\implies$  rel-option f

```

```

lemma ran-transfer[transfer-rule]: (rel-map A  $\implies$  rel-set A) ran ran
<proof>

```

```

lemma ran-alt-def: ran m = (the  $\circ$  m) ' dom m
<proof>

```

```

parametric-constant dom-transfer[transfer-rule]: dom-def

```

```

definition map-upd :: 'a  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b) where
map-upd k v m = m(k  $\mapsto$  v)

```

```

parametric-constant map-upd-transfer[transfer-rule]: map-upd-def

```

```

definition map-filter :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b) where
map-filter P m = ( $\lambda x$ . if P x then m x else None)

```

```

parametric-constant map-filter-transfer[transfer-rule]: map-filter-def

```

```

lemma map-filter-map-of[simp]: map-filter P (map-of m) = map-of [(k, -)  $\leftarrow$  m.
P k]
<proof>

```

```

lemma map-filter-finite[intro]:
  assumes finite (dom m)
  shows finite (dom (map-filter P m))
<proof>

```

```

definition map-drop :: 'a  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b) where
map-drop a = map-filter ( $\lambda a'$ . a'  $\neq$  a)

```

```

parametric-constant map-drop-transfer[transfer-rule]: map-drop-def

```



**definition** *map-drop-set* :: 'a set  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b) **where**  
*map-drop-set* A = *map-filter* ( $\lambda a. a \notin A$ )

**parametric-constant** *map-drop-set-transfer*[*transfer-rule*]: *map-drop-set-def*

**definition** *map-restrict-set* :: 'a set  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'b) **where**  
*map-restrict-set* A = *map-filter* ( $\lambda a. a \in A$ )

**parametric-constant** *map-restrict-set-transfer*[*transfer-rule*]: *map-restrict-set-def*

**definition** *map-pred* :: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  bool **where**  
*map-pred* P m  $\longleftrightarrow$  ( $\forall x. \text{case } m \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } y \Rightarrow P \ x \ y$ )

**parametric-constant** *map-pred-transfer*[*transfer-rule*]: *map-pred-def*

**definition** *rel-map-on-set* :: 'a set  $\Rightarrow$  ('b  $\Rightarrow$  'c  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\rightarrow$  'c)  $\Rightarrow$  bool **where**  
*rel-map-on-set* S P = *eq-onp* ( $\lambda x. x \in S$ )  $\implies$  *rel-option* P

**definition** *set-of-map* :: ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a  $\times$  'b) set **where**  
*set-of-map* m = {(k, v) | k v. m k = Some v}

**lemma** *set-of-map-alt-def*: *set-of-map* m = ( $\lambda k. (k, \text{the } (m \ k))$ ) ‘ *dom* m  
 <proof>

**lemma** *set-of-map-finite*: *finite* (*dom* m)  $\implies$  *finite* (*set-of-map* m)  
 <proof>

**lemma** *set-of-map-inj*: *inj* *set-of-map*  
 <proof>

**lemma** *dom-comp*: *dom* (m  $\circ_m$  n)  $\subseteq$  *dom* n  
 <proof>

**lemma** *dom-comp-finite*: *finite* (*dom* n)  $\implies$  *finite* (*dom* (*map-comp* m n))  
 <proof>

**parametric-constant** *map-comp-transfer*[*transfer-rule*]: *map-comp-def*

**end**

## 30.2 Abstract characterisation

**typedef** ('a, 'b) *fmap* = {m. *finite* (*dom* m)} :: ('a  $\rightarrow$  'b) set  
**morphisms** *fmlookup* *Abs-fmap*  
 <proof>

**setup-lifting** *type-definition-fmap*

**lemma** *dom-fmlookup-finite*[*intro, simp*]: *finite (dom (fmlookup m))*  
 ⟨*proof*⟩

**lemma** *fmap-ext*:  
**assumes**  $\bigwedge x. \text{fmlookup } m \ x = \text{fmlookup } n \ x$   
**shows**  $m = n$   
 ⟨*proof*⟩

### 30.3 Operations

**context**  
**includes** *fset.lifting*  
**begin**

**lift-definition** *fmran* :: (*'a, 'b*) *fmap*  $\Rightarrow$  *'b fset*  
**is** *ran*  
**parametric** *ran-transfer*  
 ⟨*proof*⟩

**lemma** *fmlookup-ran-iff*:  $y \in | \text{fmran } m \iff (\exists x. \text{fmlookup } m \ x = \text{Some } y)$   
 ⟨*proof*⟩

**lemma** *fmranI*:  $\text{fmlookup } m \ x = \text{Some } y \implies y \in | \text{fmran } m$  ⟨*proof*⟩

**lemma** *fmranE*[*elim*]:  
**assumes**  $y \in | \text{fmran } m$   
**obtains**  $x$  **where**  $\text{fmlookup } m \ x = \text{Some } y$   
 ⟨*proof*⟩

**lift-definition** *fmdom* :: (*'a, 'b*) *fmap*  $\Rightarrow$  *'a fset*  
**is** *dom*  
**parametric** *dom-transfer*  
 ⟨*proof*⟩

**lemma** *fmlookup-dom-iff*:  $x \in | \text{fmdom } m \iff (\exists a. \text{fmlookup } m \ x = \text{Some } a)$   
 ⟨*proof*⟩

**lemma** *fmdom-notI*:  $\text{fmlookup } m \ x = \text{None} \implies x \notin | \text{fmdom } m$  ⟨*proof*⟩

**lemma** *fmdomI*:  $\text{fmlookup } m \ x = \text{Some } y \implies x \in | \text{fmdom } m$  ⟨*proof*⟩

**lemma** *fmdom-notD*[*dest*]:  $x \notin | \text{fmdom } m \implies \text{fmlookup } m \ x = \text{None}$  ⟨*proof*⟩

**lemma** *fmdomE*[*elim*]:  
**assumes**  $x \in | \text{fmdom } m$   
**obtains**  $y$  **where**  $\text{fmlookup } m \ x = \text{Some } y$   
 ⟨*proof*⟩

**lift-definition** *fmdom'* :: (*'a, 'b*) *fmap*  $\Rightarrow$  *'a set*  
**is** *dom*

**parametric dom-transfer**  
 ⟨proof⟩

**lemma** *fmlookup-dom'-iff*:  $x \in \text{fndom}' m \iff (\exists a. \text{fmlookup } m \ x = \text{Some } a)$   
 ⟨proof⟩

**lemma** *fndom'-notI*:  $\text{fmlookup } m \ x = \text{None} \implies x \notin \text{fndom}' m$  ⟨proof⟩

**lemma** *fndom'I*:  $\text{fmlookup } m \ x = \text{Some } y \implies x \in \text{fndom}' m$  ⟨proof⟩

**lemma** *fndom'-notD[dest]*:  $x \notin \text{fndom}' m \implies \text{fmlookup } m \ x = \text{None}$  ⟨proof⟩

**lemma** *fndom'E[elim]*:  
 assumes  $x \in \text{fndom}' m$   
 obtains  $x \ y$  where  $\text{fmlookup } m \ x = \text{Some } y$   
 ⟨proof⟩

**lemma** *fndom'-alt-def*:  $\text{fndom}' m = \text{fset } (\text{fndom } m)$   
 ⟨proof⟩

**lemma** *finite-fndom'[simp]*:  $\text{finite } (\text{fndom}' m)$   
 ⟨proof⟩

**lemma** *dom-fmlookup[simp]*:  $\text{dom } (\text{fmlookup } m) = \text{fndom}' m$   
 ⟨proof⟩

**lift-definition** *fmempty* ::  $('a, 'b) \text{ fmap}$   
 is *Map.empty*  
 ⟨proof⟩

**lemma** *fmempty-lookup[simp]*:  $\text{fmlookup } \text{fmempty } x = \text{None}$   
 ⟨proof⟩

**lemma** *fndom-empty[simp]*:  $\text{fndom } \text{fmempty} = \{\}\ \langle \text{proof} \rangle$

**lemma** *fndom'-empty[simp]*:  $\text{fndom}' \text{fmempty} = \{\}\ \langle \text{proof} \rangle$

**lemma** *fmran-empty[simp]*:  $\text{fmran } \text{fmempty} = \text{fempty}$  ⟨proof⟩

**lift-definition** *fmupd* ::  $'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ fmap} \Rightarrow ('a, 'b) \text{ fmap}$   
 is *map-upd*  
**parametric** *map-upd-transfer*  
 ⟨proof⟩

**lemma** *fmupd-lookup[simp]*:  $\text{fmlookup } (\text{fmupd } a \ b \ m) \ a' = (\text{if } a = a' \text{ then } \text{Some } b \text{ else } \text{fmlookup } m \ a')$   
 ⟨proof⟩

**lemma** *fndom-fmupd[simp]*:  $\text{fndom } (\text{fmupd } a \ b \ m) = \text{finsert } a \ (\text{fndom } m)$  ⟨proof⟩

**lemma** *fndom'-fmupd[simp]*:  $\text{fndom}' (\text{fmupd } a \ b \ m) = \text{insert } a \ (\text{fndom}' m)$  ⟨proof⟩

**lemma** *fmupd-reorder-neq*:  
 assumes  $a \neq b$

**shows**  $fmupd\ a\ x\ (fmupd\ b\ y\ m) = fmupd\ b\ y\ (fmupd\ a\ x\ m)$   
 ⟨proof⟩

**lemma**  $fmupd-idem[simp]$ :  $fmupd\ a\ x\ (fmupd\ a\ y\ m) = fmupd\ a\ x\ m$   
 ⟨proof⟩

**lift-definition**  $fmfilter :: ('a \Rightarrow bool) \Rightarrow ('a, 'b)\ fmap \Rightarrow ('a, 'b)\ fmap$   
**is** *map-filter*  
**parametric** *map-filter-transfer*  
 ⟨proof⟩

**lemma**  $fmdom-filter[simp]$ :  $fmdom\ (fmfilter\ P\ m) = ffilter\ P\ (fmdom\ m)$   
 ⟨proof⟩

**lemma**  $fmdom'-filter[simp]$ :  $fmdom'\ (fmfilter\ P\ m) = Set.filter\ P\ (fmdom'\ m)$   
 ⟨proof⟩

**lemma**  $fmlookup-filter[simp]$ :  $fmlookup\ (fmfilter\ P\ m)\ x = (if\ P\ x\ then\ fmlookup\ m\ x\ else\ None)$   
 ⟨proof⟩

**lemma**  $fmfilter-empty[simp]$ :  $fmfilter\ P\ fmempty = fmempty$   
 ⟨proof⟩

**lemma**  $fmfilter-true[simp]$ :  
**assumes**  $\bigwedge x\ y. fmlookup\ m\ x = Some\ y \implies P\ x$   
**shows**  $fmfilter\ P\ m = m$   
 ⟨proof⟩

**lemma**  $fmfilter-false[simp]$ :  
**assumes**  $\bigwedge x\ y. fmlookup\ m\ x = Some\ y \implies \neg P\ x$   
**shows**  $fmfilter\ P\ m = fmempty$   
 ⟨proof⟩

**lemma**  $fmfilter-comp[simp]$ :  $fmfilter\ P\ (fmfilter\ Q\ m) = fmfilter\ (\lambda x. P\ x \wedge Q\ x)\ m$   
 ⟨proof⟩

**lemma**  $fmfilter-comm$ :  $fmfilter\ P\ (fmfilter\ Q\ m) = fmfilter\ Q\ (fmfilter\ P\ m)$   
 ⟨proof⟩

**lemma**  $fmfilter-cong[cong]$ :  
**assumes**  $\bigwedge x\ y. fmlookup\ m\ x = Some\ y \implies P\ x = Q\ x$   
**shows**  $fmfilter\ P\ m = fmfilter\ Q\ m$   
 ⟨proof⟩

**lemma**  $fmfilter-cong'[fundef-cong]$ :  
**assumes**  $m = n \wedge x. x \in fmdom'\ m \implies P\ x = Q\ x$   
**shows**  $fmfilter\ P\ m = fmfilter\ Q\ n$

*<proof>*

**lemma** *fmfilter-upd[simp]*:

*fmfilter P (fmupd x y m) = (if P x then fmupd x y (fmfilter P m) else fmfilter P m)*

*<proof>*

**lift-definition** *fmdrop* :: 'a  $\Rightarrow$  ('a, 'b) *fmap*  $\Rightarrow$  ('a, 'b) *fmap*

**is** *map-drop*

**parametric** *map-drop-transfer*

*<proof>*

**lemma** *fmdrop-lookup[simp]*: *fmlookup (fmdrop a m) a = None*

*<proof>*

**lift-definition** *fmdrop-set* :: 'a *set*  $\Rightarrow$  ('a, 'b) *fmap*  $\Rightarrow$  ('a, 'b) *fmap*

**is** *map-drop-set*

**parametric** *map-drop-set-transfer*

*<proof>*

**lift-definition** *fmdrop-fset* :: 'a *fset*  $\Rightarrow$  ('a, 'b) *fmap*  $\Rightarrow$  ('a, 'b) *fmap*

**is** *map-drop-set*

**parametric** *map-drop-set-transfer*

*<proof>*

**lift-definition** *fmrestrict-set* :: 'a *set*  $\Rightarrow$  ('a, 'b) *fmap*  $\Rightarrow$  ('a, 'b) *fmap*

**is** *map-restrict-set*

**parametric** *map-restrict-set-transfer*

*<proof>*

**lift-definition** *fmrestrict-fset* :: 'a *fset*  $\Rightarrow$  ('a, 'b) *fmap*  $\Rightarrow$  ('a, 'b) *fmap*

**is** *map-restrict-set*

**parametric** *map-restrict-set-transfer*

*<proof>*

**lemma** *fmfilter-alt-defs*:

*fmdrop a = fmfilter ( $\lambda a'. a' \neq a$ )*

*fmdrop-set A = fmfilter ( $\lambda a. a \notin A$ )*

*fmdrop-fset B = fmfilter ( $\lambda a. a \notin B$ )*

*fmrestrict-set A = fmfilter ( $\lambda a. a \in A$ )*

*fmrestrict-fset B = fmfilter ( $\lambda a. a \in B$ )*

*<proof>*

**lemma** *fmdom-drop[simp]*: *fmdom (fmdrop a m) = fmdom m - {a}* *<proof>*

**lemma** *fmdom'-drop[simp]*: *fmdom' (fmdrop a m) = fmdom' m - {a}* *<proof>*

**lemma** *fmdom'-drop-set[simp]*: *fmdom' (fmdrop-set A m) = fmdom' m - A* *<proof>*

**lemma** *fmdom-drop-fset[simp]*: *fmdom (fmdrop-fset A m) = fmdom m - A* *<proof>*

**lemma** *fmdom'-restrict-set*: *fmdom' (fmrestrict-set A m)  $\subseteq$  A* *<proof>*

**lemma** *fmdom-restrict-fset*: *fmdom (fmrestrict-fset A m)  $\subseteq$  A* *<proof>*

**lemma** *fmdrop-fmupd*:  $fmdrop\ x\ (fmupd\ y\ z\ m) = (if\ x = y\ then\ fmdrop\ x\ m\ else\ fmupd\ y\ z\ (fmdrop\ x\ m))$   
 ⟨proof⟩

**lemma** *fmdrop-idle*:  $x \notin fmdom\ B \implies fmdrop\ x\ B = B$   
 ⟨proof⟩

**lemma** *fmdrop-idle'*:  $x \notin fmdom'\ B \implies fmdrop\ x\ B = B$   
 ⟨proof⟩

**lemma** *fmdrop-fmupd-same*:  $fmdrop\ x\ (fmupd\ x\ y\ m) = fmdrop\ x\ m$   
 ⟨proof⟩

**lemma** *fmdom'-restrict-set-precise*:  $fmdom'\ (fmrestrict\ set\ A\ m) = fmdom'\ m \cap A$   
 ⟨proof⟩

**lemma** *fmdom'-restrict-fset-precise*:  $fmdom\ (fmrestrict\ fset\ A\ m) = fmdom\ m \upharpoonright A$   
 ⟨proof⟩

**lemma** *fmdom'-drop-fset[simp]*:  $fmdom'\ (fmdrop\ fset\ A\ m) = fmdom'\ m - fset\ A$   
 ⟨proof⟩

**lemma** *fmdom'-restrict-fset*:  $fmdom'\ (fmrestrict\ fset\ A\ m) \subseteq fset\ A$   
 ⟨proof⟩

**lemma** *fmlookup-drop[simp]*:  
 $fmlookup\ (fmdrop\ a\ m)\ x = (if\ x \neq a\ then\ fmlookup\ m\ x\ else\ None)$   
 ⟨proof⟩

**lemma** *fmlookup-drop-set[simp]*:  
 $fmlookup\ (fmdrop\ set\ A\ m)\ x = (if\ x \notin A\ then\ fmlookup\ m\ x\ else\ None)$   
 ⟨proof⟩

**lemma** *fmlookup-drop-fset[simp]*:  
 $fmlookup\ (fmdrop\ fset\ A\ m)\ x = (if\ x \notin A\ then\ fmlookup\ m\ x\ else\ None)$   
 ⟨proof⟩

**lemma** *fmlookup-restrict-set[simp]*:  
 $fmlookup\ (fmrestrict\ set\ A\ m)\ x = (if\ x \in A\ then\ fmlookup\ m\ x\ else\ None)$   
 ⟨proof⟩

**lemma** *fmlookup-restrict-fset[simp]*:  
 $fmlookup\ (fmrestrict\ fset\ A\ m)\ x = (if\ x \in A\ then\ fmlookup\ m\ x\ else\ None)$   
 ⟨proof⟩

**lemma** *fmrestrict-set-dom[simp]*:  $fmrestrict\ set\ (fmdom'\ m)\ m = m$

*<proof>*

**lemma** *fmrestrict-fset-dom[simp]: fmrestrict-fset (fmdom m) m = m*  
*<proof>*

**lemma** *fmdrop-empty[simp]: fmdrop a fmempty = fmempty*  
*<proof>*

**lemma** *fmdrop-set-empty[simp]: fmdrop-set A fmempty = fmempty*  
*<proof>*

**lemma** *fmdrop-fset-empty[simp]: fmdrop-fset A fmempty = fmempty*  
*<proof>*

**lemma** *fmdrop-fset-fmdom[simp]: fmdrop-fset (fmdom A) A = fmempty*  
*<proof>*

**lemma** *fmdrop-set-fmdom[simp]: fmdrop-set (fmdom' A) A = fmempty*  
*<proof>*

**lemma** *fmrestrict-set-empty[simp]: fmrestrict-set A fmempty = fmempty*  
*<proof>*

**lemma** *fmrestrict-fset-empty[simp]: fmrestrict-fset A fmempty = fmempty*  
*<proof>*

**lemma** *fmdrop-set-null[simp]: fmdrop-set {} m = m*  
*<proof>*

**lemma** *fmdrop-fset-null[simp]: fmdrop-fset {} m = m*  
*<proof>*

**lemma** *fmdrop-set-single[simp]: fmdrop-set {a} m = fmdrop a m*  
*<proof>*

**lemma** *fmdrop-fset-single[simp]: fmdrop-fset {|a|} m = fmdrop a m*  
*<proof>*

**lemma** *fmrestrict-set-null[simp]: fmrestrict-set {} m = fmempty*  
*<proof>*

**lemma** *fmrestrict-fset-null[simp]: fmrestrict-fset {} m = fmempty*  
*<proof>*

**lemma** *fmdrop-comm: fmdrop a (fmdrop b m) = fmdrop b (fmdrop a m)*  
*<proof>*

**lemma** *fmdrop-set-insert[simp]: fmdrop-set (insert x S) m = fmdrop x (fmdrop-set S m)*

*<proof>*

**lemma** *fmdrop-fset-insert[simp]*:  $fmdrop\text{-}fset\ (finsert\ x\ S)\ m = fmdrop\ x\ (fmdrop\text{-}fset\ S\ m)$

*<proof>*

**lemma** *fmrestrict-set-twice[simp]*:  $fmrestrict\text{-}set\ S\ (fmrestrict\text{-}set\ T\ m) = fmrestrict\text{-}set\ (S\ \cap\ T)\ m$

*<proof>*

**lemma** *fmrestrict-fset-twice[simp]*:  $fmrestrict\text{-}fset\ S\ (fmrestrict\text{-}fset\ T\ m) = fmrestrict\text{-}fset\ (S\ |\cap|\ T)\ m$

*<proof>*

**lemma** *fmrestrict-set-drop[simp]*:  $fmrestrict\text{-}set\ S\ (fmdrop\ b\ m) = fmrestrict\text{-}set\ (S\ -\ \{b\})\ m$

*<proof>*

**lemma** *fmrestrict-fset-drop[simp]*:  $fmrestrict\text{-}fset\ S\ (fmdrop\ b\ m) = fmrestrict\text{-}fset\ (S\ -\ \{|b|\})\ m$

*<proof>*

**lemma** *fmdrop-fmrestrict-set[simp]*:  $fmdrop\ b\ (fmrestrict\text{-}set\ S\ m) = fmrestrict\text{-}set\ (S\ -\ \{b\})\ m$

*<proof>*

**lemma** *fmdrop-fmrestrict-fset[simp]*:  $fmdrop\ b\ (fmrestrict\text{-}fset\ S\ m) = fmrestrict\text{-}fset\ (S\ -\ \{|b|\})\ m$

*<proof>*

**lemma** *fmdrop-idem[simp]*:  $fmdrop\ a\ (fmdrop\ a\ m) = fmdrop\ a\ m$

*<proof>*

**lemma** *fmdrop-set-twice[simp]*:  $fmdrop\text{-}set\ S\ (fmdrop\text{-}set\ T\ m) = fmdrop\text{-}set\ (S\ \cup\ T)\ m$

*<proof>*

**lemma** *fmdrop-fset-twice[simp]*:  $fmdrop\text{-}fset\ S\ (fmdrop\text{-}fset\ T\ m) = fmdrop\text{-}fset\ (S\ |\cup|\ T)\ m$

*<proof>*

**lemma** *fmdrop-set-fmdrop[simp]*:  $fmdrop\text{-}set\ S\ (fmdrop\ b\ m) = fmdrop\text{-}set\ (insert\ b\ S)\ m$

*<proof>*

**lemma** *fmdrop-fset-fmdrop[simp]*:  $fmdrop\text{-}fset\ S\ (fmdrop\ b\ m) = fmdrop\text{-}fset\ (finsert\ b\ S)\ m$

*<proof>*



**lift-definition**  $fmadd :: ('a, 'b) fmap \Rightarrow ('a, 'b) fmap \Rightarrow ('a, 'b) fmap$  (**infixl**  $\langle ++_f \rangle$  100)  
**is** *map-add*  
**parametric** *map-add-transfer*  
 $\langle proof \rangle$

**lemma** *fmlookup-add[simp]*:  
 $fmlookup (m ++_f n) x = (if x \in |fndom\ n\ then\ fmlookup\ n\ x\ else\ fmlookup\ m\ x)$   
 $\langle proof \rangle$

**lemma** *fmdom-add[simp]*:  $fmdom (m ++_f n) = fmdom\ m \cup |fmdom\ n$   $\langle proof \rangle$   
**lemma** *fmdom'-add[simp]*:  $fmdom' (m ++_f n) = fmdom'\ m \cup fmdom'\ n$   $\langle proof \rangle$

**lemma** *fmadd-drop-left-dom*:  $fmdrop-fset (fmdom\ n) m ++_f n = m ++_f n$   
 $\langle proof \rangle$

**lemma** *fmadd-restrict-right-dom*:  $fmrestrict-fset (fmdom\ n) (m ++_f n) = n$   
 $\langle proof \rangle$

**lemma** *fmfilter-add-distrib[simp]*:  $fmfilter\ P (m ++_f n) = fmfilter\ P\ m ++_f fmfilter\ P\ n$   
 $\langle proof \rangle$

**lemma** *fmdrop-add-distrib[simp]*:  $fmdrop\ a (m ++_f n) = fmdrop\ a\ m ++_f fmdrop\ a\ n$   
 $\langle proof \rangle$

**lemma** *fmdrop-set-add-distrib[simp]*:  $fmdrop-set\ A (m ++_f n) = fmdrop-set\ A\ m ++_f fmdrop-set\ A\ n$   
 $\langle proof \rangle$

**lemma** *fmdrop-fset-add-distrib[simp]*:  $fmdrop-fset\ A (m ++_f n) = fmdrop-fset\ A\ m ++_f fmdrop-fset\ A\ n$   
 $\langle proof \rangle$

**lemma** *fmrestrict-set-add-distrib[simp]*:  
 $fmrestrict-set\ A (m ++_f n) = fmrestrict-set\ A\ m ++_f fmrestrict-set\ A\ n$   
 $\langle proof \rangle$

**lemma** *fmrestrict-fset-add-distrib[simp]*:  
 $fmrestrict-fset\ A (m ++_f n) = fmrestrict-fset\ A\ m ++_f fmrestrict-fset\ A\ n$   
 $\langle proof \rangle$

**lemma** *fmadd-empty[simp]*:  $fmempty ++_f m = m ++_f fmempty = m$   
 $\langle proof \rangle$

**lemma** *fmadd-idempotent[simp]*:  $m ++_f m = m$   
 $\langle proof \rangle$

**lemma** *fmadd-assoc[simp]*:  $m \text{ ++}_f (n \text{ ++}_f p) = m \text{ ++}_f n \text{ ++}_f p$   
 ⟨proof⟩

**lemma** *fmadd-fmupd[simp]*:  $m \text{ ++}_f \text{fmupd } a \ b \ n = \text{fmupd } a \ b \ (m \text{ ++}_f n)$   
 ⟨proof⟩

**lift-definition** *fmpred* ::  $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a, 'b) \text{ fmap} \Rightarrow \text{bool}$   
**is** *map-pred*  
**parametric** *map-pred-transfer*  
 ⟨proof⟩

**lemma** *fmpredI[intro]*:  
**assumes**  $\bigwedge x \ y. \text{fmlookup } m \ x = \text{Some } y \Longrightarrow P \ x \ y$   
**shows** *fmpred*  $P \ m$   
 ⟨proof⟩

**lemma** *fmpredD[dest]*: *fmpred*  $P \ m \Longrightarrow \text{fmlookup } m \ x = \text{Some } y \Longrightarrow P \ x \ y$   
 ⟨proof⟩

**lemma** *fmpred-iff*: *fmpred*  $P \ m \longleftrightarrow (\forall x \ y. \text{fmlookup } m \ x = \text{Some } y \longrightarrow P \ x \ y)$   
 ⟨proof⟩

**lemma** *fmpred-alt-def*: *fmpred*  $P \ m \longleftrightarrow \text{fBall } (\text{fdom } m) (\lambda x. P \ x \ (\text{the } (\text{fmlookup } m \ x)))$   
 ⟨proof⟩

**lemma** *fmpred-mono-strong*:  
**assumes**  $\bigwedge x \ y. \text{fmlookup } m \ x = \text{Some } y \Longrightarrow P \ x \ y \Longrightarrow Q \ x \ y$   
**shows** *fmpred*  $P \ m \Longrightarrow \text{fmpred } Q \ m$   
 ⟨proof⟩

**lemma** *fmpred-mono[mono]*:  $P \leq Q \Longrightarrow \text{fmpred } P \leq \text{fmpred } Q$   
 ⟨proof⟩

**lemma** *fmpred-empty[intro!, simp]*: *fmpred*  $P \ \text{fmempty}$   
 ⟨proof⟩

**lemma** *fmpred-upd[intro]*: *fmpred*  $P \ m \Longrightarrow P \ x \ y \Longrightarrow \text{fmpred } P \ (\text{fmupd } x \ y \ m)$   
 ⟨proof⟩

**lemma** *fmpred-updD[dest]*: *fmpred*  $P \ (\text{fmupd } x \ y \ m) \Longrightarrow P \ x \ y$   
 ⟨proof⟩

**lemma** *fmpred-add[intro]*: *fmpred*  $P \ m \Longrightarrow \text{fmpred } P \ n \Longrightarrow \text{fmpred } P \ (m \text{ ++}_f n)$   
 ⟨proof⟩

**lemma** *fmpred-filter[intro]*: *fmpred*  $P \ m \Longrightarrow \text{fmpred } P \ (\text{fmfilter } Q \ m)$   
 ⟨proof⟩

**lemma** *fmpred-drop*[intro]:  $fmpred P m \implies fmpred P (fmdrop a m)$   
 ⟨proof⟩

**lemma** *fmpred-drop-set*[intro]:  $fmpred P m \implies fmpred P (fmdrop-set A m)$   
 ⟨proof⟩

**lemma** *fmpred-drop-fset*[intro]:  $fmpred P m \implies fmpred P (fmdrop-fset A m)$   
 ⟨proof⟩

**lemma** *fmpred-restrict-set*[intro]:  $fmpred P m \implies fmpred P (fmrestrict-set A m)$   
 ⟨proof⟩

**lemma** *fmpred-restrict-fset*[intro]:  $fmpred P m \implies fmpred P (fmrestrict-fset A m)$   
 ⟨proof⟩

**lemma** *fmpred-cases*[consumes 1]:  
 assumes  $fmpred P m$   
 obtains  $(none) fmlookup m x = None \mid (some) y$  **where**  $fmlookup m x = Some y P x y$   
 ⟨proof⟩

**lift-definition** *fmsubset* ::  $('a, 'b) fmap \Rightarrow ('a, 'b) fmap \Rightarrow bool$  (**infix**  $\subseteq_f$  50)  
 is *map-le*  
 ⟨proof⟩

**lemma** *fmsubset-alt-def*:  $m \subseteq_f n \iff fmpred (\lambda k v. fmlookup n k = Some v) m$   
 ⟨proof⟩

**lemma** *fmsubset-pred*:  $fmpred P m \implies n \subseteq_f m \implies fmpred P n$   
 ⟨proof⟩

**lemma** *fmsubset-filter-mono*:  $m \subseteq_f n \implies fmfiter P m \subseteq_f fmfiter P n$   
 ⟨proof⟩

**lemma** *fmsubset-drop-mono*:  $m \subseteq_f n \implies fmdrop a m \subseteq_f fmdrop a n$   
 ⟨proof⟩

**lemma** *fmsubset-drop-set-mono*:  $m \subseteq_f n \implies fmdrop-set A m \subseteq_f fmdrop-set A n$   
 ⟨proof⟩

**lemma** *fmsubset-drop-fset-mono*:  $m \subseteq_f n \implies fmdrop-fset A m \subseteq_f fmdrop-fset A n$   
 ⟨proof⟩

**lemma** *fmsubset-restrict-set-mono*:  $m \subseteq_f n \implies fmrestrict-set A m \subseteq_f fmrestrict-set A n$   
 ⟨proof⟩

**lemma** *fmsubset-restrict-fset-mono*:  $m \subseteq_f n \implies \text{fmrestrict-fset } A \ m \subseteq_f \text{fmrestrict-fset } A \ n$

*<proof>*

**lemma** *fmfilter-subset[simp]*:  $\text{fmfilter } P \ m \subseteq_f m$

*<proof>*

**lemma** *fmsubset-drop[simp]*:  $\text{fmdrop } a \ m \subseteq_f m$

*<proof>*

**lemma** *fmsubset-drop-set[simp]*:  $\text{fmdrop-set } S \ m \subseteq_f m$

*<proof>*

**lemma** *fmsubset-drop-fset[simp]*:  $\text{fmdrop-fset } S \ m \subseteq_f m$

*<proof>*

**lemma** *fmsubset-restrict-set[simp]*:  $\text{fmrestrict-set } S \ m \subseteq_f m$

*<proof>*

**lemma** *fmsubset-restrict-fset[simp]*:  $\text{fmrestrict-fset } S \ m \subseteq_f m$

*<proof>*

**lift-definition** *fset-of-fmap* ::  $('a, 'b) \text{ fmap} \Rightarrow ('a \times 'b) \text{ fset}$  **is** *set-of-map*

*<proof>*

**lemma** *fset-of-fmap-inj[intro, simp]*:  $\text{inj fset-of-fmap}$

*<proof>*

**lemma** *fset-of-fmap-iff[simp]*:  $(a, b) \in | \text{fset-of-fmap } m \iff \text{fmlookup } m \ a = \text{Some } b$

*<proof>*

**lemma** *fset-of-fmap-iff'*:  $(a, b) \in \text{fset } (\text{fset-of-fmap } m) \iff \text{fmlookup } m \ a = \text{Some } b$

*<proof>*

**lift-definition** *fmap-of-list* ::  $('a \times 'b) \text{ list} \Rightarrow ('a, 'b) \text{ fmap}$

**is** *map-of*

**parametric** *map-of-transfer*

*<proof>*

**lemma** *fmap-of-list-simps[simp]*:

$\text{fmap-of-list } [] = \text{fmempty}$

$\text{fmap-of-list } ((k, v) \# kvs) = \text{fmupd } k \ v \ (\text{fmap-of-list } kvs)$

*<proof>*

**lemma** *fmap-of-list-app[simp]*:  $\text{fmap-of-list } (xs @ ys) = \text{fmap-of-list } ys \ ++_f \ \text{fmap-of-list } xs$

*<proof>*

**lemma** *fmupd-alt-def*:  $fmupd\ k\ v\ m = m\ ++_f\ fmap\ of\ list\ [(k,\ v)]$   
 ⟨proof⟩

**lemma** *fmpred-of-list[intro]*:  
**assumes**  $\bigwedge k\ v.\ (k,\ v) \in set\ xs \implies P\ k\ v$   
**shows**  $fmpred\ P\ (fmap\ of\ list\ xs)$   
 ⟨proof⟩

**lemma** *fmap-of-list-SomeD*:  $fmlookup\ (fmap\ of\ list\ xs)\ k = Some\ v \implies (k,\ v) \in set\ xs$   
 ⟨proof⟩

**lemma** *fmdom-fmap-of-list[simp]*:  $fmdom\ (fmap\ of\ list\ xs) = fset\ of\ list\ (map\ fst\ xs)$   
 ⟨proof⟩

**lift-definition** *fmrel-on-fset* ::  $'a\ fset \Rightarrow ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow ('a,\ 'b)\ fmap \Rightarrow ('a,\ 'c)\ fmap \Rightarrow bool$   
**is** *rel-map-on-set*  
 ⟨proof⟩

**lemma** *fmrel-on-fset-alt-def*:  $fmrel\ on\ fset\ S\ P\ m\ n \longleftrightarrow fBall\ S\ (\lambda x.\ rel\ option\ P\ (fmlookup\ m\ x)\ (fmlookup\ n\ x))$   
 ⟨proof⟩

**lemma** *fmrel-on-fsetI[intro]*:  
**assumes**  $\bigwedge x.\ x \in S \implies rel\ option\ P\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)$   
**shows**  $fmrel\ on\ fset\ S\ P\ m\ n$   
 ⟨proof⟩

**lemma** *fmrel-on-fset-mono[mono]*:  $R \leq Q \implies fmrel\ on\ fset\ S\ R \leq fmrel\ on\ fset\ S\ Q$   
 ⟨proof⟩

**lemma** *fmrel-on-fsetD*:  $x \in S \implies fmrel\ on\ fset\ S\ P\ m\ n \implies rel\ option\ P\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)$   
 ⟨proof⟩

**lemma** *fmrel-on-fsubset*:  $fmrel\ on\ fset\ S\ R\ m\ n \implies T \subseteq S \implies fmrel\ on\ fset\ T\ R\ m\ n$   
 ⟨proof⟩

**lemma** *fmrel-on-fset-unionI*:  
 $fmrel\ on\ fset\ A\ R\ m\ n \implies fmrel\ on\ fset\ B\ R\ m\ n \implies fmrel\ on\ fset\ (A \cup B)\ R\ m\ n$   
 ⟨proof⟩

**lemma** *fmrel-on-fset-updateI*:

**assumes**  $fmrel\text{-}on\text{-}fset\ S\ P\ m\ n\ P\ v_1\ v_2$   
**shows**  $fmrel\text{-}on\text{-}fset\ (finsert\ k\ S)\ P\ (fmupd\ k\ v_1\ m)\ (fmupd\ k\ v_2\ n)$   
 ⟨proof⟩

**lift-definition**  $fmimage :: ('a, 'b)\ fmap \Rightarrow 'a\ fset \Rightarrow 'b\ fset$  **is**  $\lambda m\ S.\ \{b \mid a\ b.\ m\ a = Some\ b \wedge a \in S\}$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}alt\text{-}def: fmimage\ m\ S = fmran\ (fmrestrict\text{-}fset\ S\ m)$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}empty[simp]: fmimage\ m\ fempty = fempty$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}subset\text{-}ran[simp]: fmimage\ m\ S \subseteq fmran\ m$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}dom[simp]: fmimage\ m\ (fmdom\ m) = fmran\ m$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}inter: fmimage\ m\ (A \mid \cap \mid B) \subseteq fmimage\ m\ A \mid \cap \mid fmimage\ m\ B$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}inter\text{-}dom[simp]:$   
 $fmimage\ m\ (fmdom\ m \mid \cap \mid A) = fmimage\ m\ A$   
 $fmimage\ m\ (A \mid \cap \mid fmdom\ m) = fmimage\ m\ A$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}union[simp]: fmimage\ m\ (A \mid \cup \mid B) = fmimage\ m\ A \mid \cup \mid fmimage\ m\ B$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}Union[simp]: fmimage\ m\ (ffUnion\ A) = ffUnion\ (fmimage\ m \mid 'A)$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}filter[simp]: fmimage\ (fmfilter\ P\ m)\ A = fmimage\ m\ (ffilter\ P\ A)$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}drop[simp]: fmimage\ (fmdrop\ a\ m)\ A = fmimage\ m\ (A - \{a\})$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}drop\text{-}fset[simp]: fmimage\ (fmdrop\text{-}fset\ B\ m)\ A = fmimage\ m\ (A - B)$   
 ⟨proof⟩

**lemma**  $fmimage\text{-}restrict\text{-}fset[simp]: fmimage\ (fmrestrict\text{-}fset\ B\ m)\ A = fmimage\ m\ (A \mid \cap \mid B)$   
 ⟨proof⟩

**lemma** *fmfilter-ran[simp]*:  $fmran (fmfilter P m) = fmimage m (ffilter P (fndom m))$   
 ⟨proof⟩

**lemma** *fmran-drop[simp]*:  $fmran (fndrop a m) = fmimage m (fndom m - \{a\})$   
 ⟨proof⟩

**lemma** *fmran-drop-fset[simp]*:  $fmran (fndrop-fset A m) = fmimage m (fndom m - A)$   
 ⟨proof⟩

**lemma** *fmran-restrict-fset*:  $fmran (fnrestrict-fset A m) = fmimage m (fndom m \cap A)$   
 ⟨proof⟩

**lemma** *fmlookup-image-iff*:  $y \in | fmimage m A \iff (\exists x. fmlookup m x = Some y \wedge x \in | A)$   
 ⟨proof⟩

**lemma** *fmimageI*:  $fmlookup m x = Some y \implies x \in | A \implies y \in | fmimage m A$   
 ⟨proof⟩

**lemma** *fmimageE[elim]*:  
 assumes  $y \in | fmimage m A$   
 obtains  $x$  where  $fmlookup m x = Some y$   $x \in | A$   
 ⟨proof⟩

**lift-definition** *fmcomp* ::  $('b, 'c) fmap \Rightarrow ('a, 'b) fmap \Rightarrow ('a, 'c) fmap$  (**infixl**  $\langle \circ_f \rangle$  55)  
 is *map-comp*  
 parametric *map-comp-transfer*  
 ⟨proof⟩

**lemma** *fmlookup-comp[simp]*:  $fmlookup (m \circ_f n) x = Option.bind (fmlookup n x) (fmlookup m)$   
 ⟨proof⟩

**end**

### 30.4 BNF setup

**lift-bnf**  $('a, fmran': 'b) fmap$  [*wits*: *Map.empty*]  
 for *map*: *fmmmap*  
       *rel*: *fmrel*  
 ⟨proof⟩

**declare** *fmap.pred-mono*[*mono*]

**lemma** *fmran'-alt-def*:  $fmran' m = fset (fmran m)$   
**including** *fset.lifting*  
 ⟨*proof*⟩

**lemma** *fmlookup-ran'-iff*:  $y \in fmran' m \longleftrightarrow (\exists x. fmlookup m x = Some y)$   
 ⟨*proof*⟩

**lemma** *fmran'I*:  $fmlookup m x = Some y \implies y \in fmran' m$   
 ⟨*proof*⟩

**lemma** *fmran'E[elim]*:  
**assumes**  $y \in fmran' m$   
**obtains**  $x$  **where**  $fmlookup m x = Some y$   
 ⟨*proof*⟩

**lemma** *fmrel-iff*:  $fmrel R m n \longleftrightarrow (\forall x. rel-option R (fmlookup m x) (fmlookup n x))$   
 ⟨*proof*⟩

**lemma** *fmrelI[intro]*:  
**assumes**  $\bigwedge x. rel-option R (fmlookup m x) (fmlookup n x)$   
**shows**  $fmrel R m n$   
 ⟨*proof*⟩

**lemma** *fmrel-upd[intro]*:  $fmrel P m n \implies P x y \implies fmrel P (fmupd k x m) (fmupd k y n)$   
 ⟨*proof*⟩

**lemma** *fmrelD[dest]*:  $fmrel P m n \implies rel-option P (fmlookup m x) (fmlookup n x)$   
 ⟨*proof*⟩

**lemma** *fmrel-addI[intro]*:  
**assumes**  $fmrel P m n$   $fmrel P a b$   
**shows**  $fmrel P (m ++_f a) (n ++_f b)$   
 ⟨*proof*⟩

**lemma** *fmrel-cases[consumes 1]*:  
**assumes**  $fmrel P m n$   
**obtains**  $(none) fmlookup m x = None fmlookup n x = None$   
 |  $(some) a b$  **where**  $fmlookup m x = Some a fmlookup n x = Some b P a b$   
 ⟨*proof*⟩

**lemma** *fmrel-filter[intro]*:  $fmrel P m n \implies fmrel P (fmfilter Q m) (fmfilter Q n)$   
 ⟨*proof*⟩

**lemma** *fmrel-drop[intro]*:  $fmrel P m n \implies fmrel P (fmdrop a m) (fmdrop a n)$   
 ⟨*proof*⟩



**lemma** *fmrel-drop-set[intro]*:  $fmrel\ P\ m\ n \implies fmrel\ P\ (fmdrop\text{-}set\ A\ m)\ (fmdrop\text{-}set\ A\ n)$   
 ⟨proof⟩

**lemma** *fmrel-drop-fset[intro]*:  $fmrel\ P\ m\ n \implies fmrel\ P\ (fmdrop\text{-}fset\ A\ m)\ (fmdrop\text{-}fset\ A\ n)$   
 ⟨proof⟩

**lemma** *fmrel-restrict-set[intro]*:  $fmrel\ P\ m\ n \implies fmrel\ P\ (fmrestrict\text{-}set\ A\ m)\ (fmrestrict\text{-}set\ A\ n)$   
 ⟨proof⟩

**lemma** *fmrel-restrict-fset[intro]*:  $fmrel\ P\ m\ n \implies fmrel\ P\ (fmrestrict\text{-}fset\ A\ m)\ (fmrestrict\text{-}fset\ A\ n)$   
 ⟨proof⟩

**lemma** *fmrel-on-fset-fmrel-restrict*:  
 $fmrel\text{-}on\text{-}fset\ S\ P\ m\ n \iff fmrel\ P\ (fmrestrict\text{-}fset\ S\ m)\ (fmrestrict\text{-}fset\ S\ n)$   
 ⟨proof⟩

**lemma** *fmrel-on-fset-refl-strong*:  
 assumes  $\bigwedge x\ y. x \in S \implies fmlookup\ m\ x = Some\ y \implies P\ y\ y$   
 shows  $fmrel\text{-}on\text{-}fset\ S\ P\ m\ m$   
 ⟨proof⟩

**lemma** *fmrel-on-fset-addI*:  
 assumes  $fmrel\text{-}on\text{-}fset\ S\ P\ m\ n\ fmrel\text{-}on\text{-}fset\ S\ P\ a\ b$   
 shows  $fmrel\text{-}on\text{-}fset\ S\ P\ (m\ ++_f\ a)\ (n\ ++_f\ b)$   
 ⟨proof⟩

**lemma** *fmrel-fmdom-eq*:  
 assumes  $fmrel\ P\ x\ y$   
 shows  $fmdom\ x = fmdom\ y$   
 ⟨proof⟩

**lemma** *fmrel-fmdom'-eq*:  $fmrel\ P\ x\ y \implies fmdom'\ x = fmdom'\ y$   
 ⟨proof⟩

**lemma** *fmrel-rel-fmran*:  
 assumes  $fmrel\ P\ x\ y$   
 shows  $rel\text{-}fset\ P\ (fmran\ x)\ (fmran\ y)$   
 ⟨proof⟩

**lemma** *fmrel-rel-fmran'*:  $fmrel\ P\ x\ y \implies rel\text{-}set\ P\ (fmran'\ x)\ (fmran'\ y)$   
 ⟨proof⟩

**lemma** *pred-fmap-fmpred[simp]*:  $pred\text{-}fmap\ P = fmpred\ (\lambda\_. P)$   
 ⟨proof⟩

**lemma** *pred-fmap-id[simp]*:  $\text{pred-fmap id (fmmap f m)} \longleftrightarrow \text{pred-fmap f m}$   
 ⟨proof⟩

**lemma** *pred-fmapD*:  $\text{pred-fmap P m} \implies x \in | \text{fmran m} \implies P x$   
 ⟨proof⟩

**lemma** *fmlookup-map[simp]*:  $\text{fmlookup (fmmap f m) x} = \text{map-option f (fmlookup m x)}$   
 ⟨proof⟩

**lemma** *fmprered-map[simp]*:  $\text{fmprered P (fmmap f m)} \longleftrightarrow \text{fmprered } (\lambda k v. P k (f v)) m$   
 ⟨proof⟩

**lemma** *fmprered-id[simp]*:  $\text{fmprered } (\lambda-. \text{id}) \text{ (fmmap f m)} \longleftrightarrow \text{fmprered } (\lambda-. f) m$   
 ⟨proof⟩

**lemma** *fmmap-add[simp]*:  $\text{fmmap f (m ++_f n)} = \text{fmmap f m ++_f fmmap f n}$   
 ⟨proof⟩

**lemma** *fmmap-empty[simp]*:  $\text{fmmap f fmempty} = \text{fmempty}$   
 ⟨proof⟩

**lemma** *fmdom-map[simp]*:  $\text{fmdom (fmmap f m)} = \text{fmdom m}$   
**including** *fset.lifting*  
 ⟨proof⟩

**lemma** *fmdom'-map[simp]*:  $\text{fmdom}' \text{ (fmmap f m)} = \text{fmdom}' m$   
 ⟨proof⟩

**lemma** *fmran-fmmap[simp]*:  $\text{fmran (fmmap f m)} = f \mid | \text{fmran m}$   
**including** *fset.lifting*  
 ⟨proof⟩

**lemma** *fmran'-fmmap[simp]*:  $\text{fmran}' \text{ (fmmap f m)} = f \text{ ' } \text{fmran}' m$   
 ⟨proof⟩

**lemma** *fmfilter-fmmap[simp]*:  $\text{fmfilter P (fmmap f m)} = \text{fmmap f (fmfilter P m)}$   
 ⟨proof⟩

**lemma** *fmdrop-fmmap[simp]*:  $\text{fmdrop a (fmmap f m)} = \text{fmmap f (fmdrop a m)}$   
 ⟨proof⟩

**lemma** *fmdrop-set-fmmap[simp]*:  $\text{fmdrop-set A (fmmap f m)} = \text{fmmap f (fmdrop-set A m)}$   
 ⟨proof⟩

**lemma** *fmdrop-fset-fmmap[simp]*:  $\text{fmdrop-fset A (fmmap f m)} = \text{fmmap f (fmdrop-fset A m)}$   
 ⟨proof⟩

**lemma** *fmrestrict-set-fmmap[simp]*:  $fmrestrict\text{-}set\ A\ (fmmap\ f\ m) = fmmap\ f\ (fmrestrict\text{-}set\ A\ m)$   
 ⟨proof⟩

**lemma** *fmrestrict-fset-fmmap[simp]*:  $fmrestrict\text{-}fset\ A\ (fmmap\ f\ m) = fmmap\ f\ (fmrestrict\text{-}fset\ A\ m)$   
 ⟨proof⟩

**lemma** *fmmap-subset[intro]*:  $m \subseteq_f n \implies fmmap\ f\ m \subseteq_f fmmap\ f\ n$   
 ⟨proof⟩

**lemma** *fmmap-fset-of-fmap*:  $fset\text{-}of\text{-}fmap\ (fmmap\ f\ m) = (\lambda(k, v). (k, f\ v)) \upharpoonright\ fset\text{-}of\text{-}fmap\ m$   
**including** *fset.lifting*  
 ⟨proof⟩

**lemma** *fmmap-fmupd*:  $fmmap\ f\ (fmupd\ x\ y\ m) = fmupd\ x\ (f\ y)\ (fmmap\ f\ m)$   
 ⟨proof⟩

### 30.5 size setup

**definition** *size-fmap* ::  $('a \Rightarrow nat) \Rightarrow ('b \Rightarrow nat) \Rightarrow ('a, 'b)\ fmap \Rightarrow nat$  **where**  
*[simp]*:  $size\text{-}fmap\ f\ g\ m = size\text{-}fset\ (\lambda(a, b). f\ a + g\ b)\ (fset\text{-}of\text{-}fmap\ m)$

**instantiation** *fmap* ::  $(type, type)\ size$  **begin**

**definition** *size-fmap where*

*size-fmap-overloaded-def*:  $size\text{-}fmap = Finite\text{-}Map.size\text{-}fmap\ (\lambda-. 0)\ (\lambda-. 0)$

**instance** ⟨proof⟩

**end**

**lemma** *size-fmap-overloaded-simps[simp]*:  $size\ x = size\ (fset\text{-}of\text{-}fmap\ x)$   
 ⟨proof⟩

**lemma** *fmap-size-o-map*:  $size\text{-}fmap\ f\ g \circ fmmap\ h = size\text{-}fmap\ f\ (g \circ h)$   
 ⟨proof⟩

⟨ML⟩

### 30.6 Additional operations

**lift-definition** *fmmap-keys* ::  $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b)\ fmap \Rightarrow ('a, 'c)\ fmap$  **is**  
 $\lambda f\ m\ a. map\text{-}option\ (f\ a)\ (m\ a)$   
 ⟨proof⟩

**lemma** *fmpruned-fmmap-keys[simp]*:  $fmpruned\ P\ (fmmap\text{-}keys\ f\ m) = fmpruned\ (\lambda a\ b. P\ a\ (f\ a\ b))\ m$

*<proof>*

**lemma** *fmdom-fmmap-keys[simp]*:  $fmdom (fmmap-keys f m) = fmdom m$   
**including** *fset.lifting*  
*<proof>*

**lemma** *fmlookup-fmmap-keys[simp]*:  $fmlookup (fmmap-keys f m) x = map-option (f x) (fmlookup m x)$   
*<proof>*

**lemma** *fmfilter-fmmap-keys[simp]*:  $fmfilter P (fmmap-keys f m) = fmmap-keys f (fmfilter P m)$   
*<proof>*

**lemma** *fmdrop-fmmap-keys[simp]*:  $fmdrop a (fmmap-keys f m) = fmmap-keys f (fmdrop a m)$   
*<proof>*

**lemma** *fmdrop-set-fmmap-keys[simp]*:  $fmdrop-set A (fmmap-keys f m) = fmmap-keys f (fmdrop-set A m)$   
*<proof>*

**lemma** *fmdrop-fset-fmmap-keys[simp]*:  $fmdrop-fset A (fmmap-keys f m) = fmmap-keys f (fmdrop-fset A m)$   
*<proof>*

**lemma** *fmrestrict-set-fmmap-keys[simp]*:  $fmrestrict-set A (fmmap-keys f m) = fmmap-keys f (fmrestrict-set A m)$   
*<proof>*

**lemma** *fmrestrict-fset-fmmap-keys[simp]*:  $fmrestrict-fset A (fmmap-keys f m) = fmmap-keys f (fmrestrict-fset A m)$   
*<proof>*

**lemma** *fmmap-keys-subset[intro]*:  $m \subseteq_f n \implies fmmap-keys f m \subseteq_f fmmap-keys f n$   
*<proof>*

**definition** *sorted-list-of-fmap* ::  $('a::linorder, 'b) fmap \Rightarrow ('a \times 'b) list$  **where**  
*sorted-list-of-fmap m = map ( $\lambda k. (k, the (fmlookup m k))$ ) (sorted-list-of-fset (fmdom m))*

**lemma** *list-all-sorted-list[simp]*:  $list-all P (sorted-list-of-fmap m) = fmpred (curry P) m$   
*<proof>*

**lemma** *map-of-sorted-list[simp]*:  $map-of (sorted-list-of-fmap m) = fmlookup m$   
*<proof>*  
**including** *fset.lifting*

*<proof>*

### 30.7 Additional properties

**lemma** *fmchoice'*:  
**assumes** *finite S*  $\forall x \in S. \exists y. Q x y$   
**shows**  $\exists m. \text{fmdom}' m = S \wedge \text{fmpred } Q m$   
*<proof>*

### 30.8 Lifting/transfer setup

**context includes** *lifting-syntax* **begin**

**lemma** *fmempty-transfer*[*simp, intro, transfer-rule*]: *fmrel P fmempty fmempty*  
*<proof>*

**lemma** *fmadd-transfer*[*transfer-rule*]:  
 $(\text{fmrel } P \implies \text{fmrel } P \implies \text{fmrel } P) \text{ fmadd fmadd}$   
*<proof>*

**lemma** *fmupd-transfer*[*transfer-rule*]:  
 $((=) \implies P \implies \text{fmrel } P \implies \text{fmrel } P) \text{ fmupd fmupd}$   
*<proof>*

**end**

**lemma** *Quotient-fmap-bnf*[*quot-map*]:  
**assumes** *Quotient R Abs Rep T*  
**shows** *Quotient (fmrel R) (fmap Abs) (fmap Rep) (fmrel T)*  
*<proof>*

### 30.9 View as datatype

**lemma** *fmap-distinct*[*simp*]:  
 $\text{fmempty} \neq \text{fmupd } k v m$   
 $\text{fmupd } k v m \neq \text{fmempty}$   
*<proof>*

**lifting-update** *fmap.lifting*

**lemma** *fmap-exhaust*[*cases type: fmap*]:  
**obtains**  $(\text{fmempty}) m = \text{fmempty}$   
 $| (\text{fmupd}) x y m' \textbf{ where } m = \text{fmupd } x y m' x \notin \text{fmdom } m'$   
*<proof>* **including** *fmap.lifting* **and** *fset.lifting*  
*<proof>*

**lemma** *fmap-induct*[*case-names fmempty fmupd, induct type: fmap*]:  
**assumes**  $P \text{ fmempty}$   
**assumes**  $(\bigwedge x y m. P m \implies \text{fmllookup } m x = \text{None} \implies P (\text{fmupd } x y m))$   
**shows**  $P m$

⟨proof⟩

### 30.10 Code setup

**instantiation** *fmap* :: (*type*, *equal*) *equal* **begin**

**definition** *equal-fmap* ≡ *fmrel HOL.equal*

**instance** ⟨*proof*⟩

**end**

**lemma** *fBall-alt-def*:  $fBall\ S\ P \longleftrightarrow (\forall x. x \in S \longrightarrow P\ x)$   
 ⟨*proof*⟩

**lemma** *fmrel-code*:

$fmrel\ R\ m\ n \longleftrightarrow$   
 $fBall\ (fmdom\ m)\ (\lambda x. rel-option\ R\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)) \wedge$   
 $fBall\ (fmdom\ n)\ (\lambda x. rel-option\ R\ (fmlookup\ m\ x)\ (fmlookup\ n\ x))$   
 ⟨*proof*⟩

**lemmas** [*code*] =

*fmrel-code*  
*fmran'-alt-def*  
*fmdom'-alt-def*  
*fmfilter-alt-defs*  
*pred-fmap-fmpred*  
*fmsubset-alt-def*  
*fmupd-alt-def*  
*fmrel-on-fset-alt-def*  
*fmpred-alt-def*

**code-datatype** *fmap-of-list*

**quickcheck-generator** *fmap* *constructors*: *fmap-of-list*

**context** **includes** *fset.lifting* **begin**

**lemma** *fmlookup-of-list*[*code*]:  $fmlookup\ (fmap-of-list\ m) = map-of\ m$   
 ⟨*proof*⟩

**lemma** *fmempty-of-list*[*code*]:  $fmempty = fmap-of-list\ []$   
 ⟨*proof*⟩

**lemma** *fmran-of-list*[*code*]:  $fmran\ (fmap-of-list\ m) = snd\ |\cdot| fset-of-list\ (AList.clearjunk\ m)$   
 ⟨*proof*⟩

**lemma** *fmdom-of-list*[*code*]:  $fmdom\ (fmap-of-list\ m) = fst\ |\cdot| fset-of-list\ m$

*<proof>*

**lemma** *fmfilter-of-list*[code]: *fmfilter*  $P$  (*fmap-of-list*  $m$ ) = *fmap-of-list* (*filter* ( $\lambda(k, -). P\ k$ )  $m$ )  
*<proof>*

**lemma** *fmadd-of-list*[code]: *fmap-of-list*  $m$  ++<sub>*f*</sub> *fmap-of-list*  $n$  = *fmap-of-list* (*AList.merge*  $m$   $n$ )  
*<proof>*

**lemma** *fmmmap-of-list*[code]: *fmmmap*  $f$  (*fmap-of-list*  $m$ ) = *fmap-of-list* (*map* (*apsnd*  $f$ )  $m$ )  
*<proof>*

**lemma** *fmmmap-keys-of-list*[code]:  
*fmmmap-keys*  $f$  (*fmap-of-list*  $m$ ) = *fmap-of-list* (*map* ( $\lambda(a, b). (a, f\ a\ b)$ )  $m$ )  
*<proof>*

**lemma** *fmimage-of-list*[code]:  
*fmimage* (*fmap-of-list*  $m$ )  $A$  = *fset-of-list* (*map snd* (*filter* ( $\lambda(k, -). k \in A$ ) (*AList.clearjunk*  $m$ )))  
*<proof>*

**lemma** *fmcomp-list*[code]:  
*fmap-of-list*  $m$   $\circ_f$  *fmap-of-list*  $n$  = *fmap-of-list* (*AList.compose*  $n$   $m$ )  
*<proof>*

**end**

### 30.11 Instances

**lemma** *exists-map-of*:  
**assumes** *finite* (*dom*  $m$ ) **shows**  $\exists xs. \text{map-of } xs = m$   
*<proof>*

**lemma** *exists-fmap-of-list*:  $\exists xs. \text{fmap-of-list } xs = m$   
*<proof>*

**lemma** *fmap-of-list-surj*[*simp, intro*]: *surj* *fmap-of-list*  
*<proof>*

**instance** *fmap* :: (*countable, countable*) *countable*  
*<proof>*

**instance** *fmap* :: (*finite, finite*) *finite*  
*<proof>*

**lifting-update** *fmap.lifting*

**lifting-forget** *fmap.lifting*

### 30.12 Tests

#### export-code

```
Ball fset fmrel fmran fmran' fndom fndom' fmpred pred-fmap fmsubset fmupd
fmrel-on-fset
  fmdrop fmdrop-set fmdrop-fset fmrestrict-set fmrestrict-fset fmimage fmlookup
fmempty
  fmfilter fmadd fmmmap fmmmap-keys fmcomp
  checking SML Scala Haskell? OCaml?
```

— *lifting* through *fmap*

#### experiment begin

**context includes** *fset.lifting* **begin**

**lift-definition** *test1* :: ('a, 'b fset) fmap **is** fmempty :: ('a, 'b set) fmap  
 ⟨*proof*⟩

**lift-definition** *test2* :: 'a ⇒ 'b ⇒ ('a, 'b fset) fmap **is** λa b. fmupd a {b} fmempty  
 ⟨*proof*⟩

**end**

**end**

**end**

## 31 Disjoint FSets

**theory** *Disjoint-FSets*

**imports**

*HOL-Library.Finite-Map*

*Disjoint-Sets*

**begin**

**context**

**includes** *fset.lifting*

**begin**

**lift-definition** *fdisjnt* :: 'a fset ⇒ 'a fset ⇒ bool **is** *disjnt* ⟨*proof*⟩

**lemma** *fdisjnt-alt-def*: *fdisjnt* M N ↔ (M |∩| N = {||})  
 ⟨*proof*⟩

**lemma** *fdisjnt-insert*: x ∉| N ⇒ *fdisjnt* M N ⇒ *fdisjnt* (finsert x M) N  
 ⟨*proof*⟩

**lemma** *fdisjnt-subset-right*: N' |⊆| N ⇒ *fdisjnt* M N ⇒ *fdisjnt* M N'



*<proof>*

**lemma** *fdisjnt-subset-left*:  $N' \sqsubseteq N \implies \text{fdisjnt } N \ M \implies \text{fdisjnt } N' \ M$   
*<proof>*

**lemma** *fdisjnt-union-right*:  $\text{fdisjnt } M \ A \implies \text{fdisjnt } M \ B \implies \text{fdisjnt } M \ (A \mid\cup\mid B)$   
*<proof>*

**lemma** *fdisjnt-union-left*:  $\text{fdisjnt } A \ M \implies \text{fdisjnt } B \ M \implies \text{fdisjnt } (A \mid\cup\mid B) \ M$   
*<proof>*

**lemma** *fdisjnt-swap*:  $\text{fdisjnt } M \ N \implies \text{fdisjnt } N \ M$   
**including** *fset.lifting* *<proof>*

**lemma** *distinct-append-fset*:  
**assumes** *distinct xs distinct ys fdisjnt (fset-of-list xs) (fset-of-list ys)*  
**shows** *distinct (xs @ ys)*  
*<proof>*

**lemma** *fdisjnt-contrI*:  
**assumes**  $\bigwedge x. x \mid\in\mid M \implies x \mid\in\mid N \implies \text{False}$   
**shows**  $\text{fdisjnt } M \ N$   
*<proof>*

**lemma** *fdisjnt-Union-left*:  $\text{fdisjnt } (\text{ffUnion } S) \ T \longleftrightarrow \text{fBall } S \ (\lambda S. \text{fdisjnt } S \ T)$   
*<proof>*

**lemma** *fdisjnt-Union-right*:  $\text{fdisjnt } T \ (\text{ffUnion } S) \longleftrightarrow \text{fBall } S \ (\lambda S. \text{fdisjnt } T \ S)$   
*<proof>*

**lemma** *fdisjnt-ge-max*:  $\text{fBall } X \ (\lambda x. x > \text{fMax } Y) \implies \text{fdisjnt } X \ Y$   
*<proof>*

**end**

**lemma** *fmadd-disjnt*:  $\text{fdisjnt } (\text{fmdom } m) \ (\text{fmdom } n) \implies m \ ++_f \ n = n \ ++_f \ m$   
*<proof>*  
**including** *fset.lifting* **and** *fmap.lifting*  
*<proof>*

**end**

## 32 Lists with elements distinct as canonical example for datatype invariants

**theory** *Dlist*  
**imports** *Confluent-Quotient*

**begin**

### 32.1 The type of distinct lists

**typedef**  $'a$  *dlist* = { $xs::'a$  *list*. *distinct*  $xs$ }  
**morphisms** *list-of-dlist* *Abs-dlist*  
 ⟨*proof*⟩

**context begin**

**qualified definition** *dlist-eq* **where**  $dlist-eq = BNF-Def.vimage2p$  *remdups* *remdups*  
 (=)

**qualified lemma** *equivp-dlist-eq*: *equivp* *dlist-eq*  
 ⟨*proof*⟩ **definition** *abs-dlist* ::  $'a$  *list*  $\Rightarrow$   $'a$  *dlist* **where**  $abs-dlist = Abs-dlist \circ$   
*remdups*

**definition** *qcr-dlist* ::  $'a$  *list*  $\Rightarrow$   $'a$  *dlist*  $\Rightarrow$  *bool* **where**  $qcr-dlist$   $x$   $y \iff y =$   
*abs-dlist*  $x$

**qualified lemma** *Quotient-dlist-remdups*: *Quotient* *dlist-eq* *abs-dlist* *list-of-dlist*  
*qcr-dlist*  
 ⟨*proof*⟩

**end**

**locale** *Quotient-dlist* **begin**

**setup-lifting** *Dlist*.*Quotient-dlist-remdups* *Dlist*.*equivp-dlist-eq*[*THEN* *equivp-reflp2*]  
**end**

**setup-lifting** *type-definition-dlist*

**lemma** *dlist-eq-iff*:  
 $dxs = dys \iff list-of-dlist$   $dxs = list-of-dlist$   $dys$   
 ⟨*proof*⟩

**lemma** *dlist-eqI*:  
 $list-of-dlist$   $dxs = list-of-dlist$   $dys \implies dxs = dys$   
 ⟨*proof*⟩

Formal, totalized constructor for  $'a$  *dlist*:

**definition** *Dlist* ::  $'a$  *list*  $\Rightarrow$   $'a$  *dlist* **where**  
 $Dlist$   $xs = Abs-dlist$  (*remdups*  $xs$ )

**lemma** *distinct-list-of-dlist* [*simp*, *intro*]:  
 $distinct$  ( $list-of-dlist$   $dxs$ )  
 ⟨*proof*⟩

**lemma** *list-of-dlist-Dlist* [*simp*]:  
 $list-of-dlist$  ( $Dlist$   $xs$ ) = *remdups*  $xs$

*<proof>*

**lemma** *remdups-list-of-dlist* [*simp*]:  
 $\text{remdups } (\text{list-of-dlist } dxs) = \text{list-of-dlist } dxs$   
*<proof>*

**lemma** *Dlist-list-of-dlist* [*simp, code abstype*]:  
 $\text{Dlist } (\text{list-of-dlist } dxs) = dxs$   
*<proof>*

Fundamental operations:

**context**  
**begin**

**qualified definition** *empty* :: 'a dlist **where**  
 $\text{empty} = \text{Dlist } []$

**qualified definition** *insert* :: 'a  $\Rightarrow$  'a dlist  $\Rightarrow$  'a dlist **where**  
 $\text{insert } x \ dxs = \text{Dlist } (\text{List.insert } x \ (\text{list-of-dlist } dxs))$

**qualified definition** *remove* :: 'a  $\Rightarrow$  'a dlist  $\Rightarrow$  'a dlist **where**  
 $\text{remove } x \ dxs = \text{Dlist } (\text{remove1 } x \ (\text{list-of-dlist } dxs))$

**qualified definition** *map* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a dlist  $\Rightarrow$  'b dlist **where**  
 $\text{map } f \ dxs = \text{Dlist } (\text{remdups } (\text{List.map } f \ (\text{list-of-dlist } dxs)))$

**qualified definition** *filter* :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a dlist  $\Rightarrow$  'a dlist **where**  
 $\text{filter } P \ dxs = \text{Dlist } (\text{List.filter } P \ (\text{list-of-dlist } dxs))$

**qualified definition** *rotate* :: nat  $\Rightarrow$  'a dlist  $\Rightarrow$  'a dlist **where**  
 $\text{rotate } n \ dxs = \text{Dlist } (\text{List.rotate } n \ (\text{list-of-dlist } dxs))$

**end**

Derived operations:

**context**  
**begin**

**qualified definition** *null* :: 'a dlist  $\Rightarrow$  bool **where**  
 $\text{null } dxs = \text{List.null } (\text{list-of-dlist } dxs)$

**qualified definition** *member* :: 'a dlist  $\Rightarrow$  'a  $\Rightarrow$  bool **where**  
 $\text{member } dxs = \text{List.member } (\text{list-of-dlist } dxs)$

**qualified definition** *length* :: 'a dlist  $\Rightarrow$  nat **where**  
 $\text{length } dxs = \text{List.length } (\text{list-of-dlist } dxs)$

**qualified definition** *fold* :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'a dlist  $\Rightarrow$  'b  $\Rightarrow$  'b **where**  
 $\text{fold } f \ dxs = \text{List.fold } f \ (\text{list-of-dlist } dxs)$

**qualified definition**  $foldr :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \text{ dlist} \Rightarrow 'b \Rightarrow 'b$  **where**  
 $foldr f dxs = List.foldr f (list-of-dlist dxs)$

**end**

### 32.2 Executable version obeying invariant

**lemma** *list-of-dlist-empty* [*simp*, *code abstract*]:

$list-of-dlist \ Dlist.empty = []$

*<proof>*

**lemma** *list-of-dlist-insert* [*simp*, *code abstract*]:

$list-of-dlist (Dlist.insert x dxs) = List.insert x (list-of-dlist dxs)$

*<proof>*

**lemma** *list-of-dlist-remove* [*simp*, *code abstract*]:

$list-of-dlist (Dlist.remove x dxs) = remove1 x (list-of-dlist dxs)$

*<proof>*

**lemma** *list-of-dlist-map* [*simp*, *code abstract*]:

$list-of-dlist (Dlist.map f dxs) = remdups (List.map f (list-of-dlist dxs))$

*<proof>*

**lemma** *list-of-dlist-filter* [*simp*, *code abstract*]:

$list-of-dlist (Dlist.filter P dxs) = List.filter P (list-of-dlist dxs)$

*<proof>*

**lemma** *list-of-dlist-rotate* [*simp*, *code abstract*]:

$list-of-dlist (Dlist.rotate n dxs) = List.rotate n (list-of-dlist dxs)$

*<proof>*

Explicit executable conversion

**definition** *dlist-of-list* [*simp*]:

$dlist-of-list = Dlist$

**lemma** [*code abstract*]:

$list-of-dlist (dlist-of-list xs) = remdups xs$

*<proof>*

Equality

**instantiation** *dlist* :: (*equal*) *equal*

**begin**

**definition**  $HOL.equal \ dxs \ dys \longleftrightarrow HOL.equal (list-of-dlist dxs) (list-of-dlist dys)$

**instance**

*<proof>*

**end**

**declare** *equal-dlist-def* [*code*]

**lemma** [*code nbe*]: *HOL.equal* (*dxs* :: 'a::equal dlist) *dxs*  $\longleftrightarrow$  *True*  
 ⟨*proof*⟩

### 32.3 Induction principle and case distinction

**lemma** *dlist-induct* [*case-names empty insert, induct type: dlist*]:  
**assumes** *empty*: *P Dlist.empty*  
**assumes** *insrt*:  $\bigwedge x \text{ dxs. } \neg \text{Dlist.member dxs } x \implies P \text{ dxs} \implies P (\text{Dlist.insert } x \text{ dxs})$   
**shows** *P dxs*  
 ⟨*proof*⟩

**lemma** *dlist-case* [*cases type: dlist*]:  
**obtains** (*empty*) *dxs = Dlist.empty*  
 | (*insert*) *x dys* **where**  $\neg \text{Dlist.member dys } x$  **and** *dxs = Dlist.insert x dys*  
 ⟨*proof*⟩

### 32.4 Functorial structure

**functor** *map*: *map*  
 ⟨*proof*⟩

### 32.5 Quickcheck generators

**quickcheck-generator** *dlist predicate: distinct constructors: Dlist.empty, Dlist.insert*

### 32.6 BNF instance

**context** *begin*

**qualified inductive** *double* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool **where**  
*double* (*xs @ ys*) (*xs @ x # ys*) **if**  $x \in \text{set } ys$

**qualified lemma** *strong-confluentp-double*: *strong-confluentp double*  
 ⟨*proof*⟩ **lemma** *double-Cons1* [*simp*]: *double xs* ( $x \# xs$ ) **if**  $x \in \text{set } xs$   
 ⟨*proof*⟩ **lemma** *double-Cons-same* [*simp*]: *double xs ys*  $\implies$  *double* ( $x \# xs$ ) ( $x \# ys$ )  
 ⟨*proof*⟩ **lemma** *doubles-Cons-same*: *double\*\* xs ys*  $\implies$  *double\*\** ( $x \# xs$ ) ( $x \# ys$ )  
 ⟨*proof*⟩ **lemma** *remdups-into-doubles*: *double\*\* (remdups xs) xs*  
 ⟨*proof*⟩ **lemma** *dlist-eq-into-doubles*: *Dlist.dlist-eq*  $\leq$  *equivclp double*  
 ⟨*proof*⟩ **lemma** *factor-double-map*: *double (map f xs) ys*  $\implies$   $\exists zs. \text{Dlist.dlist-eq } xs \text{ } zs \wedge ys = \text{map } f \text{ } zs \wedge \text{set } zs \subseteq \text{set } xs$   
 ⟨*proof*⟩ **lemma** *dlist-eq-set-eq*: *Dlist.dlist-eq xs ys*  $\implies$  *set xs = set ys*  
 ⟨*proof*⟩ **lemma** *dlist-eq-map-respect*: *Dlist.dlist-eq xs ys*  $\implies$  *Dlist.dlist-eq (map f xs) (map f ys)*  
 ⟨*proof*⟩ **lemma** *confluent-quotient-dlist*:

```

    confluent-quotient double Dlist.dlist-eq Dlist.dlist-eq Dlist.dlist-eq Dlist.dlist-eq
    Dlist.dlist-eq
    (map fst) (map snd) (map fst) (map snd) list-all2 list-all2 list-all2 set set
    <proof>

```

```

lifting-update dlist.lifting
lifting-forget dlist.lifting

```

```

end

```

```

context begin
interpretation Quotient-dlist: Quotient-dlist <proof>

```

```

lift-bnf (plugins del: code) 'a dlist
  <proof> lemma list-of-dlist-transfer[transfer-rule]:
  bi-unique R  $\implies$  (rel-fun (Quotient-dlist.pcr-dlist R) (list-all2 R)) remdups list-of-dlist
  <proof>

```

```

lemma list-of-dlist-map-dlist[simp]:
  list-of-dlist (map-dlist f xs) = remdups (map f (list-of-dlist xs))
  <proof>

```

```

end

```

```

end

```

### 33 Type of dual ordered lattices

```

theory Dual-Ordered-Lattice
imports Main
begin

```

The *dual* of an ordered structure is an isomorphic copy of the underlying type, with the  $\leq$  relation defined as the inverse of the original one.

The class of lattices is closed under formation of dual structures. This means that for any theorem of lattice theory, the dualized statement holds as well; this important fact simplifies many proofs of lattice theory.

```

typedef 'a dual = UNIV :: 'a set
morphisms undual dual <proof>

```

```

setup-lifting type-definition-dual

```

```

code-datatype dual

```

```

lemma dual-eqI:
  x = y if undual x = undual y
  <proof>

```

**lemma** *dual-eq-iff*:

$$x = y \longleftrightarrow \text{undual } x = \text{undual } y$$

*<proof>*

**lemma** *eq-dual-iff [iff]*:

$$\text{dual } x = \text{dual } y \longleftrightarrow x = y$$

*<proof>*

**lemma** *undual-dual [simp, code]*:

$$\text{undual } (\text{dual } x) = x$$

*<proof>*

**lemma** *dual-undual [simp]*:

$$\text{dual } (\text{undual } x) = x$$

*<proof>*

**lemma** *undual-comp-dual [simp]*:

$$\text{undual } \circ \text{dual} = \text{id}$$

*<proof>*

**lemma** *dual-comp-undual [simp]*:

$$\text{dual } \circ \text{undual} = \text{id}$$

*<proof>*

**lemma** *inj-dual*:

$$\text{inj } \text{dual}$$

*<proof>*

**lemma** *inj-undual*:

$$\text{inj } \text{undual}$$

*<proof>*

**lemma** *surj-dual*:

$$\text{surj } \text{dual}$$

*<proof>*

**lemma** *surj-undual*:

$$\text{surj } \text{undual}$$

*<proof>*

**lemma** *bij-dual*:

$$\text{bij } \text{dual}$$

*<proof>*

**lemma** *bij-undual*:

$$\text{bij } \text{undual}$$

*<proof>*

```

instance dual :: (finite) finite
  ⟨proof⟩

instantiation dual :: (equal) equal
begin

lift-definition equal-dual :: 'a dual ⇒ 'a dual ⇒ bool
  is HOL.equal ⟨proof⟩

instance
  ⟨proof⟩

end

```

### 33.1 Pointwise ordering

```

instantiation dual :: (ord) ord
begin

lift-definition less-eq-dual :: 'a dual ⇒ 'a dual ⇒ bool
  is (≥) ⟨proof⟩

lift-definition less-dual :: 'a dual ⇒ 'a dual ⇒ bool
  is (>) ⟨proof⟩

instance ⟨proof⟩

end

lemma dual-less-eqI:
   $x \leq y$  if undual  $y \leq$  undual  $x$ 
  ⟨proof⟩

lemma dual-less-eq-iff:
   $x \leq y \iff$  undual  $y \leq$  undual  $x$ 
  ⟨proof⟩

lemma less-eq-dual-iff [iff]:
   $dual\ x \leq dual\ y \iff y \leq x$ 
  ⟨proof⟩

lemma dual-lessI:
   $x < y$  if undual  $y <$  undual  $x$ 
  ⟨proof⟩

lemma dual-less-iff:
   $x < y \iff$  undual  $y <$  undual  $x$ 
  ⟨proof⟩

```



**lemma** *less-dual-iff* [*iff*]:  
 $dual\ x < dual\ y \longleftrightarrow y < x$   
 ⟨*proof*⟩

**instance** *dual* :: (*preorder*) *preorder*  
 ⟨*proof*⟩

**instance** *dual* :: (*order*) *order*  
 ⟨*proof*⟩

### 33.2 Binary infimum and supremum

**instantiation** *dual* :: (*sup*) *inf*  
**begin**

**lift-definition** *inf-dual* :: 'a *dual*  $\Rightarrow$  'a *dual*  $\Rightarrow$  'a *dual*  
 is *sup* ⟨*proof*⟩

**instance** ⟨*proof*⟩

**end**

**lemma** *undual-inf-eq* [*simp*]:  
 $undual\ (inf\ x\ y) = sup\ (undual\ x)\ (undual\ y)$   
 ⟨*proof*⟩

**lemma** *dual-sup-eq* [*simp*]:  
 $dual\ (sup\ x\ y) = inf\ (dual\ x)\ (dual\ y)$   
 ⟨*proof*⟩

**instantiation** *dual* :: (*inf*) *sup*  
**begin**

**lift-definition** *sup-dual* :: 'a *dual*  $\Rightarrow$  'a *dual*  $\Rightarrow$  'a *dual*  
 is *inf* ⟨*proof*⟩

**instance** ⟨*proof*⟩

**end**

**lemma** *undual-sup-eq* [*simp*]:  
 $undual\ (sup\ x\ y) = inf\ (undual\ x)\ (undual\ y)$   
 ⟨*proof*⟩

**lemma** *dual-inf-eq* [*simp*]:  
 $dual\ (inf\ x\ y) = sup\ (dual\ x)\ (dual\ y)$   
 ⟨*proof*⟩

**instance** *dual* :: (*semilattice-sup*) *semilattice-inf*

*<proof>*

**instance** *dual* :: (*semilattice-inf*) *semilattice-sup*  
*<proof>*

**instance** *dual* :: (*lattice*) *lattice* *<proof>*

**instance** *dual* :: (*distrib-lattice*) *distrib-lattice*  
*<proof>*

### 33.3 Top and bottom elements

**instantiation** *dual* :: (*top*) *bot*  
**begin**

**lift-definition** *bot-dual* :: 'a *dual*  
**is top** *<proof>*

**instance** *<proof>*

**end**

**lemma** *undual-bot-eq* [*simp*]:  
*undual bot = top*  
*<proof>*

**lemma** *dual-top-eq* [*simp*]:  
*dual top = bot*  
*<proof>*

**instantiation** *dual* :: (*bot*) *top*  
**begin**

**lift-definition** *top-dual* :: 'a *dual*  
**is bot** *<proof>*

**instance** *<proof>*

**end**

**lemma** *undual-top-eq* [*simp*]:  
*undual top = bot*  
*<proof>*

**lemma** *dual-bot-eq* [*simp*]:  
*dual bot = top*  
*<proof>*

**instance** *dual* :: (*order-top*) *order-bot*

*<proof>*

**instance** *dual* :: (*order-bot*) *order-top*  
*<proof>*

**instance** *dual* :: (*bounded-lattice-top*) *bounded-lattice-bot* *<proof>*

**instance** *dual* :: (*bounded-lattice-bot*) *bounded-lattice-top* *<proof>*

**instance** *dual* :: (*bounded-lattice*) *bounded-lattice* *<proof>*

### 33.4 Complement

**instantiation** *dual* :: (*uminus*) *uminus*  
**begin**

**lift-definition** *uminus-dual* :: 'a *dual* ⇒ 'a *dual*  
**is** *uminus* *<proof>*

**instance** *<proof>*

**end**

**lemma** *undual-uminus-eq* [*simp*]:  
 $undual (- x) = - undual x$   
*<proof>*

**lemma** *dual-uminus-eq* [*simp*]:  
 $dual (- x) = - dual x$   
*<proof>*

**instantiation** *dual* :: (*boolean-algebra*) *boolean-algebra*  
**begin**

**lift-definition** *minus-dual* :: 'a *dual* ⇒ 'a *dual* ⇒ 'a *dual*  
**is**  $\lambda x y. - (y - x)$  *<proof>*

**instance**  
*<proof>*

**end**

**lemma** *undual-minus-eq* [*simp*]:  
 $undual (x - y) = - (undual y - undual x)$   
*<proof>*

**lemma** *dual-minus-eq* [*simp*]:  
 $dual (x - y) = - (dual y - dual x)$   
*<proof>*

### 33.5 Complete lattice operations

The class of complete lattices is closed under formation of dual structures.

```

instantiation dual :: (Sup) Inf
begin

lift-definition Inf-dual :: 'a dual set  $\Rightarrow$  'a dual
  is Sup  $\langle$ proof $\rangle$ 

instance  $\langle$ proof $\rangle$ 

end

lemma undual-Inf-eq [simp]:
  undual (Inf A) = Sup (undual ' A)
   $\langle$ proof $\rangle$ 

lemma dual-Sup-eq [simp]:
  dual (Sup A) = Inf (dual ' A)
   $\langle$ proof $\rangle$ 

instantiation dual :: (Inf) Sup
begin

lift-definition Sup-dual :: 'a dual set  $\Rightarrow$  'a dual
  is Inf  $\langle$ proof $\rangle$ 

instance  $\langle$ proof $\rangle$ 

end

lemma undual-Sup-eq [simp]:
  undual (Sup A) = Inf (undual ' A)
   $\langle$ proof $\rangle$ 

lemma dual-Inf-eq [simp]:
  dual (Inf A) = Sup (dual ' A)
   $\langle$ proof $\rangle$ 

instance dual :: (complete-lattice) complete-lattice
   $\langle$ proof $\rangle$ 

context
  fixes f :: 'a::complete-lattice  $\Rightarrow$  'a
    and g :: 'a dual  $\Rightarrow$  'a dual
  assumes mono f
  defines g  $\equiv$  dual  $\circ$  f  $\circ$  undual
begin

```

**private lemma** *mono-dual*:

*mono g*  
 $\langle \text{proof} \rangle$

**lemma** *lfp-dual-gfp*:

$\text{lfp } f = \text{undual } (\text{gfp } g)$  (**is** *?lhs = ?rhs*)  
 $\langle \text{proof} \rangle$

**lemma** *gfp-dual-lfp*:

$\text{gfp } f = \text{undual } (\text{lfp } g)$   
 $\langle \text{proof} \rangle$

**end**

Finally

**lifting-update** *dual.lifting*

**lifting-forget** *dual.lifting*

**end**

## 34 Equipollence and Other Relations Connected with Cardinality

**theory** *Equipollence*

**imports** *FuncSet Countable-Set*

**begin**

### 34.1 Eqpoll

**definition** *eqpoll* :: 'a set  $\Rightarrow$  'b set  $\Rightarrow$  bool (**infixl**  $\langle \approx \rangle$  50)  
 where  $\text{eqpoll } A B \equiv \exists f. \text{bij-betw } f A B$

**definition** *lepoll* :: 'a set  $\Rightarrow$  'b set  $\Rightarrow$  bool (**infixl**  $\langle \lesssim \rangle$  50)  
 where  $\text{lepoll } A B \equiv \exists f. \text{inj-on } f A \wedge f ' A \subseteq B$

**definition** *lesspoll* :: 'a set  $\Rightarrow$  'b set  $\Rightarrow$  bool (**infixl**  $\langle \prec \rangle$  50)  
 where  $A \prec B \equiv A \lesssim B \wedge \sim(A \approx B)$

**lemma** *lepoll-def'*:  $\text{lepoll } A B \equiv \exists f. \text{inj-on } f A \wedge f \in A \rightarrow B$   
 $\langle \text{proof} \rangle$

**lemma** *eqpoll-empty-iff-empty* [*simp*]:  $A \approx \{\} \longleftrightarrow A = \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *lepoll-empty-iff-empty* [*simp*]:  $A \lesssim \{\} \longleftrightarrow A = \{\}$   
 $\langle \text{proof} \rangle$

**lemma** *not-lesspoll-empty*:  $\neg A \prec \{\}$

*<proof>*

**lemma** *lepoll-relational-full*:

**assumes**  $\bigwedge y. y \in B \implies \exists x. x \in A \wedge R x y$

**and**  $\bigwedge x y y'. \llbracket x \in A; y \in B; y' \in B; R x y; R x y' \rrbracket \implies y = y'$

**shows**  $B \lesssim A$

*<proof>*

**lemma** *eqpoll-iff-card-of-ordIso*:  $A \approx B \longleftrightarrow \text{ordIso2 } (\text{card-of } A) (\text{card-of } B)$

*<proof>*

**lemma** *eqpoll-refl [iff]*:  $A \approx A$

*<proof>*

**lemma** *eqpoll-finite-iff*:  $A \approx B \implies \text{finite } A \longleftrightarrow \text{finite } B$

*<proof>*

**lemma** *eqpoll-iff-card*:

**assumes** *finite*  $A$  *finite*  $B$

**shows**  $A \approx B \longleftrightarrow \text{card } A = \text{card } B$

*<proof>*

**lemma** *eqpoll-singleton-iff*:  $A \approx \{x\} \longleftrightarrow (\exists u. A = \{u\})$

*<proof>*

**lemma** *eqpoll-doubleton-iff*:  $A \approx \{x, y\} \longleftrightarrow (\exists u v. A = \{u, v\} \wedge (u=v \longleftrightarrow x=y))$

*<proof>*

**lemma** *lepoll-antisym*:

**assumes**  $A \lesssim B$   $B \lesssim A$  **shows**  $A \approx B$

*<proof>*

**lemma** *lepoll-trans [trans]*:

**assumes**  $A \lesssim B$   $B \lesssim C$  **shows**  $A \lesssim C$

*<proof>*

**lemma** *lepoll-trans1 [trans]*:  $\llbracket A \approx B; B \lesssim C \rrbracket \implies A \lesssim C$

*<proof>*

**lemma** *lepoll-trans2 [trans]*:  $\llbracket A \lesssim B; B \approx C \rrbracket \implies A \lesssim C$

*<proof>*

**lemma** *eqpoll-sym*:  $A \approx B \implies B \approx A$

*<proof>*

**lemma** *eqpoll-trans [trans]*:  $\llbracket A \approx B; B \approx C \rrbracket \implies A \approx C$

*<proof>*

**lemma** *eqpoll-imp-lepoll*:  $A \approx B \implies A \lesssim B$   
 ⟨proof⟩

**lemma** *subset-imp-lepoll*:  $A \subseteq B \implies A \lesssim B$   
 ⟨proof⟩

**lemma** *lepoll-refl [iff]*:  $A \lesssim A$   
 ⟨proof⟩

**lemma** *lepoll-iff*:  $A \lesssim B \iff (\exists g. A \subseteq g \text{ ‘ } B)$   
 ⟨proof⟩

**lemma** *empty-lepoll [iff]*:  $\{\} \lesssim A$   
 ⟨proof⟩

**lemma** *subset-image-lepoll*:  $B \subseteq f \text{ ‘ } A \implies B \lesssim A$   
 ⟨proof⟩

**lemma** *image-lepoll*:  $f \text{ ‘ } A \lesssim A$   
 ⟨proof⟩

**lemma** *infinite-le-lepoll*: *infinite*  $A \iff (UNIV::\text{nat set}) \lesssim A$   
 ⟨proof⟩

**lemma** *lepoll-Pow-self*:  $A \lesssim \text{Pow } A$   
 ⟨proof⟩

**lemma** *eqpoll-iff-bijections*:  
 $A \approx B \iff (\exists f g. (\forall x \in A. f x \in B \wedge g(f x) = x) \wedge (\forall y \in B. g y \in A \wedge f(g y) = y))$   
 ⟨proof⟩

**lemma** *lepoll-restricted-funspace*:  
 $\{f. f \text{ ‘ } A \subseteq B \wedge \{x. f x \neq k x\} \subseteq A \wedge \text{finite } \{x. f x \neq k x\}\} \lesssim \text{Fpow } (A \times B)$   
 ⟨proof⟩

**lemma** *singleton-lepoll*:  $\{x\} \lesssim \text{insert } y A$   
 ⟨proof⟩

**lemma** *singleton-epoll*:  $\{x\} \approx \{y\}$   
 ⟨proof⟩

**lemma** *subset-singleton-iff-lepoll*:  $(\exists x. S \subseteq \{x\}) \iff S \lesssim \{\}$   
 ⟨proof⟩

**lemma** *infinite-insert-lepoll*:  
**assumes** *infinite*  $A$  **shows**  $\text{insert } a A \lesssim A$   
 ⟨proof⟩

**lemma** *infinite-insert-epoll*:  $\text{infinite } A \implies \text{insert } a \ A \approx A$   
 ⟨proof⟩

**lemma** *finite-lepoll-infinite*:  
**assumes** *infinite*  $A$  *finite*  $B$  **shows**  $B \lesssim A$   
 ⟨proof⟩

**lemma** *countable-lepoll*:  $\llbracket \text{countable } A; B \lesssim A \rrbracket \implies \text{countable } B$   
 ⟨proof⟩

**lemma** *countable-epoll*:  $\llbracket \text{countable } A; B \approx A \rrbracket \implies \text{countable } B$   
 ⟨proof⟩

### 34.2 The strict relation

**lemma** *lesspoll-not-refl* [*iff*]:  $\sim (i \prec i)$   
 ⟨proof⟩

**lemma** *lesspoll-imp-lepoll*:  $A \prec B \implies A \lesssim B$   
 ⟨proof⟩

**lemma** *lepoll-iff-leqpoll*:  $A \lesssim B \iff A \prec B \mid A \approx B$   
 ⟨proof⟩

**lemma** *lesspoll-trans* [*trans*]:  $\llbracket X \prec Y; Y \prec Z \rrbracket \implies X \prec Z$   
 ⟨proof⟩

**lemma** *lesspoll-trans1* [*trans*]:  $\llbracket X \lesssim Y; Y \prec Z \rrbracket \implies X \prec Z$   
 ⟨proof⟩

**lemma** *lesspoll-trans2* [*trans*]:  $\llbracket X \prec Y; Y \lesssim Z \rrbracket \implies X \prec Z$   
 ⟨proof⟩

**lemma** *eq-lesspoll-trans* [*trans*]:  $\llbracket X \approx Y; Y \prec Z \rrbracket \implies X \prec Z$   
 ⟨proof⟩

**lemma** *lesspoll-eq-trans* [*trans*]:  $\llbracket X \prec Y; Y \approx Z \rrbracket \implies X \prec Z$   
 ⟨proof⟩

**lemma** *lesspoll-Pow-self*:  $A \prec \text{Pow } A$   
 ⟨proof⟩

**lemma** *finite-lesspoll-infinite*:  
**assumes** *infinite*  $A$  *finite*  $B$  **shows**  $B \prec A$   
 ⟨proof⟩

**lemma** *countable-lesspoll*:  $\llbracket \text{countable } A; B \prec A \rrbracket \implies \text{countable } B$   
 ⟨proof⟩



**lemma** *lepoll-iff-card-le*:  $\llbracket \text{finite } A; \text{ finite } B \rrbracket \implies A \lesssim B \iff \text{card } A \leq \text{card } B$   
 ⟨proof⟩

**lemma** *lepoll-iff-finite-card*:  $A \lesssim \{..<n::\text{nat}\} \iff \text{finite } A \wedge \text{card } A \leq n$   
 ⟨proof⟩

**lemma** *eqpoll-iff-finite-card*:  $A \approx \{..<n::\text{nat}\} \iff \text{finite } A \wedge \text{card } A = n$   
 ⟨proof⟩

**lemma** *lesspoll-iff-finite-card*:  $A \prec \{..<n::\text{nat}\} \iff \text{finite } A \wedge \text{card } A < n$   
 ⟨proof⟩

### 34.3 Mapping by an injection

**lemma** *inj-on-image-epoll-self*:  $\text{inj-on } f A \implies f ' A \approx A$   
 ⟨proof⟩

**lemma** *inj-on-image-lepoll-1* [simp]:  
 assumes *inj-on*  $f A$  shows  $f ' A \lesssim B \iff A \lesssim B$   
 ⟨proof⟩

**lemma** *inj-on-image-lepoll-2* [simp]:  
 assumes *inj-on*  $f B$  shows  $A \lesssim f ' B \iff A \lesssim B$   
 ⟨proof⟩

**lemma** *inj-on-image-lesspoll-1* [simp]:  
 assumes *inj-on*  $f A$  shows  $f ' A \prec B \iff A \prec B$   
 ⟨proof⟩

**lemma** *inj-on-image-lesspoll-2* [simp]:  
 assumes *inj-on*  $f B$  shows  $A \prec f ' B \iff A \prec B$   
 ⟨proof⟩

**lemma** *inj-on-image-epoll-1* [simp]:  
 assumes *inj-on*  $f A$  shows  $f ' A \approx B \iff A \approx B$   
 ⟨proof⟩

**lemma** *inj-on-image-epoll-2* [simp]:  
 assumes *inj-on*  $f B$  shows  $A \approx f ' B \iff A \approx B$   
 ⟨proof⟩

### 34.4 Inserting elements into sets

**lemma** *insert-lepoll-insertD*:  
 assumes *insert*  $u A \lesssim \text{insert } v B$   $u \notin A$   $v \notin B$  shows  $A \lesssim B$   
 ⟨proof⟩

**lemma** *insert-epoll-insertD*:  $\llbracket \text{insert } u A \approx \text{insert } v B; u \notin A; v \notin B \rrbracket \implies A \approx B$   
 ⟨proof⟩

**lemma** *insert-lepoll-cong*:

**assumes**  $A \lesssim B$   $b \notin B$  **shows**  $\text{insert } a \ A \lesssim \text{insert } b \ B$   
 ⟨proof⟩

**lemma** *insert-epoll-cong*:

$\llbracket A \approx B; a \notin A; b \notin B \rrbracket \implies \text{insert } a \ A \approx \text{insert } b \ B$   
 ⟨proof⟩

**lemma** *insert-epoll-insert-iff*:

$\llbracket a \notin A; b \notin B \rrbracket \implies \text{insert } a \ A \approx \text{insert } b \ B \longleftrightarrow A \approx B$   
 ⟨proof⟩

**lemma** *insert-lepoll-insert-iff*:

$\llbracket a \notin A; b \notin B \rrbracket \implies (\text{insert } a \ A \lesssim \text{insert } b \ B) \longleftrightarrow (A \lesssim B)$   
 ⟨proof⟩

**lemma** *less-imp-insert-lepoll*:

**assumes**  $A \prec B$  **shows**  $\text{insert } a \ A \lesssim B$   
 ⟨proof⟩

**lemma** *finite-insert-lepoll*:  $\text{finite } A \implies (\text{insert } a \ A \lesssim A) \longleftrightarrow (a \in A)$

⟨proof⟩

### 34.5 Binary sums and unions

**lemma** *Un-lepoll-mono*:

**assumes**  $A \lesssim C$   $B \lesssim D$   $\text{disjnt } C \ D$  **shows**  $A \cup B \lesssim C \cup D$   
 ⟨proof⟩

**lemma** *Un-epoll-cong*:  $\llbracket A \approx C; B \approx D; \text{disjnt } A \ B; \text{disjnt } C \ D \rrbracket \implies A \cup B \approx C \cup D$

⟨proof⟩

**lemma** *sum-lepoll-mono*:

**assumes**  $A \lesssim C$   $B \lesssim D$  **shows**  $A \langle + \rangle B \lesssim C \langle + \rangle D$   
 ⟨proof⟩

**lemma** *sum-epoll-cong*:  $\llbracket A \approx C; B \approx D \rrbracket \implies A \langle + \rangle B \approx C \langle + \rangle D$

⟨proof⟩

### 34.6 Binary Cartesian products

**lemma** *times-square-lepoll*:  $A \lesssim A \times A$

⟨proof⟩

**lemma** *times-commute-epoll*:  $A \times B \approx B \times A$

⟨proof⟩

**lemma** *times-assoc-epoll*:  $(A \times B) \times C \approx A \times (B \times C)$

⟨proof⟩

**lemma** *times-singleton-epoll*:  $\{a\} \times A \approx A$   
 ⟨proof⟩

**lemma** *times-lepoll-mono*:  
 assumes  $A \lesssim C$   $B \lesssim D$  shows  $A \times B \lesssim C \times D$   
 ⟨proof⟩

**lemma** *times-epoll-cong*:  $\llbracket A \approx C; B \approx D \rrbracket \implies A \times B \approx C \times D$   
 ⟨proof⟩

**lemma**  
 assumes  $B \neq \{\}$  shows *lepoll-times1*:  $A \lesssim A \times B$  and *lepoll-times2*:  $A \lesssim B \times A$   
 ⟨proof⟩

**lemma** *times-0-epoll*:  $\{\} \times A \approx \{\}$   
 ⟨proof⟩

**lemma** *Sigma-inj-lepoll-mono*:  
 assumes  $h$ : *inj-on*  $h$   $A$   $h$  ‘  $A \subseteq C$  and  $\bigwedge x. x \in A \implies B x \lesssim D (h x)$   
 shows  $\text{Sigma } A B \lesssim \text{Sigma } C D$   
 ⟨proof⟩

**lemma** *Sigma-lepoll-mono*:  
 assumes  $A \subseteq C$   $\bigwedge x. x \in A \implies B x \lesssim D x$  shows  $\text{Sigma } A B \lesssim \text{Sigma } C D$   
 ⟨proof⟩

**lemma** *sum-times-distrib-epoll*:  $(A <+> B) \times C \approx (A \times C) <+> (B \times C)$   
 ⟨proof⟩

**lemma** *Sigma-epoll-cong*:  
 assumes  $h$ : *bij-betw*  $h$   $A$   $C$  and  $BD$ :  $\bigwedge x. x \in A \implies B x \approx D (h x)$   
 shows  $\text{Sigma } A B \approx \text{Sigma } C D$   
 ⟨proof⟩

**lemma** *prod-insert-epoll*:  
 assumes  $a \notin A$  shows *insert a*  $A \times B \approx B <+> A \times B$   
 ⟨proof⟩

### 34.7 General Unions

**lemma** *Union-epoll-Times*:  
 assumes  $B$ :  $\bigwedge x. x \in A \implies F x \approx B$  and *disj*: *pairwise*  $(\lambda x y. \text{disjnt } (F x) (F y)) A$   
 shows  $(\bigcup_{x \in A} F x) \approx A \times B$   
 ⟨proof⟩

**lemma** *UN-lepoll-UN*:

**assumes**  $A: \bigwedge x. x \in A \implies B x \lesssim C x$   
**and**  $disj: pairwise (\lambda x y. disjnt (C x) (C y)) A$   
**shows**  $\bigcup (B'A) \lesssim \bigcup (C'A)$   
 $\langle proof \rangle$

**lemma** *UN-eqpoll-UN*:  
**assumes**  $A: \bigwedge x. x \in A \implies B x \approx C x$   
**and**  $B: pairwise (\lambda x y. disjnt (B x) (B y)) A$   
**and**  $C: pairwise (\lambda x y. disjnt (C x) (C y)) A$   
**shows**  $(\bigcup_{x \in A}. B x) \approx (\bigcup_{x \in A}. C x)$   
 $\langle proof \rangle$

### 34.8 General Cartesian products (Pi)

**lemma** *PiE-sing-eqpoll-self*:  $(\{a\} \rightarrow_E B) \approx B$   
 $\langle proof \rangle$

**lemma** *lepoll-funcset-right*:  
**assumes**  $B \lesssim B'$  **shows**  $A \rightarrow_E B \lesssim A \rightarrow_E B'$   
 $\langle proof \rangle$

**lemma** *lepoll-funcset-left*:  
**assumes**  $B \neq \{\}$   $A \lesssim A'$   
**shows**  $A \rightarrow_E B \lesssim A' \rightarrow_E B$   
 $\langle proof \rangle$

**lemma** *lepoll-funcset*:  
 $\llbracket B \neq \{\}; A \lesssim A'; B \lesssim B' \rrbracket \implies A \rightarrow_E B \lesssim A' \rightarrow_E B'$   
 $\langle proof \rangle$

**lemma** *lepoll-PiE*:  
**assumes**  $\bigwedge i. i \in A \implies B i \lesssim C i$   
**shows**  $PiE A B \lesssim PiE A C$   
 $\langle proof \rangle$

**lemma** *card-le-PiE-subindex*:  
**assumes**  $A \subseteq A'$   $PiE A' B \neq \{\}$   
**shows**  $PiE A B \lesssim PiE A' B$   
 $\langle proof \rangle$

**lemma** *finite-restricted-funspace*:  
**assumes** *finite*  $A$  *finite*  $B$   
**shows** *finite*  $\{f. f ' A \subseteq B \wedge \{x. f x \neq k x\} \subseteq A\}$  (**is finite ?F**)  
 $\langle proof \rangle$

**proposition** *finite-PiE-iff*:

$finite(PiE I S) \longleftrightarrow PiE I S = \{\} \vee finite \{i \in I. \sim(\exists a. S i \subseteq \{a\})\} \wedge (\forall i \in I. finite(S i))$   
 (is ?lhs = ?rhs)  
 <proof>

**corollary** *finite-funcset-iff:*

$finite(I \rightarrow_E S) \longleftrightarrow (\exists a. S \subseteq \{a\}) \vee I = \{\} \vee finite I \wedge finite S$   
 <proof>

### 34.9 Misc other resultd

**lemma** *lists-lepoll-mono:*

**assumes**  $A \lesssim B$  **shows**  $lists A \lesssim lists B$   
 <proof>

**lemma** *lepoll-lists:*  $A \lesssim lists A$

<proof>

Dedekind’s definition of infinite set

**lemma** *infinite-iff-psubset:*  $infinite A \longleftrightarrow (\exists B. B \subset A \wedge A \approx B)$

<proof>

**lemma** *infinite-iff-psubset-le:*  $infinite A \longleftrightarrow (\exists B. B \subset A \wedge A \lesssim B)$

<proof>

**end**

**theory** *Simps-Case-Conv*

**imports** *Case-Converter*

**keywords** *simps-of-case case-of-simps :: thy-decl*

**abbrevs** *simps-of-case case-of-simps =*

**begin**

<ML>

**end**

**theory** *Extended*

**imports** *Simps-Case-Conv*

**begin**

**datatype** *'a extended = Fin 'a | Pinf (‹∞›) | Minf (‹−∞›)*

**instantiation** *extended :: (order)order*

**begin**

```

fun less-eq-extended :: 'a extended  $\Rightarrow$  'a extended  $\Rightarrow$  bool where
  Fin x  $\leq$  Fin y = (x  $\leq$  y) |
  -  $\leq$  Pinf = True |
  Minf  $\leq$  - = True |
  (-::'a extended)  $\leq$  - = False

```

**case-of-simps** less-eq-extended-case: less-eq-extended.simps

```

definition less-extended :: 'a extended  $\Rightarrow$  'a extended  $\Rightarrow$  bool where
  ((x::'a extended) < y) = (x  $\leq$  y  $\wedge$   $\neg$  y  $\leq$  x)

```

```

instance
  <proof>

```

**end**

```

instance extended :: (linorder)linorder
  <proof>

```

```

lemma Minf-le[simp]: Minf  $\leq$  y
  <proof>

```

```

lemma le-Pinf[simp]: x  $\leq$  Pinf
  <proof>

```

```

lemma le-Minf[simp]: x  $\leq$  Minf  $\longleftrightarrow$  x = Minf
  <proof>

```

```

lemma Pinf-le[simp]: Pinf  $\leq$  x  $\longleftrightarrow$  x = Pinf
  <proof>

```

```

lemma less-extended-simps[simp]:

```

```

  Fin x < Fin y = (x < y)

```

```

  Fin x < Pinf = True

```

```

  Fin x < Minf = False

```

```

  Pinf < h = False

```

```

  Minf < Fin x = True

```

```

  Minf < Pinf = True

```

```

  l < Minf = False

```

```

  <proof>

```

```

lemma min-extended-simps[simp]:

```

```

  min (Fin x) (Fin y) = Fin(min x y)

```

```

  min xx Pinf = xx

```

```

  min xx Minf = Minf

```

```

  min Pinf yy = yy

```

```

  min Minf yy = Minf

```

```

  <proof>

```

```

lemma max-extended-simps[simp]:

```

```

  max (Fin x) (Fin y) = Fin(max x y)

```

```

  max xx Pinf = Pinf

```

```

max xx   Minf  = xx
max Pinf yy   = Pinf
max Minf  yy   = yy
⟨proof⟩

```

```

instantiation extended :: (zero)zero
begin
definition 0 = Fin(0::'a)
instance ⟨proof⟩
end

```

```

declare zero-extended-def[symmetric, code-post]

```

```

instantiation extended :: (one)one
begin
definition 1 = Fin(1::'a)
instance ⟨proof⟩
end

```

```

declare one-extended-def[symmetric, code-post]

```

```

instantiation extended :: (plus)plus
begin

```

The following definition of addition is totalized to make it associative and commutative. Normally the sum of plus and minus infinity is undefined.

```

fun plus-extended where
Fin x + Fin y = Fin(x+y) |
Fin x + Pinf = Pinf |
Pinf + Fin x = Pinf |
Pinf + Pinf = Pinf |
Minf + Fin y = Minf |
Fin x + Minf = Minf |
Minf + Minf = Minf |
Minf + Pinf = Pinf |
Pinf + Minf = Pinf

```

```

case-of-simps plus-case: plus-extended.simps

```

```

instance ⟨proof⟩

```

```

end

```

```

instance extended :: (ab-semigroup-add)ab-semigroup-add
⟨proof⟩

```

**instance** *extended* :: (*ordered-ab-semigroup-add*)*ordered-ab-semigroup-add*  
 ⟨*proof*⟩

**instance** *extended* :: (*comm-monoid-add*)*comm-monoid-add*  
 ⟨*proof*⟩

**instantiation** *extended* :: (*uminus*)*uminus*  
**begin**

**fun** *uminus-extended* **where**  
 – (*Fin* *x*) = *Fin* (– *x*) |  
 – *Pinf* = *Minf* |  
 – *Minf* = *Pinf*

**instance** ⟨*proof*⟩

**end**

**instantiation** *extended* :: (*ab-group-add*)*minus*  
**begin**

**definition**  $x - y = x + -(y::'a \text{ extended})$

**instance** ⟨*proof*⟩

**end**

**lemma** *minus-extended-simps*[*simp*]:

$Fin\ x - Fin\ y = Fin(x - y)$

$Fin\ x - Pinf = Minf$

$Fin\ x - Minf = Pinf$

$Pinf - Fin\ y = Pinf$

$Pinf - Minf = Pinf$

$Minf - Fin\ y = Minf$

$Minf - Pinf = Minf$

$Minf - Minf = Pinf$

$Pinf - Pinf = Pinf$

⟨*proof*⟩

Numerals:

**instance** *extended* :: ({*ab-semigroup-add,one*})*numeral* ⟨*proof*⟩

**lemma** *Fin-numeral*[*code-post*]:  $Fin(\text{numeral } w) = \text{numeral } w$   
 ⟨*proof*⟩

**lemma** *Fin-neg-numeral*[*code-post*]:  $Fin(- \text{numeral } w) = - \text{numeral } w$   
 ⟨*proof*⟩

**instantiation** *extended* :: (*lattice*)*bounded-lattice*  
**begin**



**definition**  $bot = Minf$

**definition**  $top = Pinf$

**fun**  $inf\text{-extended} :: 'a\ extended \Rightarrow 'a\ extended \Rightarrow 'a\ extended$  **where**  
 $inf\text{-extended} (Fin\ i) (Fin\ j) = Fin\ (inf\ i\ j) \mid$   
 $inf\text{-extended}\ a\ Minf = Minf \mid$   
 $inf\text{-extended}\ Minf\ a = Minf \mid$   
 $inf\text{-extended}\ Pinf\ a = a \mid$   
 $inf\text{-extended}\ a\ Pinf = a$

**fun**  $sup\text{-extended} :: 'a\ extended \Rightarrow 'a\ extended \Rightarrow 'a\ extended$  **where**  
 $sup\text{-extended} (Fin\ i) (Fin\ j) = Fin\ (sup\ i\ j) \mid$   
 $sup\text{-extended}\ a\ Pinf = Pinf \mid$   
 $sup\text{-extended}\ Pinf\ a = Pinf \mid$   
 $sup\text{-extended}\ Minf\ a = a \mid$   
 $sup\text{-extended}\ a\ Minf = a$

**case-of-simps**  $inf\text{-extended}\text{-case}: inf\text{-extended}.simps$

**case-of-simps**  $sup\text{-extended}\text{-case}: sup\text{-extended}.simps$

**instance**

$\langle proof \rangle$

**end**

**end**

## 35 Continuity and iterations

**theory**  $Order\text{-Continuity}$

**imports**  $Complex\text{-Main}\ Countable\text{-Complete}\text{-Lattices}$

**begin**

**lemma**  $SUP\text{-nat}\text{-binary}:$

$(sup\ A\ (SUP\ x \in Collect\ ((<) (0::nat)).\ B)) = (sup\ A\ B::'a::countable\text{-complete}\text{-lattice})$

$\langle proof \rangle$

**lemma**  $INF\text{-nat}\text{-binary}:$

$inf\ A\ (INF\ x \in Collect\ ((<) (0::nat)).\ B) = (inf\ A\ B::'a::countable\text{-complete}\text{-lattice})$

$\langle proof \rangle$

The name *continuous* is already taken in *Complex-Main*, so we use *sup-continuous* and *inf-continuous*. These names appear sometimes in literature and have the advantage that these names are duals.

**named-theorems**  $order\text{-continuous}\text{-intros}$

### 35.1 Continuity for complete lattices

#### definition

*sup-continuous* :: ('a::countable-complete-lattice  $\Rightarrow$  'b::countable-complete-lattice)  
 $\Rightarrow$  bool

#### where

*sup-continuous* F  $\longleftrightarrow$  ( $\forall$  M::nat  $\Rightarrow$  'a. mono M  $\longrightarrow$  F (SUP i. M i) = (SUP i. F (M i)))

**lemma** *sup-continuousD*: *sup-continuous* F  $\Longrightarrow$  mono M  $\Longrightarrow$  F (SUP i::nat. M i) = (SUP i. F (M i))

*<proof>*

**lemma** *sup-continuous-mono*:

mono F **if** *sup-continuous* F

*<proof>*

**lemma** [*order-continuous-intros*]:

**shows** *sup-continuous-const*: *sup-continuous* ( $\lambda$ x. c)

**and** *sup-continuous-id*: *sup-continuous* ( $\lambda$ x. x)

**and** *sup-continuous-apply*: *sup-continuous* ( $\lambda$ f. f x)

**and** *sup-continuous-fun*: ( $\bigwedge$ s. *sup-continuous* ( $\lambda$ x. P x s))  $\Longrightarrow$  *sup-continuous*

P

**and** *sup-continuous-If*: *sup-continuous* F  $\Longrightarrow$  *sup-continuous* G  $\Longrightarrow$  *sup-continuous* ( $\lambda$ f. if C then F f else G f)

*<proof>*

**lemma** *sup-continuous-compose*:

**assumes** f: *sup-continuous* f **and** g: *sup-continuous* g

**shows** *sup-continuous* ( $\lambda$ x. f (g x))

*<proof>*

**lemma** *sup-continuous-sup*[*order-continuous-intros*]:

*sup-continuous* f  $\Longrightarrow$  *sup-continuous* g  $\Longrightarrow$  *sup-continuous* ( $\lambda$ x. sup (f x) (g x))

*<proof>*

**lemma** *sup-continuous-inf*[*order-continuous-intros*]:

**fixes** P Q :: 'a :: countable-complete-lattice  $\Rightarrow$  'b :: countable-complete-distrib-lattice

**assumes** P: *sup-continuous* P **and** Q: *sup-continuous* Q

**shows** *sup-continuous* ( $\lambda$ x. inf (P x) (Q x))

*<proof>*

**lemma** *sup-continuous-and*[*order-continuous-intros*]:

*sup-continuous* P  $\Longrightarrow$  *sup-continuous* Q  $\Longrightarrow$  *sup-continuous* ( $\lambda$ x. P x  $\wedge$  Q x)

*<proof>*

**lemma** *sup-continuous-or*[*order-continuous-intros*]:

*sup-continuous* P  $\Longrightarrow$  *sup-continuous* Q  $\Longrightarrow$  *sup-continuous* ( $\lambda$ x. P x  $\vee$  Q x)

*<proof>*

**lemma** *sup-continuous-lfp*:

**assumes** *sup-continuous F* **shows**  $\text{lfp } F = (\text{SUP } i. (F \hat{\sim} i) \text{ bot})$  (**is**  $\text{lfp } F = ?U$ )  
 ⟨*proof*⟩

**lemma** *lfp-transfer-bounded*:

**assumes**  $P: P \text{ bot} \wedge x. P x \implies P (f x) \wedge M. (\wedge i. P (M i)) \implies P (\text{SUP } i::\text{nat}. M i)$

**assumes**  $\alpha: \wedge M. \text{mono } M \implies (\wedge i::\text{nat}. P (M i)) \implies \alpha (\text{SUP } i. M i) = (\text{SUP } i. \alpha (M i))$

**assumes** *f: sup-continuous f and g: sup-continuous g*

**assumes** [*simp*]:  $\wedge x. P x \implies x \leq \text{lfp } f \implies \alpha (f x) = g (\alpha x)$

**assumes** *g-bound*:  $\wedge x. \alpha \text{ bot} \leq g x$

**shows**  $\alpha (\text{lfp } f) = \text{lfp } g$

⟨*proof*⟩

**lemma** *lfp-transfer*:

*sup-continuous*  $\alpha \implies \text{sup-continuous } f \implies \text{sup-continuous } g \implies$

$(\wedge x. \alpha \text{ bot} \leq g x) \implies (\wedge x. x \leq \text{lfp } f \implies \alpha (f x) = g (\alpha x)) \implies \alpha (\text{lfp } f) = \text{lfp } g$

⟨*proof*⟩

**definition**

*inf-continuous* :: (*a*::*countable-complete-lattice*  $\implies$  *b*::*countable-complete-lattice*)  
 $\implies \text{bool}$

**where**

*inf-continuous F*  $\longleftrightarrow (\forall M::\text{nat} \implies 'a. \text{antimono } M \longrightarrow F (\text{INF } i. M i) = (\text{INF } i. F (M i)))$

**lemma** *inf-continuousD*: *inf-continuous F*  $\implies \text{antimono } M \implies F (\text{INF } i::\text{nat}. M i) = (\text{INF } i. F (M i))$

⟨*proof*⟩

**lemma** *inf-continuous-mono*:

*mono F* **if** *inf-continuous F*

⟨*proof*⟩

**lemma** [*order-continuous-intros*]:

**shows** *inf-continuous-const*: *inf-continuous*  $(\lambda x. c)$

**and** *inf-continuous-id*: *inf-continuous*  $(\lambda x. x)$

**and** *inf-continuous-apply*: *inf-continuous*  $(\lambda f. f x)$

**and** *inf-continuous-fun*:  $(\wedge s. \text{inf-continuous } (\lambda x. P x s)) \implies \text{inf-continuous } P$

**and** *inf-continuous-If*: *inf-continuous F*  $\implies \text{inf-continuous } G \implies \text{inf-continuous } (\lambda f. \text{if } C \text{ then } F f \text{ else } G f)$

⟨*proof*⟩

**lemma** *inf-continuous-inf*[*order-continuous-intros*]:

*inf-continuous f*  $\implies \text{inf-continuous } g \implies \text{inf-continuous } (\lambda x. \text{inf } (f x) (g x))$

⟨*proof*⟩

**lemma** *inf-continuous-sup*[*order-continuous-intros*]:  
**fixes**  $P Q :: 'a :: \text{countable-complete-lattice} \Rightarrow 'b :: \text{countable-complete-distrib-lattice}$   
**assumes**  $P: \text{inf-continuous } P$  **and**  $Q: \text{inf-continuous } Q$   
**shows**  $\text{inf-continuous } (\lambda x. \text{sup } (P x) (Q x))$   
 $\langle \text{proof} \rangle$

**lemma** *inf-continuous-and*[*order-continuous-intros*]:  
 $\text{inf-continuous } P \Longrightarrow \text{inf-continuous } Q \Longrightarrow \text{inf-continuous } (\lambda x. P x \wedge Q x)$   
 $\langle \text{proof} \rangle$

**lemma** *inf-continuous-or*[*order-continuous-intros*]:  
 $\text{inf-continuous } P \Longrightarrow \text{inf-continuous } Q \Longrightarrow \text{inf-continuous } (\lambda x. P x \vee Q x)$   
 $\langle \text{proof} \rangle$

**lemma** *inf-continuous-compose*:  
**assumes**  $f: \text{inf-continuous } f$  **and**  $g: \text{inf-continuous } g$   
**shows**  $\text{inf-continuous } (\lambda x. f (g x))$   
 $\langle \text{proof} \rangle$

**lemma** *inf-continuous-gfp*:  
**assumes**  $\text{inf-continuous } F$  **shows**  $\text{gfp } F = (\text{INF } i. (F \rightsquigarrow i) \text{ top})$  (**is**  $\text{gfp } F = ?U$ )  
 $\langle \text{proof} \rangle$

**lemma** *gfp-transfer*:  
**assumes**  $\alpha: \text{inf-continuous } \alpha$  **and**  $f: \text{inf-continuous } f$  **and**  $g: \text{inf-continuous } g$   
**assumes** [*simp*]:  $\alpha \text{ top} = \text{top} \wedge x. \alpha (f x) = g (\alpha x)$   
**shows**  $\alpha (\text{gfp } f) = \text{gfp } g$   
 $\langle \text{proof} \rangle$

**lemma** *gfp-transfer-bounded*:  
**assumes**  $P: P (f \text{ top}) \wedge x. P x \Longrightarrow P (f x) \wedge M. \text{antimono } M \Longrightarrow (\bigwedge i. P (M i)) \Longrightarrow P (\text{INF } i::\text{nat. } M i)$   
**assumes**  $\alpha: \bigwedge M. \text{antimono } M \Longrightarrow (\bigwedge i::\text{nat. } P (M i)) \Longrightarrow \alpha (\text{INF } i. M i) = (\text{INF } i. \alpha (M i))$   
**assumes**  $f: \text{inf-continuous } f$  **and**  $g: \text{inf-continuous } g$   
**assumes** [*simp*]:  $\bigwedge x. P x \Longrightarrow \alpha (f x) = g (\alpha x)$   
**assumes**  $g\text{-bound}: \bigwedge x. g x \leq \alpha (f \text{ top})$   
**shows**  $\alpha (\text{gfp } f) = \text{gfp } g$   
 $\langle \text{proof} \rangle$

### 35.1.1 Least fixed points in countable complete lattices

**definition** (**in** *countable-complete-lattice*)  $\text{cclfp} :: ('a \Rightarrow 'a) \Rightarrow 'a$   
**where**  $\text{cclfp } f = (\text{SUP } i. (f \rightsquigarrow i) \text{ bot})$

**lemma** *cclfp-unfold*:  
**assumes**  $\text{sup-continuous } F$  **shows**  $\text{cclfp } F = F (\text{cclfp } F)$   
 $\langle \text{proof} \rangle$

**lemma** *cclfp-lowerbound*: **assumes**  $f$ : *mono f* **and**  $A$ :  $f A \leq A$  **shows**  $cclfp f \leq A$   
 ⟨*proof*⟩

**lemma** *cclfp-transfer*:  
**assumes** *sup-continuous  $\alpha$  mono f*  
**assumes**  $\alpha$  *bot = bot*  $\wedge x$ .  $\alpha (f x) = g (\alpha x)$   
**shows**  $\alpha (cclfp f) = cclfp g$   
 ⟨*proof*⟩

**end**

## 36 Extended natural numbers (i.e. with infinity)

**theory** *Extended-Nat*  
**imports** *Main Countable Order-Continuity*  
**begin**

**class** *infinity* =  
**fixes** *infinity* :: 'a ( $\langle \infty \rangle$ )

**context**  
**fixes**  $f$  ::  $nat \Rightarrow 'a::\{canonically-ordered-monoid-add, linorder-topology, complete-linorder\}$   
**begin**

**lemma** *sums-SUP[simp, intro]*:  $f$  *sums* ( $SUP n$ .  $\sum i < n$ .  $f i$ )  
 ⟨*proof*⟩

**lemma** *suminf-eq-SUP*:  $suminf f = (SUP n$ .  $\sum i < n$ .  $f i$ )  
 ⟨*proof*⟩

**end**

### 36.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

**typedef** *enat* = *UNIV* :: *nat option set* ⟨*proof*⟩

TODO: introduce *enat* as coinductive datatype, *enat* is just *of-nat*

**definition** *enat* ::  $nat \Rightarrow enat$  **where**  
 $enat n = Abs-enat (Some n)$

**instantiation** *enat* :: *infinity*  
**begin**

**definition**  $\infty = Abs-enat None$   
**instance** ⟨*proof*⟩

**end**

**instance** *enat* :: *countable*  
 ⟨*proof*⟩

**old-rep-datatype** *enat* ∞ :: *enat*  
 ⟨*proof*⟩

**declare** [[*coercion enat::nat⇒enat*]]

**lemmas** *enat2-cases* = *enat.exhaust*[*case-product enat.exhaust*]

**lemmas** *enat3-cases* = *enat.exhaust*[*case-product enat.exhaust enat.exhaust*]

**lemma** *not-infinity-eq* [*iff*]:  $(x \neq \infty) = (\exists i. x = \text{enat } i)$   
 ⟨*proof*⟩

**lemma** *not-enat-eq* [*iff*]:  $(\forall y. x \neq \text{enat } y) = (x = \infty)$   
 ⟨*proof*⟩

**lemma** *enat-ex-split*:  $(\exists c::\text{enat}. P c) \longleftrightarrow P \infty \vee (\exists c::\text{nat}. P c)$   
 ⟨*proof*⟩

**primrec** *the-enat* :: *enat* ⇒ *nat*  
 where *the-enat* (*enat* *n*) = *n*

## 36.2 Constructors and numbers

**instantiation** *enat* :: *zero-neq-one*  
**begin**

**definition**  
 0 = *enat* 0

**definition**  
 1 = *enat* 1

**instance**  
 ⟨*proof*⟩

**end**

**definition** *eSuc* :: *enat* ⇒ *enat* where  
*eSuc* *i* = (case *i* of *enat* *n* ⇒ *enat* (*Suc* *n*) | ∞ ⇒ ∞)

**lemma** *enat-0* [*code-post*]: *enat* 0 = 0  
 ⟨*proof*⟩

**lemma** *enat-1* [*code-post*]: *enat* 1 = 1

*<proof>*

**lemma** *enat-0-iff*:  $enat\ x = 0 \longleftrightarrow x = 0$   $0 = enat\ x \longleftrightarrow x = 0$   
*<proof>*

**lemma** *enat-1-iff*:  $enat\ x = 1 \longleftrightarrow x = 1$   $1 = enat\ x \longleftrightarrow x = 1$   
*<proof>*

**lemma** *one-eSuc*:  $1 = eSuc\ 0$   
*<proof>*

**lemma** *infinity-ne-i0* [*simp*]:  $(\infty::enat) \neq 0$   
*<proof>*

**lemma** *i0-ne-infinity* [*simp*]:  $0 \neq (\infty::enat)$   
*<proof>*

**lemma** *zero-one-enat-neq*:  
 $\neg 0 = (1::enat)$   
 $\neg 1 = (0::enat)$   
*<proof>*

**lemma** *infinity-ne-i1* [*simp*]:  $(\infty::enat) \neq 1$   
*<proof>*

**lemma** *i1-ne-infinity* [*simp*]:  $1 \neq (\infty::enat)$   
*<proof>*

**lemma** *eSuc-enat*:  $eSuc\ (enat\ n) = enat\ (Suc\ n)$   
*<proof>*

**lemma** *eSuc-infinity* [*simp*]:  $eSuc\ \infty = \infty$   
*<proof>*

**lemma** *eSuc-ne-0* [*simp*]:  $eSuc\ n \neq 0$   
*<proof>*

**lemma** *zero-ne-eSuc* [*simp*]:  $0 \neq eSuc\ n$   
*<proof>*

**lemma** *eSuc-inject* [*simp*]:  $eSuc\ m = eSuc\ n \longleftrightarrow m = n$   
*<proof>*

**lemma** *eSuc-enat-iff*:  $eSuc\ x = enat\ y \longleftrightarrow (\exists n. y = Suc\ n \wedge x = enat\ n)$   
*<proof>*

**lemma** *enat-eSuc-iff*:  $enat\ y = eSuc\ x \longleftrightarrow (\exists n. y = Suc\ n \wedge enat\ n = x)$   
*<proof>*

### 36.3 Addition

**instantiation** *enat* :: *comm-monoid-add*  
**begin**

**definition** [*nitpick-simp*]:

$m + n = (\text{case } m \text{ of } \infty \Rightarrow \infty \mid \text{enat } m \Rightarrow (\text{case } n \text{ of } \infty \Rightarrow \infty \mid \text{enat } n \Rightarrow \text{enat } (m + n)))$

**lemma** *plus-enat-simps* [*simp, code*]:

**fixes**  $q :: \text{enat}$

**shows**  $\text{enat } m + \text{enat } n = \text{enat } (m + n)$

**and**  $\infty + q = \infty$

**and**  $q + \infty = \infty$

*<proof>*

**instance**

*<proof>*

**end**

**lemma** *eSuc-plus-1*:

$e\text{Suc } n = n + 1$

*<proof>*

**lemma** *plus-1-eSuc*:

$1 + q = e\text{Suc } q$

$q + 1 = e\text{Suc } q$

*<proof>*

**lemma** *iadd-Suc*:  $e\text{Suc } m + n = e\text{Suc } (m + n)$

*<proof>*

**lemma** *iadd-Suc-right*:  $m + e\text{Suc } n = e\text{Suc } (m + n)$

*<proof>*

### 36.4 Multiplication

**instantiation** *enat* :: {*comm-semiring-1, semiring-no-zero-divisors*}  
**begin**

**definition** *times-enat-def* [*nitpick-simp*]:

$m * n = (\text{case } m \text{ of } \infty \Rightarrow \text{if } n = 0 \text{ then } 0 \text{ else } \infty \mid \text{enat } m \Rightarrow (\text{case } n \text{ of } \infty \Rightarrow \text{if } m = 0 \text{ then } 0 \text{ else } \infty \mid \text{enat } n \Rightarrow \text{enat } (m * n)))$

**lemma** *times-enat-simps* [*simp, code*]:

$\text{enat } m * \text{enat } n = \text{enat } (m * n)$

$\infty * \infty = (\infty :: \text{enat})$

$\infty * \text{enat } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \infty)$

$\text{enat } m * \infty = (\text{if } m = 0 \text{ then } 0 \text{ else } \infty)$



*<proof>*

**instance**

*<proof>*

**end**

**lemma** *mult-eSuc*:  $eSuc\ m * n = n + m * n$

*<proof>*

**lemma** *mult-eSuc-right*:  $m * eSuc\ n = m + m * n$

*<proof>*

**lemma** *of-nat-eq-enat*:  $of\text{-}nat\ n = enat\ n$

*<proof>*

**instance** *enat* :: *semiring-char-0*

*<proof>*

**lemma** *imult-is-infinity*:  $((a::enat) * b = \infty) = (a = \infty \wedge b \neq 0 \vee b = \infty \wedge a \neq 0)$

*<proof>*

### 36.5 Numerals

**lemma** *numeral-eq-enat*:

$numeral\ k = enat\ (numeral\ k)$

*<proof>*

**lemma** *enat-numeral* [*code-abbrev*]:

$enat\ (numeral\ k) = numeral\ k$

*<proof>*

**lemma** *infinity-ne-numeral* [*simp*]:  $(\infty::enat) \neq numeral\ k$

*<proof>*

**lemma** *numeral-ne-infinity* [*simp*]:  $numeral\ k \neq (\infty::enat)$

*<proof>*

**lemma** *eSuc-numeral* [*simp*]:  $eSuc\ (numeral\ k) = numeral\ (k + Num.One)$

*<proof>*

### 36.6 Subtraction

**instantiation** *enat* :: *minus*

**begin**

**definition** *diff-enat-def*:

$a - b = (case\ a\ of\ (enat\ x) \Rightarrow (case\ b\ of\ (enat\ y) \Rightarrow enat\ (x - y) \mid \infty \Rightarrow 0) \mid \infty \Rightarrow \infty)$

**instance**  $\langle proof \rangle$

**end**

**lemma** *idiff-enat-enat* [*simp*, *code*]:  $enat\ a - enat\ b = enat\ (a - b)$   
 $\langle proof \rangle$

**lemma** *idiff-infinity* [*simp*, *code*]:  $\infty - n = (\infty::enat)$   
 $\langle proof \rangle$

**lemma** *idiff-infinity-right* [*simp*, *code*]:  $enat\ a - \infty = 0$   
 $\langle proof \rangle$

**lemma** *idiff-0* [*simp*]:  $(0::enat) - n = 0$   
 $\langle proof \rangle$

**lemmas** *idiff-enat-0* [*simp*] = *idiff-0* [*unfolded zero-enat-def*]

**lemma** *idiff-0-right* [*simp*]:  $(n::enat) - 0 = n$   
 $\langle proof \rangle$

**lemmas** *idiff-enat-0-right* [*simp*] = *idiff-0-right* [*unfolded zero-enat-def*]

**lemma** *idiff-self* [*simp*]:  $n \neq \infty \implies (n::enat) - n = 0$   
 $\langle proof \rangle$

**lemma** *eSuc-minus-eSuc* [*simp*]:  $eSuc\ n - eSuc\ m = n - m$   
 $\langle proof \rangle$

**lemma** *eSuc-minus-1* [*simp*]:  $eSuc\ n - 1 = n$   
 $\langle proof \rangle$

### 36.7 Ordering

**instantiation** *enat* :: *linordered-ab-semigroup-add*  
**begin**

**definition** [*nitpick-simp*]:  
 $m \leq n = (\text{case } n \text{ of } enat\ n1 \Rightarrow (\text{case } m \text{ of } enat\ m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow \text{False})$   
 $\mid \infty \Rightarrow \text{True})$

**definition** [*nitpick-simp*]:  
 $m < n = (\text{case } m \text{ of } enat\ m1 \Rightarrow (\text{case } n \text{ of } enat\ n1 \Rightarrow m1 < n1 \mid \infty \Rightarrow \text{True})$   
 $\mid \infty \Rightarrow \text{False})$

**lemma** *enat-ord-simps* [*simp*]:  
 $enat\ m \leq enat\ n \iff m \leq n$   
 $enat\ m < enat\ n \iff m < n$

$q \leq (\infty::\text{enat})$   
 $q < (\infty::\text{enat}) \iff q \neq \infty$   
 $(\infty::\text{enat}) \leq q \iff q = \infty$   
 $(\infty::\text{enat}) < q \iff \text{False}$   
 ⟨proof⟩

**lemma** *numeral-le-enat-iff* [simp]:  
**shows**  $\text{numeral } m \leq \text{enat } n \iff \text{numeral } m \leq n$   
 ⟨proof⟩

**lemma** *numeral-less-enat-iff* [simp]:  
**shows**  $\text{numeral } m < \text{enat } n \iff \text{numeral } m < n$   
 ⟨proof⟩

**lemma** *enat-ord-code* [code]:  
 $\text{enat } m \leq \text{enat } n \iff m \leq n$   
 $\text{enat } m < \text{enat } n \iff m < n$   
 $q \leq (\infty::\text{enat}) \iff \text{True}$   
 $\text{enat } m < \infty \iff \text{True}$   
 $\infty \leq \text{enat } n \iff \text{False}$   
 $(\infty::\text{enat}) < q \iff \text{False}$   
 ⟨proof⟩

**instance**  
 ⟨proof⟩

**end**

**instance** *enat :: dioid*  
 ⟨proof⟩

**instance** *enat :: {linordered-nonzero-semiring, strict-ordered-comm-monoid-add}*  
 ⟨proof⟩

**lemma** *add-diff-assoc-enat*:  $z \leq y \implies x + (y - z) = x + y - (z::\text{enat})$   
 ⟨proof⟩

**lemma** *enat-ord-number* [simp]:  
 $(\text{numeral } m :: \text{enat}) \leq \text{numeral } n \iff (\text{numeral } m :: \text{nat}) \leq \text{numeral } n$   
 $(\text{numeral } m :: \text{enat}) < \text{numeral } n \iff (\text{numeral } m :: \text{nat}) < \text{numeral } n$   
 ⟨proof⟩

**lemma** *infinity-ileE* [elim!]:  $\infty \leq \text{enat } m \implies R$   
 ⟨proof⟩

**lemma** *infinity-ilessE* [elim!]:  $\infty < \text{enat } m \implies R$   
 ⟨proof⟩

**lemma** *eSuc-ile-mono* [simp]:  $eSuc\ n \leq eSuc\ m \longleftrightarrow n \leq m$   
 ⟨proof⟩

**lemma** *eSuc-mono* [simp]:  $eSuc\ n < eSuc\ m \longleftrightarrow n < m$   
 ⟨proof⟩

**lemma** *ile-eSuc* [simp]:  $n \leq eSuc\ n$   
 ⟨proof⟩

**lemma** *not-eSuc-ilei0* [simp]:  $\neg eSuc\ n \leq 0$   
 ⟨proof⟩

**lemma** *i0-iless-eSuc* [simp]:  $0 < eSuc\ n$   
 ⟨proof⟩

**lemma** *iless-eSuc0* [simp]:  $(n < eSuc\ 0) = (n = 0)$   
 ⟨proof⟩

**lemma** *ileI1*:  $m < n \implies eSuc\ m \leq n$   
 ⟨proof⟩

**lemma** *Suc-ile-eq*:  $enat\ (Suc\ m) \leq n \longleftrightarrow enat\ m < n$   
 ⟨proof⟩

**lemma** *iless-Suc-eq* [simp]:  $enat\ m < eSuc\ n \longleftrightarrow enat\ m \leq n$   
 ⟨proof⟩

**lemma** *imult-infinity*:  $(0::enat) < n \implies \infty * n = \infty$   
 ⟨proof⟩

**lemma** *imult-infinity-right*:  $(0::enat) < n \implies n * \infty = \infty$   
 ⟨proof⟩

**lemma** *enat-0-less-mult-iff*:  $(0 < (m::enat) * n) = (0 < m \wedge 0 < n)$   
 ⟨proof⟩

**lemma** *mono-eSuc*: *mono* *eSuc*  
 ⟨proof⟩

**lemma** *min-enat-simps* [simp]:  
 $min\ (enat\ m)\ (enat\ n) = enat\ (min\ m\ n)$   
 $min\ q\ 0 = 0$   
 $min\ 0\ q = 0$   
 $min\ q\ (\infty::enat) = q$   
 $min\ (\infty::enat)\ q = q$   
 ⟨proof⟩

**lemma** *max-enat-simps* [simp]:

$\text{max } (\text{enat } m) (\text{enat } n) = \text{enat } (\text{max } m \ n)$   
 $\text{max } q \ 0 = q$   
 $\text{max } 0 \ q = q$   
 $\text{max } q \ \infty = (\infty :: \text{enat})$   
 $\text{max } \infty \ q = (\infty :: \text{enat})$   
 ⟨proof⟩

**lemma** *enat-ile*:  $n \leq \text{enat } m \implies \exists k. n = \text{enat } k$   
 ⟨proof⟩

**lemma** *enat-iless*:  $n < \text{enat } m \implies \exists k. n = \text{enat } k$   
 ⟨proof⟩

**lemma** *iadd-le-enat-iff*:  
 $x + y \leq \text{enat } n \iff (\exists y' \ x'. x = \text{enat } x' \wedge y = \text{enat } y' \wedge x' + y' \leq n)$   
 ⟨proof⟩

**lemma** *chain-incr*:  $\forall i. \exists j. Y \ i < Y \ j \implies \exists j. \text{enat } k < Y \ j$   
 ⟨proof⟩

**lemma** *eSuc-max*:  $e\text{Suc } (\text{max } x \ y) = \text{max } (e\text{Suc } x) (e\text{Suc } y)$   
 ⟨proof⟩

**lemma** *eSuc-Max*:  
**assumes** *finite A*  $A \neq \{\}$   
**shows**  $e\text{Suc } (\text{Max } A) = \text{Max } (e\text{Suc } ` A)$   
 ⟨proof⟩

**instantiation** *enat* ::  $\{\text{order-bot}, \text{order-top}\}$   
**begin**

**definition** *bot-enat* :: *enat* **where** *bot-enat* = 0  
**definition** *top-enat* :: *enat* **where** *top-enat* =  $\infty$

**instance**  
 ⟨proof⟩

**end**

**lemma** *finite-enat-bounded*:  
**assumes** *le-fin*:  $\bigwedge y. y \in A \implies y \leq \text{enat } n$   
**shows** *finite A*  
 ⟨proof⟩

### 36.8 Cancellation simprocs

**lemma** *add-diff-cancel-enat[simp]*:  $x \neq \infty \implies x + y - x = (y :: \text{enat})$   
 ⟨proof⟩

**lemma** *enat-add-left-cancel*:  $a + b = a + c \longleftrightarrow a = (\infty::\text{enat}) \vee b = c$   
 ⟨proof⟩

**lemma** *enat-add-left-cancel-le*:  $a + b \leq a + c \longleftrightarrow a = (\infty::\text{enat}) \vee b \leq c$   
 ⟨proof⟩

**lemma** *enat-add-left-cancel-less*:  $a + b < a + c \longleftrightarrow a \neq (\infty::\text{enat}) \wedge b < c$   
 ⟨proof⟩

**lemma** *plus-eq-infty-iff-enat*:  $(m::\text{enat}) + n = \infty \longleftrightarrow m = \infty \vee n = \infty$   
 ⟨proof⟩

⟨ML⟩

TODO: add regression tests for these simprocs

TODO: add simprocs for combining and cancelling numerals

### 36.9 Well-ordering

**lemma** *less-enatE*:  
 $\llbracket n < \text{enat } m; \bigwedge k. \llbracket n = \text{enat } k; k < m \rrbracket \implies P \rrbracket \implies P$   
 ⟨proof⟩

**lemma** *less-infinityE*:  
 $\llbracket n < \infty; \bigwedge k. n = \text{enat } k \implies P \rrbracket \implies P$   
 ⟨proof⟩

**lemma** *enat-less-induct*:  
 assumes  $\bigwedge n. \forall m::\text{enat}. m < n \longrightarrow P m \implies P n$   
 shows  $P n$   
 ⟨proof⟩

**instance** *enat :: wellorder*  
 ⟨proof⟩

### 36.10 Complete Lattice

**instantiation** *enat :: complete-lattice*  
**begin**

**definition** *inf-enat* ::  $\text{enat} \Rightarrow \text{enat} \Rightarrow \text{enat}$  **where**  
*inf-enat* = *min*

**definition** *sup-enat* ::  $\text{enat} \Rightarrow \text{enat} \Rightarrow \text{enat}$  **where**  
*sup-enat* = *max*

**definition** *Inf-enat* ::  $\text{enat set} \Rightarrow \text{enat}$  **where**  
*Inf-enat*  $A = (\text{if } A = \{\} \text{ then } \infty \text{ else } (\text{LEAST } x. x \in A))$

**definition** *Sup-enat* :: *enat set*  $\Rightarrow$  *enat* **where**

*Sup-enat A* = (if  $A = \{\}$  then 0 else if finite *A* then *Max A* else  $\infty$ )

**instance**

*<proof>*

**end**

**instance** *enat* :: *complete-linorder* *<proof>*

**lemma** *eSuc-Sup*:  $A \neq \{\} \Rightarrow eSuc (Sup A) = Sup (eSuc ` A)$

*<proof>*

**lemma** *sup-continuous-eSuc*: *sup-continuous f*  $\Rightarrow$  *sup-continuous* ( $\lambda x. eSuc (f x)$ )

*<proof>*

### 36.11 Traditional theorem names

**lemmas** *enat-defs* = *zero-enat-def one-enat-def eSuc-def*

*plus-enat-def less-eq-enat-def less-enat-def*

**lemma** *iadd-is-0*:  $(m + n = (0::enat)) = (m = 0 \wedge n = 0)$

*<proof>*

**lemma** *i0-lb* :  $(0::enat) \leq n$

*<proof>*

**lemma** *ile0-eq*:  $n \leq (0::enat) \longleftrightarrow n = 0$

*<proof>*

**lemma** *not-iless0*:  $\neg n < (0::enat)$

*<proof>*

**lemma** *i0-less[simp]*:  $(0::enat) < n \longleftrightarrow n \neq 0$

*<proof>*

**lemma** *imult-is-0*:  $((m::enat) * n = 0) = (m = 0 \vee n = 0)$

*<proof>*

**end**

## 37 Liminf and Limsup on conditionally complete lattices

**theory** *Liminf-Limsup*

**imports** *Complex-Main*

**begin**

**lemma** (in *conditionally-complete-linorder*) *le-cSup-iff*:

**assumes**  $A \neq \{\}$  *bdd-above*  $A$

**shows**  $x \leq \text{Sup } A \iff (\forall y < x. \exists a \in A. y < a)$

*<proof>*

**lemma** (in *conditionally-complete-linorder*) *le-cSUP-iff*:

$A \neq \{\} \implies \text{bdd-above } (f' A) \implies x \leq \text{Sup } (f' A) \iff (\forall y < x. \exists i \in A. y < f i)$

*<proof>*

**lemma** *le-cSup-iff-less*:

**fixes**  $x :: 'a :: \{\text{conditionally-complete-linorder, dense-linorder}\}$

**shows**  $A \neq \{\} \implies \text{bdd-above } (f' A) \implies x \leq (\text{SUP } i \in A. f i) \iff (\forall y < x. \exists i \in A. y \leq f i)$

*<proof>*

**lemma** *le-Sup-iff-less*:

**fixes**  $x :: 'a :: \{\text{complete-linorder, dense-linorder}\}$

**shows**  $x \leq (\text{SUP } i \in A. f i) \iff (\forall y < x. \exists i \in A. y \leq f i)$  (is ?lhs = ?rhs)

*<proof>*

**lemma** (in *conditionally-complete-linorder*) *cInf-le-iff*:

**assumes**  $A \neq \{\}$  *bdd-below*  $A$

**shows**  $\text{Inf } A \leq x \iff (\forall y > x. \exists a \in A. y > a)$

*<proof>*

**lemma** (in *conditionally-complete-linorder*) *cINF-le-iff*:

$A \neq \{\} \implies \text{bdd-below } (f' A) \implies \text{Inf } (f' A) \leq x \iff (\forall y > x. \exists i \in A. y > f i)$

*<proof>*

**lemma** *cInf-le-iff-less*:

**fixes**  $x :: 'a :: \{\text{conditionally-complete-linorder, dense-linorder}\}$

**shows**  $A \neq \{\} \implies \text{bdd-below } (f' A) \implies (\text{INF } i \in A. f i) \leq x \iff (\forall y > x. \exists i \in A. f i \leq y)$

*<proof>*

**lemma** *Inf-le-iff-less*:

**fixes**  $x :: 'a :: \{\text{complete-linorder, dense-linorder}\}$

**shows**  $(\text{INF } i \in A. f i) \leq x \iff (\forall y > x. \exists i \in A. f i \leq y)$

*<proof>*

**lemma** *SUP-pair*:

**fixes**  $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$

**shows**  $(\text{SUP } i \in A. \text{SUP } j \in B. f i j) = (\text{SUP } p \in A \times B. f (\text{fst } p) (\text{snd } p))$

*<proof>*

**lemma** *INF-pair*:

**fixes**  $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$

**shows**  $(\text{INF } i \in A. \text{INF } j \in B. f i j) = (\text{INF } p \in A \times B. f (\text{fst } p) (\text{snd } p))$

*<proof>*



**lemma** *INF-Sigma*:

**fixes**  $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$

**shows**  $(\text{INF } i \in A. \text{INF } j \in B \ i. f \ i \ j) = (\text{INF } p \in \text{Sigma } A \ B. f \ (\text{fst } p) \ (\text{snd } p))$   
 $\langle \text{proof} \rangle$

### 37.0.1 *Liminf and Limsup*

**definition** *Liminf* ::  $'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b :: \text{complete-lattice}$  **where**

$\text{Liminf } F \ f = (\text{SUP } P \in \{P. \text{eventually } P \ F\}. \text{INF } x \in \{x. P \ x\}. f \ x)$

**definition** *Limsup* ::  $'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b :: \text{complete-lattice}$  **where**

$\text{Limsup } F \ f = (\text{INF } P \in \{P. \text{eventually } P \ F\}. \text{SUP } x \in \{x. P \ x\}. f \ x)$

**abbreviation** *liminf*  $\equiv \text{Liminf sequentially}$

**abbreviation** *limsup*  $\equiv \text{Limsup sequentially}$

**lemma** *Liminf-eqI*:

$(\bigwedge P. \text{eventually } P \ F \Longrightarrow \text{Inf } (f \ ' (\text{Collect } P)) \leq x) \Longrightarrow$   
 $(\bigwedge y. (\bigwedge P. \text{eventually } P \ F \Longrightarrow \text{Inf } (f \ ' (\text{Collect } P)) \leq y) \Longrightarrow x \leq y) \Longrightarrow \text{Liminf}$   
 $F \ f = x$   
 $\langle \text{proof} \rangle$

**lemma** *Limsup-eqI*:

$(\bigwedge P. \text{eventually } P \ F \Longrightarrow x \leq \text{Sup } (f \ ' (\text{Collect } P))) \Longrightarrow$   
 $(\bigwedge y. (\bigwedge P. \text{eventually } P \ F \Longrightarrow y \leq \text{Sup } (f \ ' (\text{Collect } P))) \Longrightarrow y \leq x) \Longrightarrow$   
 $\text{Limsup } F \ f = x$   
 $\langle \text{proof} \rangle$

**lemma** *liminf-SUP-INF*:  $\text{liminf } f = (\text{SUP } n. \text{INF } m \in \{n..\}. f \ m)$

$\langle \text{proof} \rangle$

**lemma** *limsup-INF-SUP*:  $\text{limsup } f = (\text{INF } n. \text{SUP } m \in \{n..\}. f \ m)$

$\langle \text{proof} \rangle$

**lemma** *mem-limsup-iff*:  $x \in \text{limsup } A \longleftrightarrow (\exists_F \ n \ \text{in sequentially. } x \in A \ n)$

$\langle \text{proof} \rangle$

**lemma** *mem-liminf-iff*:  $x \in \text{liminf } A \longleftrightarrow (\forall_F \ n \ \text{in sequentially. } x \in A \ n)$

$\langle \text{proof} \rangle$

**lemma** *Limsup-const*:

**assumes** *ntriv*:  $\neg \text{trivial-limit } F$

**shows**  $\text{Limsup } F \ (\lambda x. c) = c$

$\langle \text{proof} \rangle$

**lemma** *Liminf-const*:

**assumes** *ntriv*:  $\neg \text{trivial-limit } F$

**shows**  $\text{Liminf } F (\lambda x. c) = c$   
 ⟨proof⟩

**lemma** *Liminf-mono*:

**assumes**  $ev$ : eventually  $(\lambda x. f x \leq g x)$   $F$   
**shows**  $\text{Liminf } F f \leq \text{Liminf } F g$   
 ⟨proof⟩

**lemma** *Liminf-eq*:

**assumes** eventually  $(\lambda x. f x = g x)$   $F$   
**shows**  $\text{Liminf } F f = \text{Liminf } F g$   
 ⟨proof⟩

**lemma** *Limsup-mono*:

**assumes**  $ev$ : eventually  $(\lambda x. f x \leq g x)$   $F$   
**shows**  $\text{Limsup } F f \leq \text{Limsup } F g$   
 ⟨proof⟩

**lemma** *Limsup-eq*:

**assumes** eventually  $(\lambda x. f x = g x)$   $net$   
**shows**  $\text{Limsup } net f = \text{Limsup } net g$   
 ⟨proof⟩

**lemma** *Liminf-bot[simp]*:  $\text{Liminf } bot f = top$   
 ⟨proof⟩

**lemma** *Limsup-bot[simp]*:  $\text{Limsup } bot f = bot$   
 ⟨proof⟩

**lemma** *Liminf-le-Limsup*:

**assumes**  $ntriv$ :  $\neg$  trivial-limit  $F$   
**shows**  $\text{Liminf } F f \leq \text{Limsup } F f$   
 ⟨proof⟩

**lemma** *Liminf-bounded*:

**assumes**  $le$ : eventually  $(\lambda n. C \leq X n)$   $F$   
**shows**  $C \leq \text{Liminf } F X$   
 ⟨proof⟩

**lemma** *Limsup-bounded*:

**assumes**  $le$ : eventually  $(\lambda n. X n \leq C)$   $F$   
**shows**  $\text{Limsup } F X \leq C$   
 ⟨proof⟩

**lemma** *le-Limsup*:

**assumes**  $F$ :  $F \neq bot$  and  $x$ :  $\forall_F x$  in  $F. l \leq f x$   
**shows**  $l \leq \text{Limsup } F f$   
 ⟨proof⟩

**lemma** *Liminf-le*:

**assumes**  $F: F \neq \text{bot}$  **and**  $x: \forall_F x \text{ in } F. f x \leq l$

**shows**  $\text{Liminf } F f \leq l$

*<proof>*

**lemma** *le-Liminf-iff*:

**fixes**  $X :: - \Rightarrow - :: \text{complete-linorder}$

**shows**  $C \leq \text{Liminf } F X \iff (\forall y < C. \text{eventually } (\lambda x. y < X x) F)$

*<proof>*

**lemma** *Limsup-le-iff*:

**fixes**  $X :: - \Rightarrow - :: \text{complete-linorder}$

**shows**  $C \geq \text{Limsup } F X \iff (\forall y > C. \text{eventually } (\lambda x. y > X x) F)$

*<proof>*

**lemma** *less-LiminfD*:

$y < \text{Liminf } F (f :: - \Rightarrow 'a :: \text{complete-linorder}) \implies \text{eventually } (\lambda x. f x > y) F$

*<proof>*

**lemma** *Limsup-lessD*:

$y > \text{Limsup } F (f :: - \Rightarrow 'a :: \text{complete-linorder}) \implies \text{eventually } (\lambda x. f x < y) F$

*<proof>*

**lemma** *lim-imp-Liminf*:

**fixes**  $f :: 'a \Rightarrow - :: \{\text{complete-linorder}, \text{linorder-topology}\}$

**assumes**  $\text{ntriv}: \neg \text{trivial-limit } F$

**assumes**  $\text{lim}: (f \longrightarrow f0) F$

**shows**  $\text{Liminf } F f = f0$

*<proof>*

**lemma** *lim-imp-Limsup*:

**fixes**  $f :: 'a \Rightarrow - :: \{\text{complete-linorder}, \text{linorder-topology}\}$

**assumes**  $\text{ntriv}: \neg \text{trivial-limit } F$

**assumes**  $\text{lim}: (f \longrightarrow f0) F$

**shows**  $\text{Limsup } F f = f0$

*<proof>*

**lemma** *Liminf-eq-Limsup*:

**fixes**  $f0 :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$

**assumes**  $\text{ntriv}: \neg \text{trivial-limit } F$

**and**  $\text{lim}: \text{Liminf } F f = f0 \text{ Limsup } F f = f0$

**shows**  $(f \longrightarrow f0) F$

*<proof>*

**lemma** *tendsto-iff-Liminf-eq-Limsup*:

**fixes**  $f0 :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$

**shows**  $\neg \text{trivial-limit } F \implies (f \longrightarrow f0) F \iff (\text{Liminf } F f = f0 \wedge \text{Limsup } F f = f0)$

*<proof>*

**lemma** *liminf-subseq-mono*:

**fixes**  $X :: \text{nat} \Rightarrow 'a :: \text{complete-linorder}$

**assumes** *strict-mono*  $r$

**shows**  $\text{liminf } X \leq \text{liminf } (X \circ r)$

*<proof>*

**lemma** *limsup-subseq-mono*:

**fixes**  $X :: \text{nat} \Rightarrow 'a :: \text{complete-linorder}$

**assumes** *strict-mono*  $r$

**shows**  $\text{limsup } (X \circ r) \leq \text{limsup } X$

*<proof>*

**lemma** *continuous-on-imp-continuous-within*:

*continuous-on*  $s \ f \Longrightarrow t \subseteq s \Longrightarrow x \in s \Longrightarrow \text{continuous (at } x \text{ within } t) \ f$

*<proof>*

**lemma** *Liminf-compose-continuous-mono*:

**fixes**  $f :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b :: \{\text{complete-linorder}, \text{linorder-topology}\}$

**assumes**  $c$ : *continuous-on UNIV*  $f$  **and**  $am$ : *mono*  $f$  **and**  $F$ :  $F \neq \text{bot}$

**shows**  $\text{Liminf } F (\lambda n. f (g \ n)) = f (\text{Liminf } F \ g)$

*<proof>*

**lemma** *Limsup-compose-continuous-mono*:

**fixes**  $f :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b :: \{\text{complete-linorder}, \text{linorder-topology}\}$

**assumes**  $c$ : *continuous-on UNIV*  $f$  **and**  $am$ : *mono*  $f$  **and**  $F$ :  $F \neq \text{bot}$

**shows**  $\text{Limsup } F (\lambda n. f (g \ n)) = f (\text{Limsup } F \ g)$

*<proof>*

**lemma** *Liminf-compose-continuous-antimono*:

**fixes**  $f :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b :: \{\text{complete-linorder}, \text{linorder-topology}\}$

**assumes**  $c$ : *continuous-on UNIV*  $f$

**and**  $am$ : *antimono*  $f$

**and**  $F$ :  $F \neq \text{bot}$

**shows**  $\text{Liminf } F (\lambda n. f (g \ n)) = f (\text{Limsup } F \ g)$

*<proof>*

**lemma** *Limsup-compose-continuous-antimono*:

**fixes**  $f :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b :: \{\text{complete-linorder}, \text{linorder-topology}\}$

**assumes**  $c$ : *continuous-on UNIV*  $f$  **and**  $am$ : *antimono*  $f$  **and**  $F$ :  $F \neq \text{bot}$

**shows**  $\text{Limsup } F (\lambda n. f (g \ n)) = f (\text{Liminf } F \ g)$

*<proof>*

**lemma** *Liminf-filtermap-le*:  $\text{Liminf } (\text{filtermap } f \ F) \ g \leq \text{Liminf } F (\lambda x. g (f \ x))$

*<proof>*

**lemma** *Limsup-filtermap-ge*:  $\text{Limsup } (\text{filtermap } f \ F) \ g \geq \text{Limsup } F (\lambda x. g (f \ x))$

*<proof>*

**lemma** *Liminf-least*:  $(\bigwedge P. \text{eventually } P \ F \implies (\text{INF } x \in \text{Collect } P. f \ x) \leq x) \implies \text{Liminf } F \ f \leq x$   
 ⟨proof⟩

**lemma** *Limsup-greatest*:  $(\bigwedge P. \text{eventually } P \ F \implies x \leq (\text{SUP } x \in \text{Collect } P. f \ x)) \implies \text{Limsup } F \ f \geq x$   
 ⟨proof⟩

**lemma** *Liminf-filtermap-ge*:  $\text{inj } f \implies \text{Liminf } (\text{filtermap } f \ F) \ g \geq \text{Liminf } F \ (\lambda x. g \ (f \ x))$   
 ⟨proof⟩

**lemma** *Limsup-filtermap-le*:  $\text{inj } f \implies \text{Limsup } (\text{filtermap } f \ F) \ g \leq \text{Limsup } F \ (\lambda x. g \ (f \ x))$   
 ⟨proof⟩

**lemma** *Liminf-filtermap-eq*:  $\text{inj } f \implies \text{Liminf } (\text{filtermap } f \ F) \ g = \text{Liminf } F \ (\lambda x. g \ (f \ x))$   
 ⟨proof⟩

**lemma** *Limsup-filtermap-eq*:  $\text{inj } f \implies \text{Limsup } (\text{filtermap } f \ F) \ g = \text{Limsup } F \ (\lambda x. g \ (f \ x))$   
 ⟨proof⟩

### 37.1 More Limits

**lemma** *convergent-limsup-cl*:  
**fixes**  $X :: \text{nat} \Rightarrow 'a::\{\text{complete-linorder}, \text{linorder-topology}\}$   
**shows**  $\text{convergent } X \implies \text{limsup } X = \text{lim } X$   
 ⟨proof⟩

**lemma** *convergent-liminf-cl*:  
**fixes**  $X :: \text{nat} \Rightarrow 'a::\{\text{complete-linorder}, \text{linorder-topology}\}$   
**shows**  $\text{convergent } X \implies \text{liminf } X = \text{lim } X$   
 ⟨proof⟩

**lemma** *lim-increasing-cl*:  
**assumes**  $\bigwedge n \ m. n \geq m \implies f \ n \geq f \ m$   
**obtains**  $l$  **where**  $f \longrightarrow (l::'a::\{\text{complete-linorder}, \text{linorder-topology}\})$   
 ⟨proof⟩

**lemma** *lim-decreasing-cl*:  
**assumes**  $\bigwedge n \ m. n \geq m \implies f \ n \leq f \ m$   
**obtains**  $l$  **where**  $f \longrightarrow (l::'a::\{\text{complete-linorder}, \text{linorder-topology}\})$   
 ⟨proof⟩

**lemma** *compact-complete-linorder*:  
**fixes**  $X :: \text{nat} \Rightarrow 'a::\{\text{complete-linorder}, \text{linorder-topology}\}$   
**shows**  $\exists l \ r. \text{strict-mono } r \wedge (X \circ r) \longrightarrow l$

*<proof>*

**lemma** *tendsto-Limsup*:

**fixes**  $f :: - \Rightarrow 'a :: \{complete-linorder, linorder-topology\}$

**shows**  $F \neq bot \implies Limsup F f = Liminf F f \implies (f \longrightarrow Limsup F f) F$

*<proof>*

**lemma** *tendsto-Liminf*:

**fixes**  $f :: - \Rightarrow 'a :: \{complete-linorder, linorder-topology\}$

**shows**  $F \neq bot \implies Limsup F f = Liminf F f \implies (f \longrightarrow Liminf F f) F$

*<proof>*

**end**

## 38 Extended real number line

**theory** *Extended-Real*

**imports** *Complex-Main Extended-Nat Liminf-Limsup*

**begin**

This should be part of *HOL-Library.Extended-Nat* or *HOL-Library.Order-Continuity*, but then the AFP-entry *Jinja-Thread* fails, as it does overload certain named from *Complex-Main*.

**lemma** *incseq-sumI2*:

**fixes**  $f :: 'i \Rightarrow nat \Rightarrow 'a::ordered-comm-monoid-add$

**shows**  $(\bigwedge n. n \in A \implies mono (f n)) \implies mono (\lambda i. \sum_{n \in A} f n i)$

*<proof>*

**lemma** *incseq-sumI*:

**fixes**  $f :: nat \Rightarrow 'a::ordered-comm-monoid-add$

**assumes**  $\bigwedge i. 0 \leq f i$

**shows**  $incseq (\lambda i. sum f \{..< i\})$

*<proof>*

**lemma** *continuous-at-left-imp-sup-continuous*:

**fixes**  $f :: 'a::\{complete-linorder, linorder-topology\} \Rightarrow 'b::\{complete-linorder, linorder-topology\}$

**assumes**  $mono f \wedge x. continuous (at-left x) f$

**shows**  $sup-continuous f$

*<proof>*

**lemma** *sup-continuous-at-left*:

**fixes**  $f :: 'a::\{complete-linorder, linorder-topology, first-countable-topology\} \Rightarrow$

$'b::\{complete-linorder, linorder-topology\}$

**assumes**  $f: sup-continuous f$

**shows**  $continuous (at-left x) f$

*<proof>*

**lemma** *sup-continuous-iff-at-left*:

**fixes**  $f :: 'a::\{complete-linorder, linorder-topology, first-countable-topology\} \Rightarrow$

```

      'b::{complete-linorder, linorder-topology}
shows sup-continuous f  $\longleftrightarrow$  ( $\forall x.$  continuous (at-left x) f)  $\wedge$  mono f
    <proof>

lemma continuous-at-right-imp-inf-continuous:
fixes f :: 'a::{complete-linorder, linorder-topology}  $\Rightarrow$  'b::{complete-linorder, linorder-topology}
assumes mono f  $\wedge$  x. continuous (at-right x) f
shows inf-continuous f
    <proof>

lemma inf-continuous-at-right:
fixes f :: 'a::{complete-linorder, linorder-topology, first-countable-topology}  $\Rightarrow$ 
      'b::{complete-linorder, linorder-topology}
assumes f: inf-continuous f
shows continuous (at-right x) f
    <proof>

lemma inf-continuous-iff-at-right:
fixes f :: 'a::{complete-linorder, linorder-topology, first-countable-topology}  $\Rightarrow$ 
      'b::{complete-linorder, linorder-topology}
shows inf-continuous f  $\longleftrightarrow$  ( $\forall x.$  continuous (at-right x) f)  $\wedge$  mono f
    <proof>

instantiation enat :: linorder-topology
begin

definition open-enat :: enat set  $\Rightarrow$  bool where
  open-enat = generate-topology (range lessThan  $\cup$  range greaterThan)

instance
  <proof>

end

lemma open-enat: open {enat n}
  <proof>

lemma open-enat-iff:
fixes A :: enat set
shows open A  $\longleftrightarrow$  ( $\infty \in A \longrightarrow (\exists n::nat. \{n <..\} \subseteq A)$ )
  <proof>

lemma nhds-enat: nhds x = (if x =  $\infty$  then INF i. principal {enat i..} else principal
  {x})
  <proof>

instance enat :: topological-comm-monoid-add
  <proof>

```

For more lemmas about the extended real numbers see `~/src/HOL/`

Analysis/Extended\_Real\_Limits.thy.

### 38.1 Definition and basic properties

**datatype** *ereal* = *ereal real* | *PInfty* | *MInfty*

**instantiation** *ereal* :: *uminus*  
**begin**

**fun** *uminus-ereal* **where**  
  – (*ereal r*) = *ereal* (– *r*)  
  | – *PInfty* = *MInfty*  
  | – *MInfty* = *PInfty*

**instance**  $\langle$ *proof* $\rangle$

**end**

**instantiation** *ereal* :: *infinity*  
**begin**

**definition** ( $\infty$ ::*ereal*) = *PInfty*  
**instance**  $\langle$ *proof* $\rangle$

**end**

**declare** [[*coercion ereal* :: *real*  $\Rightarrow$  *ereal*]]

**lemma** *ereal-uminus-uminus*[*simp*]:  
  **fixes** *a* :: *ereal*  
  **shows** – (– *a*) = *a*  
   $\langle$ *proof* $\rangle$

**lemma**  
  **shows** *PInfty-eq-infinity*[*simp*]: *PInfty* =  $\infty$   
  **and** *MInfty-eq-minfinity*[*simp*]: *MInfty* =  $-\infty$   
  **and** *MInfty-neq-PInfty*[*simp*]:  $\infty \neq -(\infty$ ::*ereal*)  $-\infty \neq (\infty$ ::*ereal*)  
  **and** *MInfty-neq-ereal*[*simp*]: *ereal r*  $\neq -\infty$   $-\infty \neq$  *ereal r*  
  **and** *PInfty-neq-ereal*[*simp*]: *ereal r*  $\neq \infty$   $\infty \neq$  *ereal r*  
  **and** *PInfty-cases*[*simp*]: (case  $\infty$  of *ereal r*  $\Rightarrow$  *f r* | *PInfty*  $\Rightarrow$  *y* | *MInfty*  $\Rightarrow$  *z*)  
  = *y*  
  **and** *MInfty-cases*[*simp*]: (case  $-\infty$  of *ereal r*  $\Rightarrow$  *f r* | *PInfty*  $\Rightarrow$  *y* | *MInfty*  $\Rightarrow$   
  *z*) = *z*  
   $\langle$ *proof* $\rangle$

**declare**  
  *PInfty-eq-infinity*[*code-post*]  
  *MInfty-eq-minfinity*[*code-post*]



**lemma** *[code-unfold]*:

$\infty = PInfty$   
 $- PInfty = MInfty$   
 $\langle proof \rangle$

**lemma** *inj-ereal[simp]*: *inj-on ereal A*

$\langle proof \rangle$

**lemma** *ereal-cases[cases type: ereal]*:

**obtains** *(real) r where x = ereal r*  
 $| (PInf) x = \infty$   
 $| (MInf) x = -\infty$   
 $\langle proof \rangle$

**lemmas** *ereal2-cases = ereal-cases[case-product ereal-cases]*

**lemmas** *ereal3-cases = ereal2-cases[case-product ereal-cases]*

**lemma** *ereal-all-split*:  $\bigwedge P. (\forall x::ereal. P x) \longleftrightarrow P \infty \wedge (\forall x. P (ereal x)) \wedge P$   
 $(-\infty)$

$\langle proof \rangle$

**lemma** *ereal-ex-split*:  $\bigwedge P. (\exists x::ereal. P x) \longleftrightarrow P \infty \vee (\exists x. P (ereal x)) \vee P$   
 $(-\infty)$

$\langle proof \rangle$

**lemma** *ereal-uminus-eq-iff[simp]*:

**fixes** *a b :: ereal*  
**shows**  $-a = -b \longleftrightarrow a = b$   
 $\langle proof \rangle$

**function** *real-of-ereal :: ereal  $\Rightarrow$  real where*

*real-of-ereal (ereal r) = r*  
 $| real-of-ereal \infty = 0$   
 $| real-of-ereal (-\infty) = 0$   
 $\langle proof \rangle$

**termination**  $\langle proof \rangle$

**lemma** *real-of-ereal[simp]*:

*real-of-ereal (- x :: ereal) = - (real-of-ereal x)*  
 $\langle proof \rangle$

**lemma** *range-ereal[simp]*: *range ereal = UNIV - { $\infty, -\infty$ }*

$\langle proof \rangle$

**lemma** *ereal-range-uminus[simp]*: *range uminus = (UNIV::ereal set)*

$\langle proof \rangle$

**instantiation** *ereal :: abs*

**begin**

```

function abs-ereal where
  | ereal r | = ereal |r|
  | |-∞ | = (∞::ereal)
  | |∞ | = (∞::ereal)
  <proof>
termination <proof>

instance <proof>

end

lemma abs-eq-infinity-cases[elim!]:
  fixes x :: ereal
  assumes |x| = ∞
  obtains x = ∞ | x = -∞
  <proof>

lemma abs-neq-infinity-cases[elim!]:
  fixes x :: ereal
  assumes |x| ≠ ∞
  obtains r where x = ereal r
  <proof>

lemma abs-ereal-uminus[simp]:
  fixes x :: ereal
  shows |- x | = |x|
  <proof>

lemma ereal-infinity-cases:
  fixes a :: ereal
  shows a ≠ ∞ ⇒ a ≠ -∞ ⇒ |a| ≠ ∞
  <proof>

```

### 38.1.1 Addition

```

instantiation ereal :: {one,comm-monoid-add,zero-neq-one}
begin

```

```

definition 0 = ereal 0

```

```

definition 1 = ereal 1

```

```

function plus-ereal where
  ereal r + ereal p = ereal (r + p)
  | ∞ + a = (∞::ereal)
  | a + ∞ = (∞::ereal)
  | ereal r + -∞ = -∞
  | -∞ + ereal p = -(∞::ereal)
  | -∞ + -∞ = -(∞::ereal)

```

*<proof>*

**termination** *<proof>*

**lemma** *Infty-neq-0[simp]*:

$(\infty::ereal) \neq 0 \ 0 \neq (\infty::ereal)$

$-(\infty::ereal) \neq 0 \ 0 \neq -(\infty::ereal)$

*<proof>*

**lemma** *ereal-eq-0[simp]*:

$ereal \ r = 0 \longleftrightarrow r = 0$

$0 = ereal \ r \longleftrightarrow r = 0$

*<proof>*

**lemma** *ereal-eq-1[simp]*:

$ereal \ r = 1 \longleftrightarrow r = 1$

$1 = ereal \ r \longleftrightarrow r = 1$

*<proof>*

**instance**

*<proof>*

**end**

**lemma** *ereal-0-plus [simp]*:  $ereal \ 0 + x = x$

**and** *plus-ereal-0 [simp]*:  $x + ereal \ 0 = x$

*<proof>*

**instance** *ereal :: numeral* *<proof>*

**lemma** *real-of-ereal-0[simp]*:  $real-of-ereal \ (0::ereal) = 0$

*<proof>*

**lemma** *abs-ereal-zero[simp]*:  $|0| = (0::ereal)$

*<proof>*

**lemma** *ereal-uminus-zero[simp]*:  $- 0 = (0::ereal)$

*<proof>*

**lemma** *ereal-uminus-zero-iff[simp]*:

**fixes**  $a :: ereal$

**shows**  $-a = 0 \longleftrightarrow a = 0$

*<proof>*

**lemma** *ereal-plus-eq-PIInfty[simp]*:

**fixes**  $a \ b :: ereal$

**shows**  $a + b = \infty \longleftrightarrow a = \infty \vee b = \infty$

*<proof>*

**lemma** *ereal-plus-eq-MInfty[simp]*:

**fixes**  $a\ b :: \text{ereal}$   
**shows**  $a + b = -\infty \longleftrightarrow (a = -\infty \vee b = -\infty) \wedge a \neq \infty \wedge b \neq \infty$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-add-cancel-left*:  
**fixes**  $a\ b :: \text{ereal}$   
**assumes**  $a \neq -\infty$   
**shows**  $a + b = a + c \longleftrightarrow a = \infty \vee b = c$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-add-cancel-right*:  
**fixes**  $a\ b :: \text{ereal}$   
**assumes**  $a \neq -\infty$   
**shows**  $b + a = c + a \longleftrightarrow a = \infty \vee b = c$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-real*:  $\text{ereal} (\text{real-of-ereal } x) = (\text{if } |x| = \infty \text{ then } 0 \text{ else } x)$   
 $\langle \text{proof} \rangle$

**lemma** *real-of-ereal-add*:  
**fixes**  $a\ b :: \text{ereal}$   
**shows**  $\text{real-of-ereal} (a + b) =$   
 $(\text{if } (|a| = \infty) \wedge (|b| = \infty) \vee (|a| \neq \infty) \wedge (|b| \neq \infty) \text{ then } \text{real-of-ereal } a +$   
 $\text{real-of-ereal } b \text{ else } 0)$   
 $\langle \text{proof} \rangle$

### 38.1.2 Linear order on *ereal*

**instantiation**  $\text{ereal} :: \text{linorder}$   
**begin**

**function** *less-ereal*  
**where**  
 $\text{ereal } x < \text{ereal } y \quad \longleftrightarrow x < y$   
 $| (\infty :: \text{ereal}) < a \quad \longleftrightarrow \text{False}$   
 $| a < -(\infty :: \text{ereal}) \quad \longleftrightarrow \text{False}$   
 $| \text{ereal } x < \infty \quad \longleftrightarrow \text{True}$   
 $| -\infty < \text{ereal } r \quad \longleftrightarrow \text{True}$   
 $| -\infty < (\infty :: \text{ereal}) \quad \longleftrightarrow \text{True}$   
 $\langle \text{proof} \rangle$   
**termination**  $\langle \text{proof} \rangle$

**definition**  $x \leq (y :: \text{ereal}) \longleftrightarrow x < y \vee x = y$

**lemma** *ereal-infy-less[simp]*:  
**fixes**  $x :: \text{ereal}$   
**shows**  $x < \infty \longleftrightarrow (x \neq \infty)$   
 $-\infty < x \longleftrightarrow (x \neq -\infty)$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-infity-less-eq[simp]*:  
**fixes**  $x :: \text{ereal}$   
**shows**  $\infty \leq x \longleftrightarrow x = \infty$   
**and**  $x \leq -\infty \longleftrightarrow x = -\infty$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-less[simp]*:  
 $\text{ereal } r < 0 \longleftrightarrow (r < 0)$   
 $0 < \text{ereal } r \longleftrightarrow (0 < r)$   
 $\text{ereal } r < 1 \longleftrightarrow (r < 1)$   
 $1 < \text{ereal } r \longleftrightarrow (1 < r)$   
 $0 < (\infty :: \text{ereal})$   
 $-(\infty :: \text{ereal}) < 0$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-less-eq[simp]*:  
 $x \leq (\infty :: \text{ereal})$   
 $-(\infty :: \text{ereal}) \leq x$   
 $\text{ereal } r \leq \text{ereal } p \longleftrightarrow r \leq p$   
 $\text{ereal } r \leq 0 \longleftrightarrow r \leq 0$   
 $0 \leq \text{ereal } r \longleftrightarrow 0 \leq r$   
 $\text{ereal } r \leq 1 \longleftrightarrow r \leq 1$   
 $1 \leq \text{ereal } r \longleftrightarrow 1 \leq r$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-infity-less-eq2*:  
 $a \leq b \implies a = \infty \implies b = (\infty :: \text{ereal})$   
 $a \leq b \implies b = -\infty \implies a = -(\infty :: \text{ereal})$   
 $\langle \text{proof} \rangle$

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

**end**

**lemma** *ereal-dense2*:  $x < y \implies \exists z. x < \text{ereal } z \wedge \text{ereal } z < y$   
 $\langle \text{proof} \rangle$

**instance** *ereal :: dense-linorder*  
 $\langle \text{proof} \rangle$

**instance** *ereal :: ordered-comm-monoid-add*  
 $\langle \text{proof} \rangle$

**lemma** *ereal-one-not-less-zero-ereal[simp]*:  $\neg 1 < (0 :: \text{ereal})$   
 $\langle \text{proof} \rangle$

**lemma** *real-of-ereal-positive-mono*:

**fixes**  $x y :: \text{ereal}$   
**shows**  $0 \leq x \implies x \leq y \implies y \neq \infty \implies \text{real-of-ereal } x \leq \text{real-of-ereal } y$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-MInfty-lessI*[*intro, simp*]:

**fixes**  $a :: \text{ereal}$   
**shows**  $a \neq -\infty \implies -\infty < a$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-less-PInfty*[*intro, simp*]:

**fixes**  $a :: \text{ereal}$   
**shows**  $a \neq \infty \implies a < \infty$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-less-ereal-Ex*:

**fixes**  $a b :: \text{ereal}$   
**shows**  $x < \text{ereal } r \longleftrightarrow x = -\infty \vee (\exists p. p < r \wedge x = \text{ereal } p)$   
 $\langle \text{proof} \rangle$

**lemma** *less-PInf-Ex-of-nat*:  $x \neq \infty \longleftrightarrow (\exists n::\text{nat}. x < \text{ereal } (\text{real } n))$

$\langle \text{proof} \rangle$

**lemma** *ereal-add-strict-mono2*:

**fixes**  $a b c d :: \text{ereal}$   
**assumes**  $a < b$  **and**  $c < d$   
**shows**  $a + c < b + d$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-minus-le-minus*[*simp*]:

**fixes**  $a b :: \text{ereal}$   
**shows**  $-a \leq -b \longleftrightarrow b \leq a$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-minus-less-minus*[*simp*]:

**fixes**  $a b :: \text{ereal}$   
**shows**  $-a < -b \longleftrightarrow b < a$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-le-real-iff*:

$x \leq \text{real-of-ereal } y \longleftrightarrow (|y| \neq \infty \longrightarrow \text{ereal } x \leq y) \wedge (|y| = \infty \longrightarrow x \leq 0)$   
 $\langle \text{proof} \rangle$

**lemma** *real-le-ereal-iff*:

$\text{real-of-ereal } y \leq x \longleftrightarrow (|y| \neq \infty \longrightarrow y \leq \text{ereal } x) \wedge (|y| = \infty \longrightarrow 0 \leq x)$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-less-real-iff*:

$x < \text{real-of-ereal } y \longleftrightarrow (|y| \neq \infty \longrightarrow \text{ereal } x < y) \wedge (|y| = \infty \longrightarrow x < 0)$   
 $\langle \text{proof} \rangle$

**lemma** *real-less-ereal-iff*:

*real-of-ereal*  $y < x \longleftrightarrow (|y| \neq \infty \longrightarrow y < \text{ereal } x) \wedge (|y| = \infty \longrightarrow 0 < x)$   
 ⟨proof⟩

To help with inferences like  $\llbracket a < \text{ereal } x; x < y \rrbracket \Longrightarrow a < \text{ereal } y$ , where  $x$  and  $y$  are real.

**lemma** *le-ereal-le*:  $a \leq \text{ereal } x \Longrightarrow x \leq y \Longrightarrow a \leq \text{ereal } y$   
 ⟨proof⟩

**lemma** *le-ereal-less*:  $a \leq \text{ereal } x \Longrightarrow x < y \Longrightarrow a < \text{ereal } y$   
 ⟨proof⟩

**lemma** *less-ereal-le*:  $a < \text{ereal } x \Longrightarrow x \leq y \Longrightarrow a < \text{ereal } y$   
 ⟨proof⟩

**lemma** *ereal-le-le*:  $\text{ereal } y \leq a \Longrightarrow x \leq y \Longrightarrow \text{ereal } x \leq a$   
 ⟨proof⟩

**lemma** *ereal-le-less*:  $\text{ereal } y \leq a \Longrightarrow x < y \Longrightarrow \text{ereal } x < a$   
 ⟨proof⟩

**lemma** *ereal-less-le*:  $\text{ereal } y < a \Longrightarrow x \leq y \Longrightarrow \text{ereal } x < a$   
 ⟨proof⟩

**lemma** *real-of-ereal-pos*:

**fixes**  $x :: \text{ereal}$

**shows**  $0 \leq x \Longrightarrow 0 \leq \text{real-of-ereal } x$

⟨proof⟩

**lemmas** *real-of-ereal-ord-simps* =

*ereal-le-real-iff* *real-le-ereal-iff* *ereal-less-real-iff* *real-less-ereal-iff*

**lemma** *abs-ereal-ge0[simp]*:  $0 \leq x \Longrightarrow |x :: \text{ereal}| = x$   
 ⟨proof⟩

**lemma** *abs-ereal-less0[simp]*:  $x < 0 \Longrightarrow |x :: \text{ereal}| = -x$   
 ⟨proof⟩

**lemma** *abs-ereal-pos[simp]*:  $0 \leq |x :: \text{ereal}|$   
 ⟨proof⟩

**lemma** *ereal-abs-leI*:

**fixes**  $x y :: \text{ereal}$

**shows**  $\llbracket x \leq y; -x \leq y \rrbracket \Longrightarrow |x| \leq y$

⟨proof⟩

**lemma** *ereal-abs-add*:

**fixes**  $a b :: \text{ereal}$

**shows**  $\text{abs}(a+b) \leq \text{abs } a + \text{abs } b$   
 ⟨proof⟩

**lemma** *real-of-ereal-le-0[simp]*:  $\text{real-of-ereal } (x :: \text{ereal}) \leq 0 \longleftrightarrow x \leq 0 \vee x = \infty$   
 ⟨proof⟩

**lemma** *abs-real-of-ereal[simp]*:  $|\text{real-of-ereal } (x :: \text{ereal})| = \text{real-of-ereal } |x|$   
 ⟨proof⟩

**lemma** *zero-less-real-of-ereal*:  
**fixes**  $x :: \text{ereal}$   
**shows**  $0 < \text{real-of-ereal } x \longleftrightarrow 0 < x \wedge x \neq \infty$   
 ⟨proof⟩

**lemma** *ereal-0-le-uminus-iff[simp]*:  
**fixes**  $a :: \text{ereal}$   
**shows**  $0 \leq -a \longleftrightarrow a \leq 0$   
 ⟨proof⟩

**lemma** *ereal-uminus-le-0-iff[simp]*:  
**fixes**  $a :: \text{ereal}$   
**shows**  $-a \leq 0 \longleftrightarrow 0 \leq a$   
 ⟨proof⟩

**lemma** *ereal-add-strict-mono*:  
**fixes**  $a b c d :: \text{ereal}$   
**assumes**  $a \leq b$   
**and**  $0 \leq a$   
**and**  $a \neq \infty$   
**and**  $c < d$   
**shows**  $a + c < b + d$   
 ⟨proof⟩

**lemma** *ereal-less-add*:  
**fixes**  $a b c :: \text{ereal}$   
**shows**  $|a| \neq \infty \implies c < b \implies a + c < a + b$   
 ⟨proof⟩

**lemma** *ereal-uminus-eq-reorder*:  $-a = b \longleftrightarrow a = (-b :: \text{ereal})$   
 ⟨proof⟩

**lemma** *ereal-uminus-less-reorder*:  $-a < b \longleftrightarrow -b < a$   
**and** *ereal-less-uminus-reorder*:  $a < -b \longleftrightarrow b < -a$   
**and** *ereal-uminus-le-reorder*:  $-a \leq b \longleftrightarrow -b \leq a$  **for**  $a :: \text{ereal}$   
 ⟨proof⟩

**lemmas** *ereal-uminus-reorder =*  
*ereal-uminus-eq-reorder* *ereal-uminus-less-reorder* *ereal-uminus-le-reorder*



**lemma** *ereal-bot*:

**fixes**  $x :: \text{ereal}$   
**assumes**  $\bigwedge B. x \leq \text{ereal } B$   
**shows**  $x = -\infty$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-top*:

**fixes**  $x :: \text{ereal}$   
**assumes**  $\bigwedge B. x \geq \text{ereal } B$   
**shows**  $x = \infty$   
 $\langle \text{proof} \rangle$

**lemma**

**shows** *ereal-max[simp]*:  $\text{ereal } (\max x y) = \max (\text{ereal } x) (\text{ereal } y)$   
**and** *ereal-min[simp]*:  $\text{ereal } (\min x y) = \min (\text{ereal } x) (\text{ereal } y)$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-max-0*:  $\max 0 (\text{ereal } r) = \text{ereal } (\max 0 r)$

$\langle \text{proof} \rangle$

**lemma**

**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**shows** *ereal-incseq-uminus[simp]*:  $\text{incseq } (\lambda x. - f x) \longleftrightarrow \text{decseq } f$   
**and** *ereal-decseq-uminus[simp]*:  $\text{decseq } (\lambda x. - f x) \longleftrightarrow \text{incseq } f$   
 $\langle \text{proof} \rangle$

**lemma** *incseq-ereal*:  $\text{incseq } f \Longrightarrow \text{incseq } (\lambda x. \text{ereal } (f x))$

$\langle \text{proof} \rangle$

**lemma** *sum-ereal[simp]*:  $(\sum x \in A. \text{ereal } (f x)) = \text{ereal } (\sum x \in A. f x)$

$\langle \text{proof} \rangle$

**lemma** *sum-list-ereal [simp]*:  $\text{sum-list } (\text{map } (\lambda x. \text{ereal } (f x)) xs) = \text{ereal } (\text{sum-list } (\text{map } f xs))$

$\langle \text{proof} \rangle$

**lemma** *sum-Pinfity*:

**fixes**  $f :: 'a \Rightarrow \text{ereal}$   
**shows**  $(\sum x \in P. f x) = \infty \longleftrightarrow \text{finite } P \wedge (\exists i \in P. f i = \infty)$   
 $\langle \text{proof} \rangle$

**lemma** *sum-Inf*:

**fixes**  $f :: 'a \Rightarrow \text{ereal}$   
**shows**  $|\text{sum } f A| = \infty \longleftrightarrow \text{finite } A \wedge (\exists i \in A. |f i| = \infty)$   
 $\langle \text{proof} \rangle$

**lemma** *sum-real-of-ereal*:

**fixes**  $f :: 'i \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge x. x \in S \Longrightarrow |f x| \neq \infty$

**shows**  $(\sum_{x \in S}. \text{real-of-ereal } (f x)) = \text{real-of-ereal } (\text{sum } f S)$   
 ⟨proof⟩

### 38.1.3 Multiplication

**instantiation** *ereal* :: {*comm-monoid-mult*,*sgn*}  
**begin**

**function** *sgn-ereal* :: *ereal* ⇒ *ereal* **where**

*sgn* (*ereal* *r*) = *ereal* (*sgn* *r*)  
 | *sgn* ( $\infty::\text{ereal}$ ) = 1  
 | *sgn* ( $-\infty::\text{ereal}$ ) = -1  
 ⟨proof⟩

**termination** ⟨proof⟩

**function** *times-ereal* **where**

*ereal* *r* \* *ereal* *p* = *ereal* (*r* \* *p*)  
 | *ereal* *r* \*  $\infty$  = (if *r* = 0 then 0 else if *r* > 0 then  $\infty$  else  $-\infty$ )  
 |  $\infty$  \* *ereal* *r* = (if *r* = 0 then 0 else if *r* > 0 then  $\infty$  else  $-\infty$ )  
 | *ereal* *r* \*  $-\infty$  = (if *r* = 0 then 0 else if *r* > 0 then  $-\infty$  else  $\infty$ )  
 |  $-\infty$  \* *ereal* *r* = (if *r* = 0 then 0 else if *r* > 0 then  $-\infty$  else  $\infty$ )  
 | ( $\infty::\text{ereal}$ ) \*  $\infty$  =  $\infty$   
 |  $-\infty::\text{ereal}$  \*  $\infty$  =  $-\infty$   
 | ( $\infty::\text{ereal}$ ) \*  $-\infty$  =  $-\infty$   
 |  $-\infty::\text{ereal}$  \*  $-\infty$  =  $\infty$   
 ⟨proof⟩

**termination** ⟨proof⟩

**instance**

⟨proof⟩

**end**

**lemma** [*simp*]:

**shows** *ereal-1-times*: *ereal* 1 \* *x* = *x*  
**and** *times-ereal-1*: *x* \* *ereal* 1 = *x*  
 ⟨proof⟩

**lemma** *one-not-le-zero-ereal*[*simp*]:  $\neg (1 \leq (0::\text{ereal}))$   
 ⟨proof⟩

**lemma** *real-ereal-1*[*simp*]: *real-of-ereal* (1::*ereal*) = 1  
 ⟨proof⟩

**lemma** *real-of-ereal-le-1*:

**fixes** *a* :: *ereal*  
**shows**  $a \leq 1 \implies \text{real-of-ereal } a \leq 1$   
 ⟨proof⟩

**lemma** *abs-ereal-one[simp]*:  $|1| = (1::ereal)$   
 ⟨proof⟩

**lemma** *ereal-mult-zero[simp]*:  
**fixes**  $a :: ereal$   
**shows**  $a * 0 = 0$   
 ⟨proof⟩

**lemma** *ereal-zero-mult[simp]*:  
**fixes**  $a :: ereal$   
**shows**  $0 * a = 0$   
 ⟨proof⟩

**lemma** *ereal-m1-less-0[simp]*:  $-(1::ereal) < 0$   
 ⟨proof⟩

**lemma** *ereal-times[simp]*:  
 $1 \neq (\infty::ereal)$   $(\infty::ereal) \neq 1$   
 $1 \neq -(\infty::ereal)$   $-(\infty::ereal) \neq 1$   
 ⟨proof⟩

**lemma** *ereal-plus-1[simp]*:  
 $1 + ereal\ r = ereal\ (r + 1)$   
 $ereal\ r + 1 = ereal\ (r + 1)$   
 $1 + -(\infty::ereal) = -\infty$   
 $-(\infty::ereal) + 1 = -\infty$   
 ⟨proof⟩

**lemma** *ereal-zero-times[simp]*:  
**fixes**  $a\ b :: ereal$   
**shows**  $a * b = 0 \longleftrightarrow a = 0 \vee b = 0$   
 ⟨proof⟩

**lemma** *ereal-mult-eq-PIfty[simp]*:  
 $a * b = (\infty::ereal) \longleftrightarrow$   
 $(a = \infty \wedge b > 0) \vee (a > 0 \wedge b = \infty) \vee (a = -\infty \wedge b < 0) \vee (a < 0 \wedge b =$   
 $-\infty)$   
 ⟨proof⟩

**lemma** *ereal-mult-eq-MIfty[simp]*:  
 $a * b = -(\infty::ereal) \longleftrightarrow$   
 $(a = \infty \wedge b < 0) \vee (a < 0 \wedge b = \infty) \vee (a = -\infty \wedge b > 0) \vee (a > 0 \wedge b =$   
 $-\infty)$   
 ⟨proof⟩

**lemma** *ereal-abs-mult*:  $|x * y :: ereal| = |x| * |y|$   
 ⟨proof⟩

**lemma** *ereal-0-less-1[simp]*:  $0 < (1::ereal)$

*<proof>*

**lemma** *ereal-mult-minus-left[simp]*:

**fixes**  $a\ b :: \text{ereal}$

**shows**  $-a * b = -(a * b)$

*<proof>*

**lemma** *ereal-mult-minus-right[simp]*:

**fixes**  $a\ b :: \text{ereal}$

**shows**  $a * -b = -(a * b)$

*<proof>*

**lemma** *ereal-mult-infty[simp]*:

$a * (\infty :: \text{ereal}) = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } \infty \text{ else } -\infty)$

*<proof>*

**lemma** *ereal-infty-mult[simp]*:

$(\infty :: \text{ereal}) * a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } \infty \text{ else } -\infty)$

*<proof>*

**lemma** *ereal-mult-strict-right-mono*:

**assumes**  $a < b$

**and**  $0 < c$

**and**  $c < (\infty :: \text{ereal})$

**shows**  $a * c < b * c$

*<proof>*

**lemma** *ereal-mult-strict-left-mono*:

$a < b \implies 0 < c \implies c < (\infty :: \text{ereal}) \implies c * a < c * b$

*<proof>*

**lemma** *ereal-mult-right-mono*:

**fixes**  $a\ b\ c :: \text{ereal}$

**assumes**  $a \leq b\ 0 \leq c$

**shows**  $a * c \leq b * c$

*<proof>*

**lemma** *ereal-mult-left-mono*:

**fixes**  $a\ b\ c :: \text{ereal}$

**shows**  $a \leq b \implies 0 \leq c \implies c * a \leq c * b$

*<proof>*

**lemma** *ereal-mult-mono*:

**fixes**  $a\ b\ c\ d :: \text{ereal}$

**assumes**  $b \geq 0\ c \geq 0\ a \leq b\ c \leq d$

**shows**  $a * c \leq b * d$

*<proof>*

**lemma** *ereal-mult-mono'*:

**fixes**  $a\ b\ c\ d::ereal$   
**assumes**  $a \geq 0\ c \geq 0\ a \leq b\ c \leq d$   
**shows**  $a * c \leq b * d$   
 $\langle proof \rangle$

**lemma** *ereal-mult-mono-strict*:  
**fixes**  $a\ b\ c\ d::ereal$   
**assumes**  $b > 0\ c > 0\ a < b\ c < d$   
**shows**  $a * c < b * d$   
 $\langle proof \rangle$

**lemma** *ereal-mult-mono-strict'*:  
**fixes**  $a\ b\ c\ d::ereal$   
**assumes**  $a > 0\ c > 0\ a < b\ c < d$   
**shows**  $a * c < b * d$   
 $\langle proof \rangle$

**lemma** *zero-less-one-ereal[simp]*:  $0 \leq (1::ereal)$   
 $\langle proof \rangle$

**lemma** *ereal-0-le-mult[simp]*:  $0 \leq a \implies 0 \leq b \implies 0 \leq a * (b :: ereal)$   
 $\langle proof \rangle$

**lemma** *ereal-right-distrib*:  
**fixes**  $r\ a\ b :: ereal$   
**shows**  $0 \leq a \implies 0 \leq b \implies r * (a + b) = r * a + r * b$   
 $\langle proof \rangle$

**lemma** *ereal-left-distrib*:  
**fixes**  $r\ a\ b :: ereal$   
**shows**  $0 \leq a \implies 0 \leq b \implies (a + b) * r = a * r + b * r$   
 $\langle proof \rangle$

**lemma** *ereal-mult-le-0-iff*:  
**fixes**  $a\ b :: ereal$   
**shows**  $a * b \leq 0 \iff (0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b)$   
 $\langle proof \rangle$

**lemma** *ereal-zero-le-0-iff*:  
**fixes**  $a\ b :: ereal$   
**shows**  $0 \leq a * b \iff (0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0)$   
 $\langle proof \rangle$

**lemma** *ereal-mult-less-0-iff*:  
**fixes**  $a\ b :: ereal$   
**shows**  $a * b < 0 \iff (0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b)$   
 $\langle proof \rangle$

**lemma** *ereal-zero-less-0-iff*:

**fixes**  $a\ b :: \text{ereal}$   
**shows**  $0 < a * b \iff (0 < a \wedge 0 < b) \vee (a < 0 \wedge b < 0)$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-left-mult-cong*:  
**fixes**  $a\ b\ c :: \text{ereal}$   
**shows**  $c = d \implies (d \neq 0 \implies a = b) \implies a * c = b * d$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-right-mult-cong*:  
**fixes**  $a\ b\ c :: \text{ereal}$   
**shows**  $c = d \implies (d \neq 0 \implies a = b) \implies c * a = d * b$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-distrib*:  
**fixes**  $a\ b\ c :: \text{ereal}$   
**assumes**  $a \neq \infty \vee b \neq -\infty$   
**and**  $a \neq -\infty \vee b \neq \infty$   
**and**  $|c| \neq \infty$   
**shows**  $(a + b) * c = a * c + b * c$   
 $\langle \text{proof} \rangle$

**lemma** *numeral-eq-ereal* [simp]: *numeral*  $w = \text{ereal}$  (*numeral*  $w$ )  
 $\langle \text{proof} \rangle$

**lemma** *distrib-left-ereal-nn*:  
 $c \geq 0 \implies (x + y) * \text{ereal } c = x * \text{ereal } c + y * \text{ereal } c$   
 $\langle \text{proof} \rangle$

**lemma** *sum-ereal-right-distrib*:  
**fixes**  $f :: 'a \Rightarrow \text{ereal}$   
**shows**  $(\bigwedge i. i \in A \implies 0 \leq f\ i) \implies r * \text{sum } f\ A = (\sum n \in A. r * f\ n)$   
 $\langle \text{proof} \rangle$

**lemma** *sum-ereal-left-distrib*:  
 $(\bigwedge i. i \in A \implies 0 \leq f\ i) \implies \text{sum } f\ A * r = (\sum n \in A. f\ n * r :: \text{ereal})$   
 $\langle \text{proof} \rangle$

**lemma** *sum-distrib-right-ereal*:  
 $c \geq 0 \implies \text{sum } f\ A * \text{ereal } c = (\sum x \in A. f\ x * c :: \text{ereal})$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-le-epsilon*:  
**fixes**  $x\ y :: \text{ereal}$   
**assumes**  $\bigwedge e. 0 < e \implies x \leq y + e$   
**shows**  $x \leq y$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-le-epsilon2*:

**fixes**  $x y :: \text{ereal}$   
**assumes**  $\bigwedge e :: \text{real}. 0 < e \implies x \leq y + \text{ereal } e$   
**shows**  $x \leq y$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-le-real*:  
**fixes**  $x y :: \text{ereal}$   
**assumes**  $\bigwedge z. x \leq \text{ereal } z \implies y \leq \text{ereal } z$   
**shows**  $y \leq x$   
 $\langle \text{proof} \rangle$

**lemma** *prod-ereal-0*:  
**fixes**  $f :: 'a \Rightarrow \text{ereal}$   
**shows**  $(\prod_{i \in A}. f i) = 0 \iff \text{finite } A \wedge (\exists i \in A. f i = 0)$   
 $\langle \text{proof} \rangle$

**lemma** *prod-ereal-pos*:  
**fixes**  $f :: 'a \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge i. i \in I \implies 0 \leq f i$   
**shows**  $0 \leq (\prod_{i \in I}. f i)$   
 $\langle \text{proof} \rangle$

**lemma** *prod-PInf*:  
**fixes**  $f :: 'a \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge i. i \in I \implies 0 \leq f i$   
**shows**  $(\prod_{i \in I}. f i) = \infty \iff \text{finite } I \wedge (\exists i \in I. f i = \infty) \wedge (\forall i \in I. f i \neq 0)$   
 $\langle \text{proof} \rangle$

**lemma** *prod-ereal*:  $(\prod_{i \in A}. \text{ereal } (f i)) = \text{ereal } (\text{prod } f A)$   
 $\langle \text{proof} \rangle$

### 38.1.4 Power

**lemma** *ereal-power[simp]*:  $(\text{ereal } x) ^ n = \text{ereal } (x ^ n)$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-power-PInf[simp]*:  $(\infty :: \text{ereal}) ^ n = (\text{if } n = 0 \text{ then } 1 \text{ else } \infty)$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-power-uminus[simp]*:  
**fixes**  $x :: \text{ereal}$   
**shows**  $(- x) ^ n = (\text{if even } n \text{ then } x ^ n \text{ else } - (x ^ n))$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-power-numeral[simp]*:  
 $(\text{numeral } \text{num} :: \text{ereal}) ^ n = \text{ereal } (\text{numeral } \text{num} ^ n)$   
 $\langle \text{proof} \rangle$

**lemma** *zero-le-power-ereal[simp]*:

**fixes**  $a :: \text{ereal}$   
**assumes**  $0 \leq a$   
**shows**  $0 \leq a \wedge n$   
 $\langle \text{proof} \rangle$

### 38.1.5 Subtraction

**lemma** *ereal-minus-minus-image*[simp]:  
**fixes**  $S :: \text{ereal set}$   
**shows**  $\text{uminus } ' S = S$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-uminus-lessThan*[simp]:  
**fixes**  $a :: \text{ereal}$   
**shows**  $\text{uminus } ' \{..<a\} = \{-a<..\}$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-uminus-greaterThan*[simp]:  $\text{uminus } ' \{(a::\text{ereal})<..\} = \{..<-a\}$   
 $\langle \text{proof} \rangle$

**instantiation**  $\text{ereal} :: \text{minus}$   
**begin**

**definition**  $x - y = x + -(y::\text{ereal})$   
**instance**  $\langle \text{proof} \rangle$

**end**

**lemma** *ereal-minus*[simp]:  
 $\text{ereal } r - \text{ereal } p = \text{ereal } (r - p)$   
 $-\infty - \text{ereal } r = -\infty$   
 $\text{ereal } r - \infty = -\infty$   
 $(\infty::\text{ereal}) - x = \infty$   
 $-(\infty::\text{ereal}) - \infty = -\infty$   
 $x - -y = x + y$   
 $x - 0 = x$   
 $0 - x = -x$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-x-minus-x*[simp]:  $x - x = (\text{if } |x| = \infty \text{ then } \infty \text{ else } 0::\text{ereal})$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-eq-minus-iff*:  
**fixes**  $x y z :: \text{ereal}$   
**shows**  $x = z - y \longleftrightarrow$   
 $(|y| \neq \infty \longrightarrow x + y = z) \wedge$   
 $(y = -\infty \longrightarrow x = \infty) \wedge$   
 $(y = \infty \longrightarrow z = \infty \longrightarrow x = \infty) \wedge$   
 $(y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty)$



*<proof>*

**lemma** *ereal-eq-minus*:

**fixes**  $x\ y\ z :: \text{ereal}$

**shows**  $|y| \neq \infty \implies x = z - y \longleftrightarrow x + y = z$

*<proof>*

**lemma** *ereal-less-minus-iff*:

**fixes**  $x\ y\ z :: \text{ereal}$

**shows**  $x < z - y \longleftrightarrow$

$(y = \infty \longrightarrow z = \infty \wedge x \neq \infty) \wedge$

$(y = -\infty \longrightarrow x \neq \infty) \wedge$

$(|y| \neq \infty \longrightarrow x + y < z)$

*<proof>*

**lemma** *ereal-less-minus*:

**fixes**  $x\ y\ z :: \text{ereal}$

**shows**  $|y| \neq \infty \implies x < z - y \longleftrightarrow x + y < z$

*<proof>*

**lemma** *ereal-le-minus-iff*:

**fixes**  $x\ y\ z :: \text{ereal}$

**shows**  $x \leq z - y \longleftrightarrow (y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty) \wedge (|y| \neq \infty \longrightarrow x + y \leq z)$

*<proof>*

**lemma** *ereal-le-minus*:

**fixes**  $x\ y\ z :: \text{ereal}$

**shows**  $|y| \neq \infty \implies x \leq z - y \longleftrightarrow x + y \leq z$

*<proof>*

**lemma** *ereal-minus-less-iff*:

**fixes**  $x\ y\ z :: \text{ereal}$

**shows**  $x - y < z \longleftrightarrow y \neq -\infty \wedge (y = \infty \longrightarrow x \neq \infty \wedge z \neq -\infty) \wedge (y \neq \infty \longrightarrow x < z + y)$

*<proof>*

**lemma** *ereal-minus-less*:

**fixes**  $x\ y\ z :: \text{ereal}$

**shows**  $|y| \neq \infty \implies x - y < z \longleftrightarrow x < z + y$

*<proof>*

**lemma** *ereal-minus-le-iff*:

**fixes**  $x\ y\ z :: \text{ereal}$

**shows**  $x - y \leq z \longleftrightarrow$

$(y = -\infty \longrightarrow z = \infty) \wedge$

$(y = \infty \longrightarrow x = \infty \longrightarrow z = \infty) \wedge$

$(|y| \neq \infty \longrightarrow x \leq z + y)$

*<proof>*

**lemma** *ereal-minus-le*:

**fixes**  $x y z :: \text{ereal}$

**shows**  $|y| \neq \infty \implies x - y \leq z \longleftrightarrow x \leq z + y$

*<proof>*

**lemma** *ereal-minus-eq-minus-iff*:

**fixes**  $a b c :: \text{ereal}$

**shows**  $a - b = a - c \longleftrightarrow$

$b = c \vee a = \infty \vee (a = -\infty \wedge b \neq -\infty \wedge c \neq -\infty)$

*<proof>*

**lemma** *ereal-add-le-add-iff*:

**fixes**  $a b c :: \text{ereal}$

**shows**  $c + a \leq c + b \longleftrightarrow$

$a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$

*<proof>*

**lemma** *ereal-add-le-add-iff2*:

**fixes**  $a b c :: \text{ereal}$

**shows**  $a + c \leq b + c \longleftrightarrow a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$

*<proof>*

**lemma** *ereal-mult-le-mult-iff*:

**fixes**  $a b c :: \text{ereal}$

**shows**  $|c| \neq \infty \implies c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$

*<proof>*

**lemma** *ereal-minus-mono*:

**fixes**  $A B C D :: \text{ereal}$  **assumes**  $A \leq B \ D \leq C$

**shows**  $A - C \leq B - D$

*<proof>*

**lemma** *ereal-mono-minus-cancel*:

**fixes**  $a b c :: \text{ereal}$

**shows**  $c - a \leq c - b \implies 0 \leq c \implies c < \infty \implies b \leq a$

*<proof>*

**lemma** *real-of-ereal-minus*:

**fixes**  $a b :: \text{ereal}$

**shows**  $\text{real-of-ereal } (a - b) = (\text{if } |a| = \infty \vee |b| = \infty \text{ then } 0 \text{ else } \text{real-of-ereal } a - \text{real-of-ereal } b)$

*<proof>*

**lemma** *real-of-ereal-minus'*:  $|x| = \infty \longleftrightarrow |y| = \infty \implies \text{real-of-ereal } x - \text{real-of-ereal } y = \text{real-of-ereal } (x - y :: \text{ereal})$

*<proof>*

**lemma** *ereal-diff-positive*:

**fixes**  $a\ b :: \text{ereal}$  **shows**  $a \leq b \implies 0 \leq b - a$   
 ⟨proof⟩

**lemma** *ereal-between*:

**fixes**  $x\ e :: \text{ereal}$   
**assumes**  $|x| \neq \infty$  **and**  $0 < e$   
**shows**  $x - e < x$   
**and**  $x < x + e$   
 ⟨proof⟩

**lemma** *ereal-minus-eq-PIfty-iff*:

**fixes**  $x\ y :: \text{ereal}$   
**shows**  $x - y = \infty \longleftrightarrow y = -\infty \vee x = \infty$   
 ⟨proof⟩

**lemma** *ereal-diff-add-eq-diff-diff-swap*:

**fixes**  $x\ y\ z :: \text{ereal}$   
**shows**  $|y| \neq \infty \implies x - (y + z) = x - y - z$   
 ⟨proof⟩

**lemma** *ereal-diff-add-assoc2*:

**fixes**  $x\ y\ z :: \text{ereal}$   
**shows**  $x + y - z = x - z + y$   
 ⟨proof⟩

**lemma** *ereal-add-uminus-conv-diff*: **fixes**  $x\ y\ z :: \text{ereal}$  **shows**  $-x + y = y - x$   
 ⟨proof⟩

**lemma** *ereal-minus-diff-eq*:

**fixes**  $x\ y :: \text{ereal}$   
**shows**  $\llbracket x = \infty \longrightarrow y \neq \infty; x = -\infty \longrightarrow y \neq -\infty \rrbracket \implies -(x - y) = y - x$   
 ⟨proof⟩

**lemma** *ediff-le-self* [simp]:  $x - y \leq (x :: \text{enat})$

⟨proof⟩

**lemma** *ereal-abs-diff*:

**fixes**  $a\ b :: \text{ereal}$   
**shows**  $\text{abs}(a - b) \leq \text{abs}\ a + \text{abs}\ b$   
 ⟨proof⟩

### 38.1.6 Division

**instantiation** *ereal* :: *inverse*

**begin**

**function** *inverse-ereal* **where**

$\text{inverse}\ (\text{ereal}\ r) = (\text{if}\ r = 0\ \text{then}\ \infty\ \text{else}\ \text{ereal}\ (\text{inverse}\ r))$   
 |  $\text{inverse}\ (\infty :: \text{ereal}) = 0$

|  $inverse (-\infty::ereal) = 0$   
 ⟨proof⟩

**termination** ⟨proof⟩

**definition**  $x \text{ div } y = x * inverse (y :: ereal)$

**instance** ⟨proof⟩

**end**

**lemma** *real-of-ereal-inverse[simp]*:

**fixes**  $a :: ereal$

**shows**  $real-of-ereal (inverse a) = 1 / real-of-ereal a$   
 ⟨proof⟩

**lemma** *ereal-inverse[simp]*:

$inverse (0::ereal) = \infty$

$inverse (1::ereal) = 1$

⟨proof⟩

**lemma** *ereal-divide[simp]*:

$ereal r / ereal p = (if p = 0 then ereal r * \infty else ereal (r / p))$

⟨proof⟩

**lemma** *ereal-divide-same[simp]*:

**fixes**  $x :: ereal$

**shows**  $x / x = (if |x| = \infty \vee x = 0 then 0 else 1)$

⟨proof⟩

**lemma** *ereal-inv-inv[simp]*:

**fixes**  $x :: ereal$

**shows**  $inverse (inverse x) = (if x \neq -\infty then x else \infty)$

⟨proof⟩

**lemma** *ereal-inverse-minus[simp]*:

**fixes**  $x :: ereal$

**shows**  $inverse (-x) = (if x = 0 then \infty else -inverse x)$

⟨proof⟩

**lemma** *ereal-uminus-divide[simp]*:

**fixes**  $x y :: ereal$

**shows**  $-x / y = -(x / y)$

⟨proof⟩

**lemma** *ereal-divide-Infty[simp]*:

**fixes**  $x :: ereal$

**shows**  $x / \infty = 0 \ x / -\infty = 0$

⟨proof⟩

**lemma** *ereal-divide-one[simp]*:  $x / 1 = (x :: \text{ereal})$   
 ⟨proof⟩

**lemma** *ereal-divide-ereal[simp]*:  $\infty / \text{ereal } r = (\text{if } 0 \leq r \text{ then } \infty \text{ else } -\infty)$   
 ⟨proof⟩

**lemma** *ereal-inverse-nonneg-iff*:  $0 \leq \text{inverse } (x :: \text{ereal}) \longleftrightarrow 0 \leq x \vee x = -\infty$   
 ⟨proof⟩

**lemma** *inverse-ereal-ge0I*:  $0 \leq (x :: \text{ereal}) \implies 0 \leq \text{inverse } x$   
 ⟨proof⟩

**lemma** *zero-le-divide-ereal[simp]*:  
 fixes  $a :: \text{ereal}$   
 assumes  $0 \leq a$  and  $0 \leq b$   
 shows  $0 \leq a / b$   
 ⟨proof⟩

**lemma** *ereal-le-divide-pos*:  
 fixes  $x y z :: \text{ereal}$   
 shows  $x > 0 \implies x \neq \infty \implies y \leq z / x \longleftrightarrow x * y \leq z$   
 ⟨proof⟩

**lemma** *ereal-divide-le-pos*:  
 fixes  $x y z :: \text{ereal}$   
 shows  $x > 0 \implies x \neq \infty \implies z / x \leq y \longleftrightarrow z \leq x * y$   
 ⟨proof⟩

**lemma** *ereal-le-divide-neg*:  
 fixes  $x y z :: \text{ereal}$   
 shows  $x < 0 \implies x \neq -\infty \implies y \leq z / x \longleftrightarrow z \leq x * y$   
 ⟨proof⟩

**lemma** *ereal-divide-le-neg*:  
 fixes  $x y z :: \text{ereal}$   
 shows  $x < 0 \implies x \neq -\infty \implies z / x \leq y \longleftrightarrow x * y \leq z$   
 ⟨proof⟩

**lemma** *ereal-inverse-antimono-strict*:  
 fixes  $x y :: \text{ereal}$   
 shows  $0 \leq x \implies x < y \implies \text{inverse } y < \text{inverse } x$   
 ⟨proof⟩

**lemma** *ereal-inverse-antimono*:  
 fixes  $x y :: \text{ereal}$   
 shows  $0 \leq x \implies x \leq y \implies \text{inverse } y \leq \text{inverse } x$   
 ⟨proof⟩

**lemma** *inverse-inverse-Pinfity-iff[simp]*:

**fixes**  $x :: \text{ereal}$   
**shows**  $\text{inverse } x = \infty \longleftrightarrow x = 0$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-inverse-eq-0*:  
**fixes**  $x :: \text{ereal}$   
**shows**  $\text{inverse } x = 0 \longleftrightarrow x = \infty \vee x = -\infty$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-0-gt-inverse*:  
**fixes**  $x :: \text{ereal}$   
**shows**  $0 < \text{inverse } x \longleftrightarrow x \neq \infty \wedge 0 \leq x$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-inverse-le-0-iff*:  
**fixes**  $x :: \text{ereal}$   
**shows**  $\text{inverse } x \leq 0 \longleftrightarrow x < 0 \vee x = \infty$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-divide-eq-0-iff*:  $x / y = 0 \longleftrightarrow x = 0 \vee |y :: \text{ereal}| = \infty$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-mult-less-right*:  
**fixes**  $a b c :: \text{ereal}$   
**assumes**  $b * a < c * a \ 0 < a \ a < \infty$   
**shows**  $b < c$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-mult-divide*:  
**fixes**  $a b :: \text{ereal}$   
**shows**  $0 < b \implies b < \infty \implies b * (a / b) = a$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-power-divide*:  
**fixes**  $x y :: \text{ereal}$   
**shows**  $y \neq 0 \implies (x / y) ^ n = x ^ n / y ^ n$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-le-mult-one-interval*:  
**fixes**  $x y :: \text{ereal}$   
**assumes**  $y: y \neq -\infty$   
**assumes**  $z: \bigwedge z. 0 < z \implies z < 1 \implies z * x \leq y$   
**shows**  $x \leq y$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-divide-right-mono[simp]*:  
**fixes**  $x y z :: \text{ereal}$   
**assumes**  $x \leq y$   
**and**  $0 < z$

**shows**  $x / z \leq y / z$   
 ⟨proof⟩

**lemma** *ereal-divide-left-mono*[simp]:  
**fixes**  $x y z :: \text{ereal}$   
**assumes**  $y \leq x$   
**and**  $0 < z$   
**and**  $0 < x * y$   
**shows**  $z / x \leq z / y$   
 ⟨proof⟩

**lemma** *ereal-divide-zero-left*[simp]:  
**fixes**  $a :: \text{ereal}$   
**shows**  $0 / a = 0$   
 ⟨proof⟩

**lemma** *ereal-times-divide-eq-left*[simp]:  
**fixes**  $a b c :: \text{ereal}$   
**shows**  $b / c * a = b * a / c$   
 ⟨proof⟩

**lemma** *ereal-times-divide-eq*:  $a * (b / c :: \text{ereal}) = a * b / c$   
 ⟨proof⟩

**lemma** *ereal-inverse-real* [simp]:  $|z| \neq \infty \implies z \neq 0 \implies \text{ereal} (\text{inverse} (\text{real-of-ereal } z)) = \text{inverse } z$   
 ⟨proof⟩

**lemma** *ereal-inverse-mult*:  
 $a \neq 0 \implies b \neq 0 \implies \text{inverse} (a * (b :: \text{ereal})) = \text{inverse } a * \text{inverse } b$   
 ⟨proof⟩

**lemma** *inverse-eq-infinity-iff-eq-zero* [simp]:  
 $1 / (x :: \text{ereal}) = \infty \iff x = 0$   
 ⟨proof⟩

**lemma** *ereal-distrib-left*:  
**fixes**  $a b c :: \text{ereal}$   
**assumes**  $a \neq \infty \vee b \neq -\infty$   
**and**  $a \neq -\infty \vee b \neq \infty$   
**and**  $|c| \neq \infty$   
**shows**  $c * (a + b) = c * a + c * b$   
 ⟨proof⟩

**lemma** *ereal-distrib-minus-left*:  
**fixes**  $a b c :: \text{ereal}$   
**assumes**  $a \neq \infty \vee b \neq \infty$   
**and**  $a \neq -\infty \vee b \neq -\infty$   
**and**  $|c| \neq \infty$

**shows**  $c * (a - b) = c * a - c * b$   
 ⟨*proof*⟩

**lemma** *ereal-distrib-minus-right*:

**fixes**  $a b c :: \text{ereal}$   
**assumes**  $a \neq \infty \vee b \neq \infty$   
**and**  $a \neq -\infty \vee b \neq -\infty$   
**and**  $|c| \neq \infty$   
**shows**  $(a - b) * c = a * c - b * c$   
 ⟨*proof*⟩

## 38.2 Complete lattice

**instantiation** *ereal* :: *lattice*  
**begin**

**definition** [*simp*]:  $\text{sup } x y = (\text{max } x y :: \text{ereal})$   
**definition** [*simp*]:  $\text{inf } x y = (\text{min } x y :: \text{ereal})$   
**instance** ⟨*proof*⟩

**end**

**instantiation** *ereal* :: *complete-lattice*  
**begin**

**definition**  $\text{bot} = (-\infty :: \text{ereal})$   
**definition**  $\text{top} = (\infty :: \text{ereal})$

**definition**  $\text{Sup } S = (\text{SOME } x :: \text{ereal}. (\forall y \in S. y \leq x) \wedge (\forall z. (\forall y \in S. y \leq z) \longrightarrow x \leq z))$

**definition**  $\text{Inf } S = (\text{SOME } x :: \text{ereal}. (\forall y \in S. x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq x))$

**lemma** *ereal-complete-Sup*:

**fixes**  $S :: \text{ereal set}$   
**shows**  $\exists x. (\forall y \in S. y \leq x) \wedge (\forall z. (\forall y \in S. y \leq z) \longrightarrow x \leq z)$   
 ⟨*proof*⟩

**lemma** *ereal-complete-uminus-eq*:

**fixes**  $S :: \text{ereal set}$   
**shows**  $(\forall y \in \text{uminus } S. y \leq x) \wedge (\forall z. (\forall y \in \text{uminus } S. y \leq z) \longrightarrow x \leq z)$   
 $\longleftrightarrow (\forall y \in S. -x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq -x)$   
 ⟨*proof*⟩

**lemma** *ereal-complete-Inf*:

$\exists x. (\forall y \in S :: \text{ereal set}. x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq x)$   
 ⟨*proof*⟩

**instance**



*<proof>*

**end**

**instance** *ereal* :: *complete-linorder* *<proof>*

**instance** *ereal* :: *linear-continuum*  
*<proof>*

**lemma** *min-PIInf* [*simp*]:  $\min (\infty::ereal) x = x$   
*<proof>*

**lemma** *min-PIInf2* [*simp*]:  $\min x (\infty::ereal) = x$   
*<proof>*

**lemma** *max-PIInf* [*simp*]:  $\max (\infty::ereal) x = \infty$   
*<proof>*

**lemma** *max-PIInf2* [*simp*]:  $\max x (\infty::ereal) = \infty$   
*<proof>*

**lemma** *min-MInf* [*simp*]:  $\min (-\infty::ereal) x = -\infty$   
*<proof>*

**lemma** *min-MInf2* [*simp*]:  $\min x (-\infty::ereal) = -\infty$   
*<proof>*

**lemma** *max-MInf* [*simp*]:  $\max (-\infty::ereal) x = x$   
*<proof>*

**lemma** *max-MInf2* [*simp*]:  $\max x (-\infty::ereal) = x$   
*<proof>*

### 38.3 Extended real intervals

**lemma** *real-greaterThanLessThan-infinity-eq*:  
 $real-of-ereal \text{ ‘ } \{N::ereal <..<\infty\} =$   
 $(if N = \infty then \{\} else if N = -\infty then UNIV else \{real-of-ereal N <.. \})$   
*<proof>*

**lemma** *real-greaterThanLessThan-minus-infinity-eq*:  
 $real-of-ereal \text{ ‘ } \{-\infty <..<N::ereal\} =$   
 $(if N = \infty then UNIV else if N = -\infty then \{\} else \{..<real-of-ereal N\})$   
*<proof>*

**lemma** *real-greaterThanLessThan-inter*:  
 $real-of-ereal \text{ ‘ } \{N <..<M::ereal\} = real-of-ereal \text{ ‘ } \{-\infty <..<M\} \cap real-of-ereal \text{ ‘ } \{N <..<\infty\}$   
*<proof>*

**lemma** *real-atLeastGreaterThan-eq*: *real-of-ereal* ‘  $\{N <.. < M :: ereal\}$  =  
 (if  $N = \infty$  then  $\{\}$  else  
 if  $N = -\infty$  then  
 (if  $M = \infty$  then UNIV  
 else if  $M = -\infty$  then  $\{\}$   
 else  $\{.. < \text{real-of-ereal } M\}$ )  
 else if  $M = -\infty$  then  $\{\}$   
 else if  $M = \infty$  then  $\{\text{real-of-ereal } N <..\}$   
 else  $\{\text{real-of-ereal } N <.. < \text{real-of-ereal } M\}$ )  
 ⟨proof⟩

**lemma** *real-image-ereal-ivl*:  
**fixes**  $a b :: ereal$   
**shows**  
*real-of-ereal* ‘  $\{a <.. < b\}$  =  
 (if  $a < b$  then (if  $a = -\infty$  then if  $b = \infty$  then UNIV else  $\{.. < \text{real-of-ereal } b\}$   
 else if  $b = \infty$  then  $\{\text{real-of-ereal } a <..\}$  else  $\{\text{real-of-ereal } a <.. < \text{real-of-ereal } b\}$ )  
 else  $\{\}$ )  
 ⟨proof⟩

**lemma** **fixes**  $a b c :: ereal$   
**shows** *not-inftyI*:  $a < b \implies b < c \implies \text{abs } b \neq \infty$   
 ⟨proof⟩

**context**  
**fixes**  $r s t :: real$   
**begin**

**lemma** *interval-Ioo-neq-Ioi*:  $\{r <.. < s\} \neq \{t <..\}$   
 ⟨proof⟩

**lemma** *interval-Ioo-neq-Iio*:  $\{r <.. < s\} \neq \{.. < t\}$   
 ⟨proof⟩

**lemma** *interval-neq-ooo-UNIV*:  $\{r <.. < s\} \neq \text{UNIV}$   
**and** *interval-Ioi-neq-UNIV*:  $\{r <..\} \neq \text{UNIV}$   
**and** *interval-Iio-neq-UNIV*:  $\{.. < r\} \neq \text{UNIV}$   
 ⟨proof⟩

**lemma** *interval-Ioi-neq-Iio*:  $\{r <..\} \neq \{.. < s\}$   
 ⟨proof⟩

**lemma** *interval-empty-neq-Ioi*:  $\{\} \neq \{r <..\}$   
**and** *interval-empty-neq-Iio*:  $\{\} \neq \{.. < r\}$   
 ⟨proof⟩

**end**

**lemmas** *interval-neqs* = *interval-Ioo-neq-Ioi interval-Ioo-neq-Iio*  
*interval-neq-foo-UNIV interval-Ioi-neq-Iio*  
*interval-Ioi-neq-UNIV interval-Iio-neq-UNIV*  
*interval-empty-neq-Ioi interval-empty-neq-Iio*

**lemma** *greaterThanLessThan-eq-iff*:

**fixes**  $r\ s\ t\ u::\text{real}$

**shows**  $(\{r <..< s\} = \{t <..< u\}) = (r \geq s \wedge u \leq t \vee r = t \wedge s = u)$

*<proof>*

**lemma** *real-of-ereal-image-greaterThanLessThan-iff*:

*real-of-ereal* ‘  $\{a <..< b\} = \text{real-of-ereal}$  ‘  $\{c <..< d\} \iff (a \geq b \wedge c \geq d \vee a = c \wedge b = d)$

*<proof>*

**lemma** *uminus-image-real-of-ereal-image-greaterThanLessThan*:

*uminus* ‘ *real-of-ereal* ‘  $\{l <..< u\} = \text{real-of-ereal}$  ‘  $\{-u <..< -l\}$

*<proof>*

**lemma** *add-image-real-of-ereal-image-greaterThanLessThan*:

$(+)$  *c* ‘ *real-of-ereal* ‘  $\{l <..< u\} = \text{real-of-ereal}$  ‘  $\{c + l <..< c + u\}$

*<proof>*

**lemma** *add2-image-real-of-ereal-image-greaterThanLessThan*:

$(\lambda x. x + c)$  ‘ *real-of-ereal* ‘  $\{l <..< u\} = \text{real-of-ereal}$  ‘  $\{l + c <..< u + c\}$

*<proof>*

**lemma** *minus-image-real-of-ereal-image-greaterThanLessThan*:

$(-)$  *c* ‘ *real-of-ereal* ‘  $\{l <..< u\} = \text{real-of-ereal}$  ‘  $\{c - u <..< c - l\}$

**(is ?l = ?r)**

*<proof>*

**lemma** *real-ereal-bound-lemma-up*:

**assumes**  $s \in \text{real-of-ereal}$  ‘  $\{a <..< b\}$

**assumes**  $t \notin \text{real-of-ereal}$  ‘  $\{a <..< b\}$

**assumes**  $s \leq t$

**shows**  $b \neq \infty$

*<proof>*

**lemma** *real-ereal-bound-lemma-down*:

**assumes**  $s: s \in \text{real-of-ereal}$  ‘  $\{a <..< b\}$

**and**  $t: t \notin \text{real-of-ereal}$  ‘  $\{a <..< b\}$

**and**  $t \leq s$

**shows**  $a \neq -\infty$

*<proof>*

## 38.4 Topological space

**instantiation** *ereal* :: *linear-continuum-topology*

**begin**

**definition** *open-ereal* :: *ereal set*  $\Rightarrow$  *bool* **where**

*open-ereal-generated*: *open-ereal* = *generate-topology* (*range lessThan*  $\cup$  *range greaterThan*)

**instance**

*<proof>*

**end**

**lemma** *continuous-on-ereal*[*continuous-intros*]:

**assumes** *f*: *continuous-on s f* **shows** *continuous-on s* ( $\lambda x. \text{ereal } (f x)$ )

*<proof>*

**lemma** *tendsto-ereal*[*tendsto-intros, simp, intro*]: ( $f \longrightarrow x$ ) *F*  $\Longrightarrow$  ( $(\lambda x. \text{ereal } (f x)) \longrightarrow \text{ereal } x$ ) *F*

*<proof>*

**lemma** *tendsto-uminus-ereal*[*tendsto-intros, simp, intro*]:

**assumes** ( $f \longrightarrow x$ ) *F*

**shows** ( $(\lambda x. - f x :: \text{ereal}) \longrightarrow - x$ ) *F*

*<proof>*

**lemma** *at-infty-ereal-eq-at-top*:  $\text{at } \infty = \text{filtermap } \text{ereal } \text{at-top}$

*<proof>*

**lemma** *ereal-Lim-uminus*: ( $f \longrightarrow f0$ ) *net*  $\longleftrightarrow$  ( $(\lambda x. - f x :: \text{ereal}) \longrightarrow - f0$ ) *net*

*<proof>*

**lemma** *ereal-divide-less-iff*:  $0 < (c :: \text{ereal}) \Longrightarrow c < \infty \Longrightarrow a / c < b \longleftrightarrow a < b * c$

*<proof>*

**lemma** *ereal-less-divide-iff*:  $0 < (c :: \text{ereal}) \Longrightarrow c < \infty \Longrightarrow a < b / c \longleftrightarrow a * c < b$

*<proof>*

**lemma** *tendsto-cmult-ereal*[*tendsto-intros, simp, intro*]:

**assumes** *c*:  $|c| \neq \infty$  **and** *f*: ( $f \longrightarrow x$ ) *F*

**shows** ( $(\lambda x. c * f x :: \text{ereal}) \longrightarrow c * x$ ) *F*

*<proof>*

**lemma** *tendsto-cmult-ereal-not-0*[*tendsto-intros, simp, intro*]:

**assumes** *x*  $\neq 0$  **and** *f*: ( $f \longrightarrow x$ ) *F*

**shows** ( $(\lambda x. c * f x :: \text{ereal}) \longrightarrow c * x$ ) *F*

*<proof>*

**lemma** *tendsto-cadd-ereal*[*tendsto-intros, simp, intro*]:  
**assumes**  $c: y \neq -\infty \text{ and } x \neq -\infty$  **and**  $f: (f \longrightarrow x) F$   
**shows**  $((\lambda x. f x + y)::ereal) \longrightarrow x + y) F$   
*<proof>*

**lemma** *tendsto-add-left-ereal*[*tendsto-intros, simp, intro*]:  
**assumes**  $c: |y| \neq \infty$  **and**  $f: (f \longrightarrow x) F$   
**shows**  $((\lambda x. f x + y)::ereal) \longrightarrow x + y) F$   
*<proof>*

**lemma** *continuous-at-ereal*[*continuous-intros*]: *continuous*  $F f \implies \text{continuous } F$   
 $(\lambda x. ereal (f x))$   
*<proof>*

**lemma** *ereal-Sup*:  
**assumes**  $*$ :  $|SUP a \in A. ereal a| \neq \infty$   
**shows**  $ereal (Sup A) = (SUP a \in A. ereal a)$   
*<proof>*

**lemma** *ereal-SUP*:  $|SUP a \in A. ereal (f a)| \neq \infty \implies ereal (SUP a \in A. f a) = (SUP a \in A. ereal (f a))$   
*<proof>*

**lemma** *ereal-Inf*:  
**assumes**  $*$ :  $|INF a \in A. ereal a| \neq \infty$   
**shows**  $ereal (Inf A) = (INF a \in A. ereal a)$   
*<proof>*

**lemma** *ereal-Inf'*:  
**assumes**  $*$ : *bdd-below*  $A A \neq \{\}$   
**shows**  $ereal (Inf A) = (INF a \in A. ereal a)$   
*<proof>*

**lemma** *ereal-INF*:  $|INF a \in A. ereal (f a)| \neq \infty \implies ereal (INF a \in A. f a) = (INF a \in A. ereal (f a))$   
*<proof>*

**lemma** *ereal-Sup-uminus-image-eq*:  $Sup (uminus ' S::ereal set) = - Inf S$   
*<proof>*

**lemma** *ereal-SUP-uminus-eq*:  
**fixes**  $f :: 'a \Rightarrow ereal$   
**shows**  $(SUP x \in S. uminus (f x)) = - (INF x \in S. f x)$   
*<proof>*

**lemma** *ereal-inj-on-uminus*[*intro, simp*]: *inj-on* *uminus*  $(A :: ereal set)$   
*<proof>*

**lemma** *ereal-Inf-uminus-image-eq*:  $\text{Inf} (\text{uminus } ' S :: \text{ereal set}) = - \text{Sup } S$   
 ⟨proof⟩

**lemma** *ereal-INF-uminus-eq*:  
**fixes**  $f :: 'a \Rightarrow \text{ereal}$   
**shows**  $(\text{INF } x \in S. - f x) = - (\text{SUP } x \in S. f x)$   
 ⟨proof⟩

**lemma** *ereal-SUP-not-infty*:  
**fixes**  $f :: - \Rightarrow \text{ereal}$   
**shows**  $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall a \in A. l \leq f a \wedge f a \leq u \Longrightarrow |\text{Sup} (f ' A)| \neq \infty$   
 ⟨proof⟩

**lemma** *ereal-INF-not-infty*:  
**fixes**  $f :: - \Rightarrow \text{ereal}$   
**shows**  $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall a \in A. l \leq f a \wedge f a \leq u \Longrightarrow |\text{Inf} (f ' A)| \neq \infty$   
 ⟨proof⟩

**lemma** *ereal-image-uminus-shift*:  
**fixes**  $X Y :: \text{ereal set}$   
**shows**  $\text{uminus } ' X = Y \longleftrightarrow X = \text{uminus } ' Y$   
 ⟨proof⟩

**lemma** *Sup-eq-MInfty*:  
**fixes**  $S :: \text{ereal set}$   
**shows**  $\text{Sup } S = -\infty \longleftrightarrow S = \{\} \vee S = \{-\infty\}$   
 ⟨proof⟩

**lemma** *Inf-eq-PInfty*:  
**fixes**  $S :: \text{ereal set}$   
**shows**  $\text{Inf } S = \infty \longleftrightarrow S = \{\} \vee S = \{\infty\}$   
 ⟨proof⟩

**lemma** *Inf-eq-MInfty*:  
**fixes**  $S :: \text{ereal set}$   
**shows**  $-\infty \in S \Longrightarrow \text{Inf } S = -\infty$   
 ⟨proof⟩

**lemma** *Sup-eq-PInfty*:  
**fixes**  $S :: \text{ereal set}$   
**shows**  $\infty \in S \Longrightarrow \text{Sup } S = \infty$   
 ⟨proof⟩

**lemma** *not-MInfty-nonneg[simp]*:  $0 \leq (x :: \text{ereal}) \Longrightarrow x \neq -\infty$   
 ⟨proof⟩

**lemma** *Sup-ereal-close*:

**fixes**  $e :: \text{ereal}$   
**assumes**  $0 < e$   
**and**  $S: |Sup S| \neq \infty \ S \neq \{\}$   
**shows**  $\exists x \in S. Sup S - e < x$   
 $\langle \text{proof} \rangle$

**lemma** *Inf-ereal-close*:

**fixes**  $e :: \text{ereal}$   
**assumes**  $|Inf X| \neq \infty$   
**and**  $0 < e$   
**shows**  $\exists x \in X. x < Inf X + e$   
 $\langle \text{proof} \rangle$

**lemma** *SUP-PInfy*:

$(\bigwedge n :: \text{nat}. \exists i \in A. \text{ereal}(\text{real } n) \leq f i) \implies (SUP i \in A. f i :: \text{ereal}) = \infty$   
 $\langle \text{proof} \rangle$

**lemma** *SUP-nat-Infy*:  $(SUP i. \text{ereal}(\text{real } i)) = \infty$

$\langle \text{proof} \rangle$

**lemma** *SUP-ereal-add-left*:

**assumes**  $I \neq \{\}$   $c \neq -\infty$   
**shows**  $(SUP i \in I. f i + c :: \text{ereal}) = (SUP i \in I. f i) + c$   
 $\langle \text{proof} \rangle$

**lemma** *SUP-ereal-add-right*:

**fixes**  $c :: \text{ereal}$   
**shows**  $I \neq \{\} \implies c \neq -\infty \implies (SUP i \in I. c + f i) = c + (SUP i \in I. f i)$   
 $\langle \text{proof} \rangle$

**lemma** *SUP-ereal-minus-right*:

**assumes**  $I \neq \{\}$   $c \neq -\infty$   
**shows**  $(SUP i \in I. c - f i :: \text{ereal}) = c - (INF i \in I. f i)$   
 $\langle \text{proof} \rangle$

**lemma** *SUP-ereal-minus-left*:

**assumes**  $I \neq \{\}$   $c \neq \infty$   
**shows**  $(SUP i \in I. f i - c :: \text{ereal}) = (SUP i \in I. f i) - c$   
 $\langle \text{proof} \rangle$

**lemma** *INF-ereal-minus-right*:

**assumes**  $I \neq \{\}$  **and**  $|c| \neq \infty$   
**shows**  $(INF i \in I. c - f i) = c - (SUP i \in I. f i :: \text{ereal})$   
 $\langle \text{proof} \rangle$

**lemma** *SUP-ereal-le-addI*:

**fixes**  $f :: 'i \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge i. f i + y \leq z$  **and**  $y \neq -\infty$

**shows**  $Sup (f \text{ ‘ } UNIV) + y \leq z$   
 ⟨proof⟩

**lemma** *SUP-combine*:

**fixes**  $f :: 'a::semilattice-sup \Rightarrow 'a::semilattice-sup \Rightarrow 'b::complete-lattice$   
**assumes**  $mono: \bigwedge a b c d. a \leq b \Longrightarrow c \leq d \Longrightarrow f a c \leq f b d$   
**shows**  $(SUP i \in UNIV. SUP j \in UNIV. f i j) = (SUP i. f i i)$   
 ⟨proof⟩

**lemma** *SUP-ereal-add*:

**fixes**  $f g :: nat \Rightarrow ereal$   
**assumes**  $inc: incseq f incseq g$   
**and**  $pos: \bigwedge i. f i \neq -\infty \bigwedge i. g i \neq -\infty$   
**shows**  $(SUP i. f i + g i) = Sup (f \text{ ‘ } UNIV) + Sup (g \text{ ‘ } UNIV)$   
 ⟨proof⟩

**lemma** *INF-eq-minf*:  $(INF i \in I. f i :: ereal) \neq -\infty \longleftrightarrow (\exists b > -\infty. \forall i \in I. b \leq f i)$   
 ⟨proof⟩

**lemma** *INF-ereal-add-left*:

**assumes**  $I \neq \{\}$   $c \neq -\infty \bigwedge x. x \in I \Longrightarrow 0 \leq f x$   
**shows**  $(INF i \in I. f i + c :: ereal) = (INF i \in I. f i) + c$   
 ⟨proof⟩

**lemma** *INF-ereal-add-right*:

**assumes**  $I \neq \{\}$   $c \neq -\infty \bigwedge x. x \in I \Longrightarrow 0 \leq f x$   
**shows**  $(INF i \in I. c + f i :: ereal) = c + (INF i \in I. f i)$   
 ⟨proof⟩

**lemma** *INF-ereal-add-directed*:

**fixes**  $f g :: 'a \Rightarrow ereal$   
**assumes**  $nonneg: \bigwedge i. i \in I \Longrightarrow 0 \leq f i \bigwedge i. i \in I \Longrightarrow 0 \leq g i$   
**assumes**  $directed: \bigwedge i j. i \in I \Longrightarrow j \in I \Longrightarrow \exists k \in I. f i + g j \geq f k + g k$   
**shows**  $(INF i \in I. f i + g i) = (INF i \in I. f i) + (INF i \in I. g i)$   
 ⟨proof⟩

**lemma** *INF-ereal-add*:

**fixes**  $f :: nat \Rightarrow ereal$   
**assumes**  $decseq f decseq g$   
**and**  $fin: \bigwedge i. f i \neq \infty \bigwedge i. g i \neq \infty$   
**shows**  $(INF i. f i + g i) = Inf (f \text{ ‘ } UNIV) + Inf (g \text{ ‘ } UNIV)$   
 ⟨proof⟩

**lemma** *SUP-ereal-add-pos*:

**fixes**  $f g :: nat \Rightarrow ereal$   
**assumes**  $incseq f incseq g$   
**and**  $\bigwedge i. 0 \leq f i \bigwedge i. 0 \leq g i$   
**shows**  $(SUP i. f i + g i) = Sup (f \text{ ‘ } UNIV) + Sup (g \text{ ‘ } UNIV)$   
 ⟨proof⟩



**lemma** *SUP-ereal-sum*:

**fixes**  $f\ g :: 'a \Rightarrow \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge n. n \in A \Longrightarrow \text{incseq } (f\ n)$   
**and**  $\text{pos}: \bigwedge n\ i. n \in A \Longrightarrow 0 \leq f\ n\ i$   
**shows**  $(\text{SUP } i. \sum_{n \in A}. f\ n\ i) = (\sum_{n \in A}. \text{Sup } ((f\ n) \text{ ' } \text{UNIV}))$   
 $\langle \text{proof} \rangle$

**lemma** *SUP-ereal-mult-left*:

**fixes**  $f :: 'a \Rightarrow \text{ereal}$   
**assumes**  $I \neq \{\}$   
**assumes**  $f: \bigwedge i. i \in I \Longrightarrow 0 \leq f\ i$  **and**  $c: 0 \leq c$   
**shows**  $(\text{SUP } i \in I. c * f\ i) = c * (\text{SUP } i \in I. f\ i)$   
 $\langle \text{proof} \rangle$

**lemma** *countable-approach*:

**fixes**  $x :: \text{ereal}$   
**assumes**  $x \neq -\infty$   
**shows**  $\exists f. \text{incseq } f \wedge (\forall i :: \text{nat}. f\ i < x) \wedge (f \longrightarrow x)$   
 $\langle \text{proof} \rangle$

**lemma** *Sup-countable-SUP*:

**assumes**  $A \neq \{\}$   
**shows**  $\exists f :: \text{nat} \Rightarrow \text{ereal}. \text{incseq } f \wedge \text{range } f \subseteq A \wedge \text{Sup } A = (\text{SUP } i. f\ i)$   
 $\langle \text{proof} \rangle$

**lemma** *Inf-countable-INF*:

**assumes**  $A \neq \{\}$  **shows**  $\exists f :: \text{nat} \Rightarrow \text{ereal}. \text{decseq } f \wedge \text{range } f \subseteq A \wedge \text{Inf } A =$   
 $(\text{INF } i. f\ i)$   
 $\langle \text{proof} \rangle$

**lemma** *SUP-countable-SUP*:

$A \neq \{\} \Longrightarrow \exists f :: \text{nat} \Rightarrow \text{ereal}. \text{range } f \subseteq g\ A \wedge \text{Sup } (g \text{ ' } A) = \text{Sup } (f \text{ ' } \text{UNIV})$   
 $\langle \text{proof} \rangle$

### 38.5 Relation to *enat*

**definition** *ereal-of-enat*  $n = (\text{case } n \text{ of } \text{enat } n \Rightarrow \text{ereal } (\text{real } n) \mid \infty \Rightarrow \infty)$

**declare**  $[[\text{coercion } \text{ereal-of-enat} :: \text{enat} \Rightarrow \text{ereal}]]$

**declare**  $[[\text{coercion } (\lambda n. \text{ereal } (\text{real } n)) :: \text{nat} \Rightarrow \text{ereal}]]$

**lemma** *ereal-of-enat-simps* $[\text{simp}]$ :

$\text{ereal-of-enat } (\text{enat } n) = \text{ereal } n$   
 $\text{ereal-of-enat } \infty = \infty$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-of-enat-le-iff* $[\text{simp}]$ :  $\text{ereal-of-enat } m \leq \text{ereal-of-enat } n \longleftrightarrow m \leq n$

$\langle \text{proof} \rangle$

**lemma** *ereal-of-enat-less-iff[simp]*:  $\text{ereal-of-enat } m < \text{ereal-of-enat } n \longleftrightarrow m < n$   
 ⟨proof⟩

**lemma** *numeral-le-ereal-of-enat-iff[simp]*:  $\text{numeral } m \leq \text{ereal-of-enat } n \longleftrightarrow \text{numeral } m \leq n$   
 ⟨proof⟩

**lemma** *numeral-less-ereal-of-enat-iff[simp]*:  $\text{numeral } m < \text{ereal-of-enat } n \longleftrightarrow \text{numeral } m < n$   
 ⟨proof⟩

**lemma** *ereal-of-enat-ge-zero-cancel-iff[simp]*:  $0 \leq \text{ereal-of-enat } n \longleftrightarrow 0 \leq n$   
 ⟨proof⟩

**lemma** *ereal-of-enat-gt-zero-cancel-iff[simp]*:  $0 < \text{ereal-of-enat } n \longleftrightarrow 0 < n$   
 ⟨proof⟩

**lemma** *ereal-of-enat-zero[simp]*:  $\text{ereal-of-enat } 0 = 0$   
 ⟨proof⟩

**lemma** *ereal-of-enat-inf[simp]*:  $\text{ereal-of-enat } n = \infty \longleftrightarrow n = \infty$   
 ⟨proof⟩

**lemma** *ereal-of-enat-add*:  $\text{ereal-of-enat } (m + n) = \text{ereal-of-enat } m + \text{ereal-of-enat } n$   
 ⟨proof⟩

**lemma** *ereal-of-enat-sub*:  
**assumes**  $n \leq m$   
**shows**  $\text{ereal-of-enat } (m - n) = \text{ereal-of-enat } m - \text{ereal-of-enat } n$   
 ⟨proof⟩

**lemma** *ereal-of-enat-mult*:  
 $\text{ereal-of-enat } (m * n) = \text{ereal-of-enat } m * \text{ereal-of-enat } n$   
 ⟨proof⟩

**lemmas** *ereal-of-enat-pushin* = *ereal-of-enat-add* *ereal-of-enat-sub* *ereal-of-enat-mult*  
**lemmas** *ereal-of-enat-pushout* = *ereal-of-enat-pushin[symmetric]*

**lemma** *ereal-of-enat-nonneg*:  $\text{ereal-of-enat } n \geq 0$   
 ⟨proof⟩

**lemma** *ereal-of-enat-Sup*:  
**assumes**  $A \neq \{\}$  **shows**  $\text{ereal-of-enat } (\text{Sup } A) = (\text{SUP } a \in A. \text{ereal-of-enat } a)$   
 ⟨proof⟩

**lemma** *ereal-of-enat-SUP*:  
 $A \neq \{\} \implies \text{ereal-of-enat } (\text{SUP } a \in A. f a) = (\text{SUP } a \in A. \text{ereal-of-enat } (f a))$

*<proof>*

### 38.6 Limits on *ereal*

**lemma** *open-PInfty*:  $open\ A \implies \infty \in A \implies (\exists x. \{ereal\ x < ..\} \subseteq A)$   
*<proof>*

**lemma** *open-MInfty*:  $open\ A \implies -\infty \in A \implies (\exists x. \{.. <ereal\ x\} \subseteq A)$   
*<proof>*

**lemma** *open-ereal-vimage*:  $open\ S \implies open\ (ereal - ' S)$   
*<proof>*

**lemma** *open-ereal*:  $open\ S \implies open\ (ereal ' S)$   
*<proof>*

**lemma** *open-image-real-of-ereal*:  
**fixes**  $X :: ereal\ set$   
**assumes**  $open\ X$   
**assumes** *infty*:  $\infty \notin X \quad -\infty \notin X$   
**shows**  $open\ (real-of-ereal ' X)$   
*<proof>*

**lemma** *eventually-finite*:  
**fixes**  $x :: ereal$   
**assumes**  $|x| \neq \infty \quad (f \longrightarrow x)\ F$   
**shows** *eventually*  $(\lambda x. |f\ x| \neq \infty)\ F$   
*<proof>*

**lemma** *open-ereal-def*:  
 $open\ A \iff open\ (ereal - ' A) \wedge (\infty \in A \implies (\exists x. \{ereal\ x < ..\} \subseteq A)) \wedge (-\infty \in A \implies (\exists x. \{.. <ereal\ x\} \subseteq A))$   
**(is**  $open\ A \iff ?rhs$   
*<proof>*

**lemma** *open-PInfty2*:  
**assumes**  $open\ A$  **and**  $\infty \in A$   
**obtains**  $x$  **where**  $\{ereal\ x < ..\} \subseteq A$   
*<proof>*

**lemma** *open-MInfty2*:  
**assumes**  $open\ A$  **and**  $-\infty \in A$   
**obtains**  $x$  **where**  $\{.. <ereal\ x\} \subseteq A$   
*<proof>*

**lemma** *ereal-openE*:  
**assumes**  $open\ A$   
**obtains**  $x\ y$  **where**  $open\ (ereal - ' A)$

**and**  $\infty \in A \implies \{ereal\ x < ..\} \subseteq A$   
**and**  $-\infty \in A \implies \{.. < ereal\ y\} \subseteq A$   
 ⟨proof⟩

**lemmas**  $open\text{-}ereal\text{-}lessThan = open\text{-}lessThan[\mathbf{where}\ 'a=ereal]$   
**lemmas**  $open\text{-}ereal\text{-}greaterThan = open\text{-}greaterThan[\mathbf{where}\ 'a=ereal]$   
**lemmas**  $ereal\text{-}open\text{-}greaterThanLessThan = open\text{-}greaterThanLessThan[\mathbf{where}\ 'a=ereal]$   
**lemmas**  $closed\text{-}ereal\text{-}atLeast = closed\text{-}atLeast[\mathbf{where}\ 'a=ereal]$   
**lemmas**  $closed\text{-}ereal\text{-}atMost = closed\text{-}atMost[\mathbf{where}\ 'a=ereal]$   
**lemmas**  $closed\text{-}ereal\text{-}atLeastAtMost = closed\text{-}atLeastAtMost[\mathbf{where}\ 'a=ereal]$   
**lemmas**  $closed\text{-}ereal\text{-}singleton = closed\text{-}singleton[\mathbf{where}\ 'a=ereal]$

**lemma**  $ereal\text{-}open\text{-}cont\text{-}interval$ :  
**fixes**  $S :: ereal\ set$   
**assumes**  $open\ S$   
**and**  $x \in S$   
**and**  $|x| \neq \infty$   
**obtains**  $e$  **where**  $e > 0$  **and**  $\{x-e < .. < x+e\} \subseteq S$   
 ⟨proof⟩

**lemma**  $ereal\text{-}open\text{-}cont\text{-}interval2$ :  
**fixes**  $S :: ereal\ set$   
**assumes**  $open\ S$  **and**  $x \in S$  **and**  $|x| \neq \infty$   
**obtains**  $a\ b$  **where**  $a < x$  **and**  $x < b$  **and**  $\{a < .. < b\} \subseteq S$   
 ⟨proof⟩

### 38.6.1 Convergent sequences

**lemma**  $lim\text{-}real\text{-}of\text{-}ereal[simp]$ :  
**assumes**  $lim: (f \longrightarrow ereal\ x)$  *net*  
**shows**  $((\lambda x. real\text{-}of\text{-}ereal\ (f\ x)) \longrightarrow x)$  *net*  
 ⟨proof⟩

**lemma**  $lim\text{-}ereal[simp]$ :  $((\lambda n. ereal\ (f\ n)) \longrightarrow ereal\ x)$  *net*  $\longleftrightarrow (f \longrightarrow x)$  *net*  
 ⟨proof⟩

**lemma**  $convergent\text{-}real\text{-}imp\text{-}convergent\text{-}ereal$ :  
**assumes**  $convergent\ a$   
**shows**  $convergent\ (\lambda n. ereal\ (a\ n))$  **and**  $lim\ (\lambda n. ereal\ (a\ n)) = ereal\ (lim\ a)$   
 ⟨proof⟩

**lemma**  $tendsto\text{-}PInfty$ :  $(f \longrightarrow \infty)$   $F \longleftrightarrow (\forall r. eventually\ (\lambda x. ereal\ r < f\ x)\ F)$   
 ⟨proof⟩

**lemma**  $tendsto\text{-}PInfty'$ :  $(f \longrightarrow \infty)$   $F = (\forall r > c. eventually\ (\lambda x. ereal\ r < f\ x)\ F)$   
 ⟨proof⟩

**lemma**  $tendsto\text{-}PInfty\text{-}eq\text{-}at\text{-}top$ :

$((\lambda z. \text{ereal } (f z)) \longrightarrow \infty) F \longleftrightarrow (\text{LIM } z F. f z \text{ :> at-top})$   
 ⟨proof⟩

**lemma** *tendsto-MInfty*:  $(f \longrightarrow -\infty) F \longleftrightarrow (\forall r. \text{eventually } (\lambda x. f x < \text{ereal } r) F)$   
 ⟨proof⟩

**lemma** *tendsto-MInfty'*:  $(f \longrightarrow -\infty) F = (\forall r < c. \text{eventually } (\lambda x. \text{ereal } r > f x) F)$   
 ⟨proof⟩

**lemma** *Lim-PInfty*:  $f \longrightarrow \infty \longleftrightarrow (\forall B. \exists N. \forall n \geq N. f n \geq \text{ereal } B)$   
 ⟨proof⟩

**lemma** *Lim-MInfty*:  $f \longrightarrow -\infty \longleftrightarrow (\forall B. \exists N. \forall n \geq N. \text{ereal } B \geq f n)$   
 ⟨proof⟩

**lemma** *Lim-bounded-PInfty*:  $f \longrightarrow l \implies (\bigwedge n. f n \leq \text{ereal } B) \implies l \neq \infty$   
 ⟨proof⟩

**lemma** *Lim-bounded-MInfty*:  $f \longrightarrow l \implies (\bigwedge n. \text{ereal } B \leq f n) \implies l \neq -\infty$   
 ⟨proof⟩

**lemma** *tendsto-zero-erealI*:  
**assumes**  $\bigwedge e. e > 0 \implies \text{eventually } (\lambda x. |f x| < \text{ereal } e) F$   
**shows**  $(f \longrightarrow 0) F$   
 ⟨proof⟩

**lemma** *Lim-bounded-PInfty2*:  $f \longrightarrow l \implies \forall n \geq N. f n \leq \text{ereal } B \implies l \neq \infty$   
 ⟨proof⟩

**lemma** *real-of-ereal-mult[simp]*:  
**fixes**  $a b :: \text{ereal}$   
**shows**  $\text{real-of-ereal } (a * b) = \text{real-of-ereal } a * \text{real-of-ereal } b$   
 ⟨proof⟩

**lemma** *real-of-ereal-eq-0*:  
**fixes**  $x :: \text{ereal}$   
**shows**  $\text{real-of-ereal } x = 0 \longleftrightarrow x = \infty \vee x = -\infty \vee x = 0$   
 ⟨proof⟩

**lemma** *tendsto-ereal-realD*:  
**fixes**  $f :: 'a \Rightarrow \text{ereal}$   
**assumes**  $x \neq 0$   
**and** *tendsto*:  $(\lambda x. \text{ereal } (\text{real-of-ereal } (f x))) \longrightarrow x$  *net*  
**shows**  $(f \longrightarrow x)$  *net*  
 ⟨proof⟩

**lemma** *tendsto-ereal-realI*:

**fixes**  $f :: 'a \Rightarrow ereal$   
**assumes**  $x: |x| \neq \infty$  **and**  $tendsto: (f \longrightarrow x)$  *net*  
**shows**  $((\lambda x. ereal (real-of-ereal (f x))) \longrightarrow x)$  *net*  
 $\langle proof \rangle$

**lemma** *ereal-mult-cancel-left*:

**fixes**  $a b c :: ereal$   
**shows**  $a * b = a * c \longleftrightarrow (|a| = \infty \wedge 0 < b * c) \vee a = 0 \vee b = c$   
 $\langle proof \rangle$

**lemma** *tendsto-add-ereal*:

**fixes**  $x y :: ereal$   
**assumes**  $x: |x| \neq \infty$  **and**  $y: |y| \neq \infty$   
**assumes**  $f: (f \longrightarrow x)$   $F$  **and**  $g: (g \longrightarrow y)$   $F$   
**shows**  $((\lambda x. f x + g x) \longrightarrow x + y)$   $F$   
 $\langle proof \rangle$

**lemma** *tendsto-add-ereal-nonneg*:

**fixes**  $x y :: ereal$   
**assumes**  $x \neq -\infty$   $y \neq -\infty$   $(f \longrightarrow x)$   $F$   $(g \longrightarrow y)$   $F$   
**shows**  $((\lambda x. f x + g x) \longrightarrow x + y)$   $F$   
 $\langle proof \rangle$

**lemma** *ereal-inj-affinity*:

**fixes**  $m t :: ereal$   
**assumes**  $|m| \neq \infty$   
**and**  $m \neq 0$   
**and**  $|t| \neq \infty$   
**shows** *inj-on*  $(\lambda x. m * x + t)$   $A$   
 $\langle proof \rangle$

**lemma** *ereal-PInfty-eq-plus[simp]*:

**fixes**  $a b :: ereal$   
**shows**  $\infty = a + b \longleftrightarrow a = \infty \vee b = \infty$   
 $\langle proof \rangle$

**lemma** *ereal-MInfty-eq-plus[simp]*:

**fixes**  $a b :: ereal$   
**shows**  $-\infty = a + b \longleftrightarrow (a = -\infty \wedge b \neq \infty) \vee (b = -\infty \wedge a \neq \infty)$   
 $\langle proof \rangle$

**lemma** *ereal-less-divide-pos*:

**fixes**  $x y :: ereal$   
**shows**  $x > 0 \implies x \neq \infty \implies y < z / x \longleftrightarrow x * y < z$   
 $\langle proof \rangle$

**lemma** *ereal-divide-less-pos*:

**fixes**  $x y z :: ereal$   
**shows**  $x > 0 \implies x \neq \infty \implies y / x < z \longleftrightarrow y < x * z$

*<proof>*

**lemma** *ereal-divide-eq*:

**fixes**  $a\ b\ c :: \text{ereal}$

**shows**  $b \neq 0 \implies |b| \neq \infty \implies a / b = c \longleftrightarrow a = b * c$

*<proof>*

**lemma** *ereal-inverse-not-MInfty[simp]*:  $\text{inverse } (a :: \text{ereal}) \neq -\infty$

*<proof>*

**lemma** *ereal-mult-m1[simp]*:  $x * \text{ereal } (-1) = -x$

*<proof>*

**lemma** *ereal-real'*:

**assumes**  $|x| \neq \infty$

**shows**  $\text{ereal } (\text{real-of-ereal } x) = x$

*<proof>*

**lemma** *real-ereal-id*:  $\text{real-of-ereal} \circ \text{ereal} = \text{id}$

*<proof>*

**lemma** *open-image-ereal*:  $\text{open}(UNIV - \{\infty, (-\infty :: \text{ereal})\})$

*<proof>*

**lemma** *ereal-le-distrib*:

**fixes**  $a\ b\ c :: \text{ereal}$

**shows**  $c * (a + b) \leq c * a + c * b$

*<proof>*

**lemma** *ereal-pos-distrib*:

**fixes**  $a\ b\ c :: \text{ereal}$

**assumes**  $0 \leq c$

**and**  $c \neq \infty$

**shows**  $c * (a + b) = c * a + c * b$

*<proof>*

**lemma** *ereal-LimI-finite*:

**fixes**  $x :: \text{ereal}$

**assumes**  $|x| \neq \infty$

**and**  $\bigwedge r. 0 < r \implies \exists N. \forall n \geq N. u\ n < x + r \wedge x < u\ n + r$

**shows**  $u \longrightarrow x$

*<proof>*

**lemma** *tendsto-obtains-N*:

**assumes**  $f \longrightarrow f0$  *open*  $S$   $f0 \in S$

**obtains**  $N$  **where**  $\forall n \geq N. f\ n \in S$

*<proof>*

**lemma** *ereal-LimI-finite-iff*:

**fixes**  $x :: \text{ereal}$   
**assumes**  $|x| \neq \infty$   
**shows**  $u \longrightarrow x \longleftrightarrow (\forall r. 0 < r \longrightarrow (\exists N. \forall n \geq N. u\ n < x + r \wedge x < u\ n + r))$   
**(is ?lhs  $\longleftrightarrow$  ?rhs)**  
 <proof>

**lemma** *ereal-Limsup-uminus*:  
**fixes**  $f :: 'a \Rightarrow \text{ereal}$   
**shows**  $\text{Limsup net } (\lambda x. - (f\ x)) = - \text{Liminf net } f$   
 <proof>

**lemma** *liminf-bounded-iff*:  
**fixes**  $x :: \text{nat} \Rightarrow \text{ereal}$   
**shows**  $C \leq \text{liminf } x \longleftrightarrow (\forall B < C. \exists N. \forall n \geq N. B < x\ n)$   
**(is ?lhs  $\longleftrightarrow$  ?rhs)**  
 <proof>

**lemma** *Liminf-add-le*:  
**fixes**  $f\ g :: - \Rightarrow \text{ereal}$   
**assumes**  $F: F \neq \text{bot}$   
**assumes**  $ev: \text{eventually } (\lambda x. 0 \leq f\ x)\ F \text{ eventually } (\lambda x. 0 \leq g\ x)\ F$   
**shows**  $\text{Liminf } F\ f + \text{Liminf } F\ g \leq \text{Liminf } F\ (\lambda x. f\ x + g\ x)$   
 <proof>

**lemma** *Sup-ereal-mult-right'*:  
**assumes**  $\text{nonempty}: Y \neq \{\}$   
**and**  $x: x \geq 0$   
**shows**  $(\text{SUP } i \in Y. f\ i) * \text{ereal } x = (\text{SUP } i \in Y. f\ i * \text{ereal } x)$  **(is ?lhs = ?rhs)**  
 <proof>

**lemma** *Sup-ereal-mult-left'*:  
 $\llbracket Y \neq \{\}; x \geq 0 \rrbracket \Longrightarrow \text{ereal } x * (\text{SUP } i \in Y. f\ i) = (\text{SUP } i \in Y. \text{ereal } x * f\ i)$   
 <proof>

**lemma** *sup-continuous-add[order-continuous-intros]*:  
**fixes**  $f\ g :: 'a::\text{complete-lattice} \Rightarrow \text{ereal}$   
**assumes**  $nn: \bigwedge x. 0 \leq f\ x \wedge x. 0 \leq g\ x$  **and**  $\text{cont}: \text{sup-continuous } f\ \text{sup-continuous } g$   
**shows**  $\text{sup-continuous } (\lambda x. f\ x + g\ x)$   
 <proof>

**lemma** *sup-continuous-mult-right[order-continuous-intros]*:  
 $0 \leq c \Longrightarrow c < \infty \Longrightarrow \text{sup-continuous } f \Longrightarrow \text{sup-continuous } (\lambda x. f\ x * c :: \text{ereal})$   
 <proof>

**lemma** *sup-continuous-mult-left[order-continuous-intros]*:  
 $0 \leq c \Longrightarrow c < \infty \Longrightarrow \text{sup-continuous } f \Longrightarrow \text{sup-continuous } (\lambda x. c * f\ x :: \text{ereal})$   
 <proof>



**lemma** *sup-continuous-ereal-of-enat*[*order-continuous-intros*]:  
**assumes**  $f$ : *sup-continuous*  $f$   
**shows** *sup-continuous*  $(\lambda x. \text{ereal-of-enat } (f x))$   
 $\langle \text{proof} \rangle$

### 38.6.2 Sums

**lemma** *sums-ereal-positive*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge i. 0 \leq f i$   
**shows**  $f \text{ sums } (\text{SUP } n. \sum i < n. f i)$   
 $\langle \text{proof} \rangle$

**lemma** *summable-ereal-pos*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge i. 0 \leq f i$   
**shows** *summable*  $f$   
 $\langle \text{proof} \rangle$

**lemma** *sums-ereal*:  $(\lambda x. \text{ereal } (f x)) \text{ sums } \text{ereal } x \longleftrightarrow f \text{ sums } x$   
 $\langle \text{proof} \rangle$

**lemma** *suminf-ereal-eq-SUP*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge i. 0 \leq f i$   
**shows**  $(\sum x. f x) = (\text{SUP } n. \sum i < n. f i)$   
 $\langle \text{proof} \rangle$

**lemma** *suminf-bound*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $\forall N. (\sum n < N. f n) \leq x \wedge \bigwedge n. 0 \leq f n$   
**shows**  $\text{suminf } f \leq x$   
 $\langle \text{proof} \rangle$

**lemma** *suminf-bound-add*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $\forall N. (\sum n < N. f n) + y \leq x$   
**and**  $\bigwedge n. 0 \leq f n$   
**and**  $y \neq -\infty$   
**shows**  $\text{suminf } f + y \leq x$   
 $\langle \text{proof} \rangle$

**lemma** *suminf-upper*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge n. 0 \leq f n$   
**shows**  $(\sum n < N. f n) \leq (\sum n. f n)$   
 $\langle \text{proof} \rangle$

**lemma** *suminf-0-le*:

**fixes**  $f :: nat \Rightarrow ereal$   
**assumes**  $\bigwedge n. 0 \leq f\ n$   
**shows**  $0 \leq (\sum n. f\ n)$   
 $\langle proof \rangle$

**lemma** *suminf-le-pos*:

**fixes**  $f\ g :: nat \Rightarrow ereal$   
**assumes**  $\bigwedge N. f\ N \leq g\ N$   
**and**  $\bigwedge N. 0 \leq f\ N$   
**shows**  $suminf\ f \leq suminf\ g$   
 $\langle proof \rangle$

**lemma** *suminf-half-series-ereal*:  $(\sum n. (1/2 :: ereal) \wedge Suc\ n) = 1$   
 $\langle proof \rangle$

**lemma** *suminf-add-ereal*:

**fixes**  $f\ g :: nat \Rightarrow ereal$   
**assumes**  $\bigwedge i. 0 \leq f\ i \wedge i. 0 \leq g\ i$   
**shows**  $(\sum i. f\ i + g\ i) = suminf\ f + suminf\ g$   
 $\langle proof \rangle$

**lemma** *suminf-cmult-ereal*:

**fixes**  $f\ g :: nat \Rightarrow ereal$   
**assumes**  $\bigwedge i. 0 \leq f\ i$  **and**  $0 \leq a$   
**shows**  $(\sum i. a * f\ i) = a * suminf\ f$   
 $\langle proof \rangle$

**lemma** *suminf-PInfty*:

**fixes**  $f :: nat \Rightarrow ereal$   
**assumes**  $\bigwedge i. 0 \leq f\ i$   
**and**  $suminf\ f \neq \infty$   
**shows**  $f\ i \neq \infty$   
 $\langle proof \rangle$

**lemma** *suminf-PInfty-fun*:

**assumes**  $\bigwedge i. 0 \leq f\ i$   
**and**  $suminf\ f \neq \infty$   
**shows**  $\exists f'. f = (\lambda x. ereal\ (f'\ x))$   
 $\langle proof \rangle$

**lemma** *summable-ereal*:

**assumes**  $\bigwedge i. 0 \leq f\ i$   
**and**  $(\sum i. ereal\ (f\ i)) \neq \infty$   
**shows** *summable*  $f$   
 $\langle proof \rangle$

**lemma** *suminf-ereal*:

**assumes**  $\bigwedge i. 0 \leq f\ i$

**and**  $(\sum i. \text{ereal } (f i)) \neq \infty$   
**shows**  $(\sum i. \text{ereal } (f i)) = \text{ereal } (\text{suminf } f)$   
 ⟨proof⟩

**lemma** *suminf-ereal-minus*:  
**fixes**  $f g :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $\text{ord}: \bigwedge i. g i \leq f i \wedge i. 0 \leq g i$   
**and**  $\text{fin}: \text{suminf } f \neq \infty \text{ suminf } g \neq \infty$   
**shows**  $(\sum i. f i - g i) = \text{suminf } f - \text{suminf } g$   
 ⟨proof⟩

**lemma** *suminf-ereal-PIInf [simp]*:  $(\sum x. \infty :: \text{ereal}) = \infty$   
 ⟨proof⟩

**lemma** *summable-real-of-ereal*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $f: \bigwedge i. 0 \leq f i$   
**and**  $\text{fin}: (\sum i. f i) \neq \infty$   
**shows** *summable*  $(\lambda i. \text{real-of-ereal } (f i))$   
 ⟨proof⟩

**lemma** *suminf-SUP-eq*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge i. \text{incseq } (\lambda n. f n i)$   
**and**  $\bigwedge n i. 0 \leq f n i$   
**shows**  $(\sum i. \text{SUP } n. f n i) = (\text{SUP } n. \sum i. f n i)$   
 ⟨proof⟩

**lemma** *suminf-sum-ereal*:  
**fixes**  $f :: - \Rightarrow - \Rightarrow \text{ereal}$   
**assumes**  $\text{nonneg}: \bigwedge i a. a \in A \implies 0 \leq f i a$   
**shows**  $(\sum i. \sum a \in A. f i a) = (\sum a \in A. \sum i. f i a)$   
 ⟨proof⟩

**lemma** *suminf-ereal-eq-0*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $\text{nneg}: \bigwedge i. 0 \leq f i$   
**shows**  $(\sum i. f i) = 0 \iff (\forall i. f i = 0)$   
 ⟨proof⟩

**lemma** *suminf-ereal-offset-le*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $f: \bigwedge i. 0 \leq f i$   
**shows**  $(\sum i. f (i + k)) \leq \text{suminf } f$   
 ⟨proof⟩

**lemma** *sums-suminf-ereal*:  $f \text{ sums } x \implies (\sum i. \text{ereal } (f i)) = \text{ereal } x$   
 ⟨proof⟩

**lemma** *suminf-ereal'*:  $\text{summable } f \implies (\sum i. \text{ereal } (f i)) = \text{ereal } (\sum i. f i)$   
 ⟨proof⟩

**lemma** *suminf-ereal-finite*:  $\text{summable } f \implies (\sum i. \text{ereal } (f i)) \neq \infty$   
 ⟨proof⟩

**lemma** *suminf-ereal-finite-neg*:  
**assumes** *summable*  $f$   
**shows**  $(\sum x. \text{ereal } (f x)) \neq -\infty$   
 ⟨proof⟩

**lemma** *SUP-ereal-add-directed*:  
**fixes**  $f g :: 'a \Rightarrow \text{ereal}$   
**assumes** *nonneg*:  $\bigwedge i. i \in I \implies 0 \leq f i \wedge i. i \in I \implies 0 \leq g i$   
**assumes** *directed*:  $\bigwedge i j. i \in I \implies j \in I \implies \exists k \in I. f i + g j \leq f k + g k$   
**shows**  $(\text{SUP } i \in I. f i + g i) = (\text{SUP } i \in I. f i) + (\text{SUP } i \in I. g i)$   
 ⟨proof⟩

**lemma** *SUP-ereal-sum-directed*:  
**fixes**  $f g :: 'a \Rightarrow 'b \Rightarrow \text{ereal}$   
**assumes**  $I \neq \{\}$   
**assumes** *directed*:  $\bigwedge N i j. N \subseteq A \implies i \in I \implies j \in I \implies \exists k \in I. \forall n \in N. f n i \leq f n k \wedge f n j \leq f n k$   
**assumes** *nonneg*:  $\bigwedge n i. i \in I \implies n \in A \implies 0 \leq f n i$   
**shows**  $(\text{SUP } i \in I. \sum n \in A. f n i) = (\sum n \in A. \text{SUP } i \in I. f n i)$   
 ⟨proof⟩

**lemma** *suminf-SUP-eq-directed*:  
**fixes**  $f :: - \Rightarrow \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $I \neq \{\}$   
**assumes** *directed*:  $\bigwedge N i j. i \in I \implies j \in I \implies \text{finite } N \implies \exists k \in I. \forall n \in N. f i n \leq f k n \wedge f j n \leq f k n$   
**assumes** *nonneg*:  $\bigwedge n i. 0 \leq f n i$   
**shows**  $(\sum i. \text{SUP } n \in I. f n i) = (\text{SUP } n \in I. \sum i. f n i)$   
 ⟨proof⟩

**lemma** *ereal-dense3*:  
**fixes**  $x y :: \text{ereal}$   
**shows**  $x < y \implies \exists r :: \text{rat}. x < \text{real-of-rat } r \wedge \text{real-of-rat } r < y$   
 ⟨proof⟩

**lemma** *continuous-within-ereal*[*intro, simp*]:  $x \in A \implies \text{continuous (at } x \text{ within } A)$   
*ereal*  
 ⟨proof⟩

**lemma** *ereal-open-uminus*:  
**fixes**  $S :: \text{ereal set}$   
**assumes** *open*  $S$   
**shows** *open* (*uminus* ‘  $S$ )

*<proof>*

**lemma** *ereal-uminus-complement*:

**fixes**  $S :: \text{ereal set}$

**shows**  $\text{uminus } '(- S) = - \text{uminus } ' S$

*<proof>*

**lemma** *ereal-closed-uminus*:

**fixes**  $S :: \text{ereal set}$

**assumes**  $\text{closed } S$

**shows**  $\text{closed } (\text{uminus } ' S)$

*<proof>*

**lemma** *ereal-open-affinity-pos*:

**fixes**  $S :: \text{ereal set}$

**assumes**  $\text{open } S$

**and**  $m: m \neq \infty \ 0 < m$

**and**  $t: |t| \neq \infty$

**shows**  $\text{open } ((\lambda x. m * x + t) ' S)$

*<proof>*

**lemma** *ereal-open-affinity*:

**fixes**  $S :: \text{ereal set}$

**assumes**  $\text{open } S$

**and**  $m: |m| \neq \infty \ m \neq 0$

**and**  $t: |t| \neq \infty$

**shows**  $\text{open } ((\lambda x. m * x + t) ' S)$

*<proof>*

**lemma** *open-uminus-iff*:

**fixes**  $S :: \text{ereal set}$

**shows**  $\text{open } (\text{uminus } ' S) \longleftrightarrow \text{open } S$

*<proof>*

**lemma** *ereal-Liminf-uminus*:

**fixes**  $f :: 'a \Rightarrow \text{ereal}$

**shows**  $\text{Liminf net } (\lambda x. - (f x)) = - \text{Limsup net } f$

*<proof>*

**lemma** *Liminf-PIfty*:

**fixes**  $f :: 'a \Rightarrow \text{ereal}$

**assumes**  $\neg \text{trivial-limit net}$

**shows**  $(f \longrightarrow \infty) \text{ net} \longleftrightarrow \text{Liminf net } f = \infty$

*<proof>*

**lemma** *Limsup-MIfty*:

**fixes**  $f :: 'a \Rightarrow \text{ereal}$

**assumes**  $\neg \text{trivial-limit net}$

**shows**  $(f \longrightarrow -\infty) \text{ net} \longleftrightarrow \text{Limsup net } f = -\infty$

*<proof>*

**lemma** *convergent-ereal*: — RENAME  
**fixes**  $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$   
**shows**  $\text{convergent } X \longleftrightarrow \text{limsup } X = \text{liminf } X$   
*<proof>*

**lemma** *limsup-le-liminf-real*:  
**fixes**  $X :: \text{nat} \Rightarrow \text{real}$  **and**  $L :: \text{real}$   
**assumes** 1:  $\text{limsup } X \leq L$  **and** 2:  $L \leq \text{liminf } X$   
**shows**  $X \longrightarrow L$   
*<proof>*

**lemma** *liminf-PIInfty*:  
**fixes**  $X :: \text{nat} \Rightarrow \text{ereal}$   
**shows**  $X \longrightarrow \infty \longleftrightarrow \text{liminf } X = \infty$   
*<proof>*

**lemma** *limsup-MInfty*:  
**fixes**  $X :: \text{nat} \Rightarrow \text{ereal}$   
**shows**  $X \longrightarrow -\infty \longleftrightarrow \text{limsup } X = -\infty$   
*<proof>*

**lemma** *SUP-eq-LIMSEQ*:  
**assumes** *mono f*  
**shows**  $(\text{SUP } n. \text{ereal } (f n)) = \text{ereal } x \longleftrightarrow f \longrightarrow x$   
*<proof>*

**lemma** *liminf-ereal-cminus*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ereal}$   
**assumes**  $c \neq -\infty$   
**shows**  $\text{liminf } (\lambda x. c - f x) = c - \text{limsup } f$   
*<proof>*

### 38.6.3 Continuity

**lemma** *continuous-at-of-ereal*:  
 $|x0 :: \text{ereal}| \neq \infty \implies \text{continuous } (\text{at } x0) \text{ real-of-ereal}$   
*<proof>*

**lemma** *nhds-ereal*:  $\text{nhds } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{nhds } r)$   
*<proof>*

**lemma** *at-ereal*:  $\text{at } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{at } r)$   
*<proof>*

**lemma** *at-left-ereal*:  $\text{at-left } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{at-left } r)$   
*<proof>*

**lemma** *at-right-ereal*:  $at\_right (ereal\ r) = filtermap\ ereal\ (at\_right\ r)$   
 ⟨proof⟩

**lemma**  
**shows** *at-left-PInf*:  $at\_left\ \infty = filtermap\ ereal\ at\_top$   
**and** *at-right-MInf*:  $at\_right\ (-\infty) = filtermap\ ereal\ at\_bot$   
 ⟨proof⟩

**lemma** *ereal-tendsto-simps1*:  
 $((f \circ real\ of\ ereal) \longrightarrow y)\ (at\_left\ (ereal\ x)) \longleftrightarrow (f \longrightarrow y)\ (at\_left\ x)$   
 $((f \circ real\ of\ ereal) \longrightarrow y)\ (at\_right\ (ereal\ x)) \longleftrightarrow (f \longrightarrow y)\ (at\_right\ x)$   
 $((f \circ real\ of\ ereal) \longrightarrow y)\ (at\_left\ (\infty::ereal)) \longleftrightarrow (f \longrightarrow y)\ at\_top$   
 $((f \circ real\ of\ ereal) \longrightarrow y)\ (at\_right\ (-\infty::ereal)) \longleftrightarrow (f \longrightarrow y)\ at\_bot$   
 ⟨proof⟩

**lemma** *ereal-tendsto-simps2*:  
 $((ereal \circ f) \longrightarrow ereal\ a)\ F \longleftrightarrow (f \longrightarrow a)\ F$   
 $((ereal \circ f) \longrightarrow \infty)\ F \longleftrightarrow (LIM\ x\ F.\ f\ x\ :>\ at\_top)$   
 $((ereal \circ f) \longrightarrow -\infty)\ F \longleftrightarrow (LIM\ x\ F.\ f\ x\ :>\ at\_bot)$   
 ⟨proof⟩

**lemma** *inverse-infty-ereal-tendsto-0*:  $inverse\ -\infty \rightarrow (0::ereal)$   
 ⟨proof⟩

**lemma** *inverse-ereal-tendsto-pos*:  
**fixes**  $x :: ereal$  **assumes**  $0 < x$   
**shows**  $inverse\ -x \rightarrow inverse\ x$   
 ⟨proof⟩

**lemma** *inverse-ereal-tendsto-at-right-0*:  $(inverse\ \longrightarrow\ \infty)\ (at\_right\ (0::ereal))$   
 ⟨proof⟩

**lemmas** *ereal-tendsto-simps* = *ereal-tendsto-simps1* *ereal-tendsto-simps2*

**lemma** *continuous-at-iff-ereal*:  
**fixes**  $f :: 'a::t2\_space \Rightarrow real$   
**shows**  $continuous\ (at\ x0\ within\ s)\ f \longleftrightarrow continuous\ (at\ x0\ within\ s)\ (ereal \circ f)$   
 ⟨proof⟩

**lemma** *continuous-on-iff-ereal*:  
**fixes**  $f :: 'a::t2\_space \Rightarrow real$   
**assumes** *open*  $A$   
**shows**  $continuous\_on\ A\ f \longleftrightarrow continuous\_on\ A\ (ereal \circ f)$   
 ⟨proof⟩

**lemma** *continuous-on-real*:  $continuous\_on\ (UNIV - \{\infty, -\infty::ereal\})\ real\ of\ ereal$   
 ⟨proof⟩

**lemma** *continuous-on-iff-real*:

**fixes**  $f :: 'a::t2\text{-space} \Rightarrow \text{ereal}$   
**assumes**  $\bigwedge x. x \in A \implies |f\ x| \neq \infty$   
**shows**  $\text{continuous-on } A\ f \longleftrightarrow \text{continuous-on } A\ (\text{real-of-ereal } \circ f)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{continuous-uminus-ereal}$  [*continuous-intros*]:  $\text{continuous-on } (A :: \text{ereal set})$   
 $\text{uminus}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{ereal-uminus-atMost}$  [*simp*]:  $\text{uminus } \{..(a::\text{ereal})\} = \{-a..\}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{continuous-on-inverse-ereal}$  [*continuous-intros*]:  
 $\text{continuous-on } \{0::\text{ereal } ..\} \text{ inverse}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{continuous-inverse-ereal-nonpos}$ :  $\text{continuous-on } (\{..<0\} :: \text{ereal set})$  *inverse*  
 $\langle \text{proof} \rangle$

**lemma**  $\text{tendsto-inverse-ereal}$ :  
**assumes**  $(f \longrightarrow (c :: \text{ereal}))\ F$   
**assumes**  $\text{eventually } (\lambda x. f\ x \geq 0)\ F$   
**shows**  $((\lambda x. \text{inverse } (f\ x)) \longrightarrow \text{inverse } c)\ F$   
 $\langle \text{proof} \rangle$

### 38.6.4 liminf and limsup

**lemma**  $\text{Limsup-ereal-mult-right}$ :  
**assumes**  $F \neq \text{bot } (c::\text{real}) \geq 0$   
**shows**  $\text{Limsup } F\ (\lambda n. f\ n * \text{ereal } c) = \text{Limsup } F\ f * \text{ereal } c$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{Liminf-ereal-mult-right}$ :  
**assumes**  $F \neq \text{bot } (c::\text{real}) \geq 0$   
**shows**  $\text{Liminf } F\ (\lambda n. f\ n * \text{ereal } c) = \text{Liminf } F\ f * \text{ereal } c$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{Liminf-ereal-mult-left}$ :  
**assumes**  $F \neq \text{bot } (c::\text{real}) \geq 0$   
**shows**  $\text{Liminf } F\ (\lambda n. \text{ereal } c * f\ n) = \text{ereal } c * \text{Liminf } F\ f$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{Limsup-ereal-mult-left}$ :  
**assumes**  $F \neq \text{bot } (c::\text{real}) \geq 0$   
**shows**  $\text{Limsup } F\ (\lambda n. \text{ereal } c * f\ n) = \text{ereal } c * \text{Limsup } F\ f$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{limsup-ereal-mult-right}$ :



$(c::\text{real}) \geq 0 \implies \text{limsup } (\lambda n. f n * \text{ereal } c) = \text{limsup } f * \text{ereal } c$   
 ⟨proof⟩

**lemma** *limsup-ereal-mult-left*:

$(c::\text{real}) \geq 0 \implies \text{limsup } (\lambda n. \text{ereal } c * f n) = \text{ereal } c * \text{limsup } f$   
 ⟨proof⟩

**lemma** *Limsup-add-ereal-right*:

$F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Limsup } F (\lambda n. g n + (c :: \text{ereal})) = \text{Limsup } F g + c$   
 ⟨proof⟩

**lemma** *Limsup-add-ereal-left*:

$F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Limsup } F (\lambda n. (c :: \text{ereal}) + g n) = c + \text{Limsup } F g$   
 ⟨proof⟩

**lemma** *Liminf-add-ereal-right*:

$F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Liminf } F (\lambda n. g n + (c :: \text{ereal})) = \text{Liminf } F g + c$   
 ⟨proof⟩

**lemma** *Liminf-add-ereal-left*:

$F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Liminf } F (\lambda n. (c :: \text{ereal}) + g n) = c + \text{Liminf } F g$   
 ⟨proof⟩

**lemma**

**assumes**  $F \neq \text{bot}$

**assumes** *nonneg: eventually*  $(\lambda x. f x \geq (0::\text{ereal})) F$

**shows** *Liminf-inverse-ereal*:  $\text{Liminf } F (\lambda x. \text{inverse } (f x)) = \text{inverse } (\text{Limsup } F f)$

**and** *Limsup-inverse-ereal*:  $\text{Limsup } F (\lambda x. \text{inverse } (f x)) = \text{inverse } (\text{Liminf } F f)$

⟨proof⟩

**lemma** *ereal-diff-le-mono-left*:  $\llbracket x \leq z; 0 \leq y \rrbracket \implies x - y \leq (z :: \text{ereal})$

⟨proof⟩

**lemma** *neg-0-less-iff-less-erea* [*simp*]:  $0 < - a \longleftrightarrow (a :: \text{ereal}) < 0$

⟨proof⟩

**lemma** *not-infty-ereal*:  $|x| \neq \infty \longleftrightarrow (\exists x'. x = \text{ereal } x')$

⟨proof⟩

**lemma** *neg-PInf-trans*: **fixes**  $x y :: \text{ereal}$  **shows**  $\llbracket y \neq \infty; x \leq y \rrbracket \implies x \neq \infty$

⟨proof⟩

**lemma** *mult-2-ereal*:  $\text{ereal } 2 * x = x + x$

⟨proof⟩

**lemma** *ereal-diff-le-self*:  $0 \leq y \implies x - y \leq (x :: \text{ereal})$   
 ⟨proof⟩

**lemma** *ereal-le-add-self*:  $0 \leq y \implies x \leq x + (y :: \text{ereal})$   
 ⟨proof⟩

**lemma** *ereal-le-add-self2*:  $0 \leq y \implies x \leq y + (x :: \text{ereal})$   
 ⟨proof⟩

**lemma** *ereal-diff-nonpos*:  
**fixes**  $a\ b :: \text{ereal}$  **shows**  $\llbracket a \leq b; a = \infty \implies b \neq \infty; a = -\infty \implies b \neq -\infty \rrbracket$   
 $\implies a - b \leq 0$   
 ⟨proof⟩

**lemma** *minus-ereal-0* [*simp*]:  $x - \text{ereal } 0 = x$   
 ⟨proof⟩

**lemma** *ereal-diff-eq-0-iff*: **fixes**  $a\ b :: \text{ereal}$   
**shows**  $(|a| = \infty \implies |b| \neq \infty) \implies a - b = 0 \longleftrightarrow a = b$   
 ⟨proof⟩

**lemma** *SUP-ereal-eq-0-iff-nonneg*:  
**fixes**  $f :: - \Rightarrow \text{ereal}$  **and**  $A$   
**assumes** *nonneg*:  $\forall x \in A. f\ x \geq 0$   
**and**  $A: A \neq \{\}$   
**shows**  $(\text{SUP } x \in A. f\ x) = 0 \longleftrightarrow (\forall x \in A. f\ x = 0)$  (**is** ?lhs  $\longleftrightarrow$  -)  
 ⟨proof⟩

**lemma** *ereal-divide-le-posI*:  
**fixes**  $x\ y\ z :: \text{ereal}$   
**shows**  $x > 0 \implies z \neq -\infty \implies z \leq x * y \implies z / x \leq y$   
 ⟨proof⟩

**lemma** *add-diff-eq-ereal*:  
**fixes**  $x\ y\ z :: \text{ereal}$   
**shows**  $x + (y - z) = x + y - z$   
 ⟨proof⟩

**lemma** *ereal-diff-gr0*:  
**fixes**  $a\ b :: \text{ereal}$   
**shows**  $a < b \implies 0 < b - a$   
 ⟨proof⟩

**lemma** *ereal-minus-minus*:  
**fixes**  $x\ y\ z :: \text{ereal}$  **shows**  
 $(|y| = \infty \implies |z| \neq \infty) \implies x - (y - z) = x + z - y$   
 ⟨proof⟩

**lemma** *diff-diff-commute-ereal*:

**fixes**  $x y z :: \text{ereal}$   
**shows**  $x - y - z = x - z - y$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-diff-eq-MInfty-iff*:

**fixes**  $x y :: \text{ereal}$   
**shows**  $x - y = -\infty \longleftrightarrow x = -\infty \wedge y \neq -\infty \vee y = \infty \wedge |x| \neq \infty$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-diff-add-inverse*:

**fixes**  $x y :: \text{ereal}$   
**shows**  $|x| \neq \infty \implies x + y - x = y$   
 $\langle \text{proof} \rangle$

**lemma** *tendsto-diff-ereal*:

**fixes**  $x y :: \text{ereal}$   
**assumes**  $x: |x| \neq \infty$  **and**  $y: |y| \neq \infty$   
**assumes**  $f: (f \longrightarrow x) F$  **and**  $g: (g \longrightarrow y) F$   
**shows**  $((\lambda x. f x - g x) \longrightarrow x - y) F$   
 $\langle \text{proof} \rangle$

**lemma** *continuous-on-diff-ereal*:

*continuous-on*  $A f \implies \text{continuous-on } A g \implies (\bigwedge x. x \in A \implies |f x| \neq \infty) \implies$   
 $(\bigwedge x. x \in A \implies |g x| \neq \infty) \implies \text{continuous-on } A (\lambda z. f z - g z :: \text{ereal})$   
 $\langle \text{proof} \rangle$

### 38.6.5 Tests for code generator

A small list of simple arithmetic expressions.

**value**  $-\infty :: \text{ereal}$   
**value**  $|-\infty| :: \text{ereal}$   
**value**  $4 + 5 / 4 - \text{ereal } 2 :: \text{ereal}$   
**value**  $\text{ereal } 3 < \infty$   
**value** *real-of-ereal*  $(\infty :: \text{ereal}) = 0$

**end**

## 39 Indicator Function

**theory** *Indicator-Function*  
**imports** *Complex-Main Disjoint-Sets*  
**begin**

**definition** *indicator*  $S x = \text{of-bool } (x \in S)$

Type constrained version

**abbreviation** *indicat-real*  $:: 'a \text{ set} \Rightarrow 'a \Rightarrow \text{real}$  **where** *indicat-real*  $S \equiv \text{indicator } S$

**lemma** *indicator-simps*[simp]:

$x \in S \implies \text{indicator } S \ x = 1$   
 $x \notin S \implies \text{indicator } S \ x = 0$   
 ⟨proof⟩

**lemma** *indicator-pos-le*[intro, simp]:  $(0 :: 'a :: \text{linordered-semidom}) \leq \text{indicator } S \ x$   
**and** *indicator-le-1*[intro, simp]:  $\text{indicator } S \ x \leq (1 :: 'a :: \text{linordered-semidom})$   
 ⟨proof⟩

**lemma** *indicator-abs-le-1*:  $|\text{indicator } S \ x| \leq (1 :: 'a :: \text{linordered-idom})$   
 ⟨proof⟩

**lemma** *indicator-eq-0-iff*:  $\text{indicator } A \ x = (0 :: 'a :: \text{zero-neq-one}) \longleftrightarrow x \notin A$   
 ⟨proof⟩

**lemma** *indicator-eq-1-iff*:  $\text{indicator } A \ x = (1 :: 'a :: \text{zero-neq-one}) \longleftrightarrow x \in A$   
 ⟨proof⟩

**lemma** *indicator-UNIV* [simp]:  $\text{indicator } UNIV = (\lambda x. 1)$   
 ⟨proof⟩

**lemma** *indicator-leI*:

$(x \in A \implies y \in B) \implies (\text{indicator } A \ x :: 'a :: \text{linordered-nonzero-semiring}) \leq \text{indicator } B \ y$   
 ⟨proof⟩

**lemma** *split-indicator*:  $P (\text{indicator } S \ x) \longleftrightarrow ((x \in S \longrightarrow P \ 1) \wedge (x \notin S \longrightarrow P \ 0))$   
 ⟨proof⟩

**lemma** *split-indicator-asm*:  $P (\text{indicator } S \ x) \longleftrightarrow (\neg (x \in S \wedge \neg P \ 1 \vee x \notin S \wedge \neg P \ 0))$   
 ⟨proof⟩

**lemma** *indicator-inter-arith*:  $\text{indicator } (A \cap B) \ x = \text{indicator } A \ x * (\text{indicator } B \ x :: 'a :: \text{semiring-1})$   
 ⟨proof⟩

**lemma** *indicator-union-arith*:

$\text{indicator } (A \cup B) \ x = \text{indicator } A \ x + \text{indicator } B \ x - \text{indicator } A \ x * (\text{indicator } B \ x :: 'a :: \text{ring-1})$   
 ⟨proof⟩

**lemma** *indicator-inter-min*:  $\text{indicator } (A \cap B) \ x = \min (\text{indicator } A \ x) (\text{indicator } B \ x :: 'a :: \text{linordered-semidom})$

**and** *indicator-union-max*:  $\text{indicator } (A \cup B) \ x = \max (\text{indicator } A \ x) (\text{indicator } B \ x :: 'a :: \text{linordered-semidom})$   
 ⟨proof⟩

**lemma** *indicator-disj-union*:

$A \cap B = \{\} \implies \text{indicator } (A \cup B) x = (\text{indicator } A x + \text{indicator } B x :: 'a::\text{linordered-semidom})$   
 ⟨proof⟩

**lemma** *indicator-compl*:  $\text{indicator } (- A) x = 1 - (\text{indicator } A x :: 'a::\text{ring-1})$

**and** *indicator-diff*:  $\text{indicator } (A - B) x = \text{indicator } A x * (1 - \text{indicator } B x :: 'a::\text{ring-1})$   
 ⟨proof⟩

**lemma** *indicator-times*:

$\text{indicator } (A \times B) x = \text{indicator } A (\text{fst } x) * (\text{indicator } B (\text{snd } x) :: 'a::\text{semiring-1})$   
 ⟨proof⟩

**lemma** *indicator-sum*:

$\text{indicator } (A <+> B) x = (\text{case } x \text{ of } \text{Inl } x \implies \text{indicator } A x \mid \text{Inr } x \implies \text{indicator } B x)$   
 ⟨proof⟩

**lemma** *indicator-image*:  $\text{inj } f \implies \text{indicator } (f ' X) (f x) = (\text{indicator } X x :: \text{zero-neq-one})$

⟨proof⟩

**lemma** *indicator-vimage*:  $\text{indicator } (f -' A) x = \text{indicator } A (f x)$

⟨proof⟩

**lemma** *mult-indicator-cong*:

**fixes**  $f g :: - \implies 'a :: \text{semiring-1}$

**shows**  $(\bigwedge x. x \in A \implies f x = g x) \implies \text{indicator } A x * f x = \text{indicator } A x * g x$   
 ⟨proof⟩

**lemma**

**fixes**  $f :: 'a \implies 'b :: \text{semiring-1}$

**assumes** *finite A*

**shows** *sum-mult-indicator[simp]*:  $(\sum x \in A. f x * \text{indicator } B x) = (\sum x \in A \cap B. f x)$

**and** *sum-indicator-mult[simp]*:  $(\sum x \in A. \text{indicator } B x * f x) = (\sum x \in A \cap B. f x)$

⟨proof⟩

**lemma** *sum-indicator-eq-card*:

**assumes** *finite A*

**shows**  $(\sum x \in A. \text{indicator } B x) = \text{card } (A \text{ Int } B)$

⟨proof⟩

**lemma** *sum-indicator-scaleR[simp]*:

*finite A*  $\implies$

$(\sum x \in A. \text{indicator } (B x) (g x) *_R f x) = (\sum x \in \{x \in A. g x \in B x\}. f x :: 'a::\text{real-vector})$

*<proof>*

**lemma** *LIMSEQ-indicator-incseq*:

**assumes** *incseq A*

**shows**  $(\lambda i. \text{indicator } (A \ i) \ x :: 'a::\{\text{topological-space, zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcup i. A \ i) \ x$

*<proof>*

**lemma** *LIMSEQ-indicator-UN*:

$(\lambda k. \text{indicator } (\bigcup i < k. A \ i) \ x :: 'a::\{\text{topological-space, zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcup i. A \ i) \ x$

*<proof>*

**lemma** *LIMSEQ-indicator-decseq*:

**assumes** *decseq A*

**shows**  $(\lambda i. \text{indicator } (A \ i) \ x :: 'a::\{\text{topological-space, zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcap i. A \ i) \ x$

*<proof>*

**lemma** *LIMSEQ-indicator-INT*:

$(\lambda k. \text{indicator } (\bigcap i < k. A \ i) \ x :: 'a::\{\text{topological-space, zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcap i. A \ i) \ x$

*<proof>*

**lemma** *indicator-add*:

$A \cap B = \{\} \implies (\text{indicator } A \ x :: \text{monoid-add}) + \text{indicator } B \ x = \text{indicator } (A \cup B) \ x$

*<proof>*

**lemma** *of-real-indicator*:  $\text{of-real } (\text{indicator } A \ x) = \text{indicator } A \ x$

*<proof>*

**lemma** *real-of-nat-indicator*:  $\text{real } (\text{indicator } A \ x :: \text{nat}) = \text{indicator } A \ x$

*<proof>*

**lemma** *abs-indicator*:  $|\text{indicator } A \ x :: 'a::\text{linordered-idom}| = \text{indicator } A \ x$

*<proof>*

**lemma** *mult-indicator-subset*:

$A \subseteq B \implies \text{indicator } A \ x * \text{indicator } B \ x = (\text{indicator } A \ x :: 'a::\text{comm-semiring-1})$

*<proof>*

**lemma** *indicator-times-eq-if*:

**fixes**  $f :: 'a \Rightarrow 'b::\text{comm-ring-1}$

**shows**  $\text{indicator } S \ x * f \ x = (\text{if } x \in S \text{ then } f \ x \text{ else } 0) \ f \ x * \text{indicator } S \ x = (\text{if } x \in S \text{ then } f \ x \text{ else } 0)$

*<proof>*

**lemma** *indicator-scaleR-eq-if*:

**fixes**  $f :: 'a \Rightarrow 'b::\text{real-vector}$   
**shows**  $\text{indicator } S \ x \ *_R \ f \ x = (\text{if } x \in S \ \text{then } f \ x \ \text{else } 0)$   
 $\langle \text{proof} \rangle$

**lemma** *indicator-sums*:  
**assumes**  $\bigwedge i \ j. \ i \neq j \implies A \ i \cap A \ j = \{\}$   
**shows**  $(\lambda i. \ \text{indicator } (A \ i) \ x::\text{real}) \ \text{sums} \ \text{indicator } (\bigcup i. \ A \ i) \ x$   
 $\langle \text{proof} \rangle$

The indicator function of the union of a disjoint family of sets is the sum over all the individual indicators.

**lemma** *indicator-UN-disjoint*:  
 $\text{finite } A \implies \text{disjoint-family-on } f \ A \implies \text{indicator } (\bigcup (f \ 'A)) \ x = (\sum y \in A. \ \text{indicator } (f \ y) \ x)$   
 $\langle \text{proof} \rangle$

**end**

## 40 The type of non-negative extended real numbers

**theory** *Extended-Nonnegative-Real*  
**imports** *Extended-Real Indicator-Function*  
**begin**

**lemma** *ereal-ineq-diff-add*:  
**assumes**  $b \neq (-\infty::\text{ereal}) \ a \geq b$   
**shows**  $a = b + (a - b)$   
 $\langle \text{proof} \rangle$

**lemma** *Limsup-const-add*:  
**fixes**  $c :: 'a::\{\text{complete-linorder}, \text{linorder-topology}, \text{topological-monoid-add}, \text{ordered-ab-semigroup-add}\}$   
**shows**  $F \neq \text{bot} \implies \text{Limsup } F \ (\lambda x. \ c + f \ x) = c + \text{Limsup } F \ f$   
 $\langle \text{proof} \rangle$

**lemma** *Liminf-const-add*:  
**fixes**  $c :: 'a::\{\text{complete-linorder}, \text{linorder-topology}, \text{topological-monoid-add}, \text{ordered-ab-semigroup-add}\}$   
**shows**  $F \neq \text{bot} \implies \text{Liminf } F \ (\lambda x. \ c + f \ x) = c + \text{Liminf } F \ f$   
 $\langle \text{proof} \rangle$

**lemma** *Liminf-add-const*:  
**fixes**  $c :: 'a::\{\text{complete-linorder}, \text{linorder-topology}, \text{topological-monoid-add}, \text{ordered-ab-semigroup-add}\}$   
**shows**  $F \neq \text{bot} \implies \text{Liminf } F \ (\lambda x. \ f \ x + c) = \text{Liminf } F \ f + c$   
 $\langle \text{proof} \rangle$

**lemma** *sums-offset*:

**fixes**  $f g :: nat \Rightarrow 'a :: \{t2\text{-space, topological-comm-monoid-add}\}$   
**assumes**  $(\lambda n. f (n + i)) \text{ sums } l$  **shows**  $f \text{ sums } (l + (\sum j < i. f j))$   
 $\langle \text{proof} \rangle$

**lemma** *suminf-offset*:

**fixes**  $f g :: nat \Rightarrow 'a :: \{t2\text{-space, topological-comm-monoid-add}\}$   
**shows**  $\text{summable } (\lambda j. f (j + i)) \implies \text{suminf } f = (\sum j. f (j + i)) + (\sum j < i. f j)$   
 $\langle \text{proof} \rangle$

**lemma** *eventually-at-left-1*:  $(\bigwedge z :: \text{real}. 0 < z \implies z < 1 \implies P z) \implies \text{eventually } P \text{ (at-left } 1)$   
 $\langle \text{proof} \rangle$

**lemma** *mult-eq-1*:

**fixes**  $a b :: 'a :: \{\text{ordered-semiring, comm-monoid-mult}\}$   
**shows**  $0 \leq a \implies a \leq 1 \implies b \leq 1 \implies a * b = 1 \longleftrightarrow (a = 1 \wedge b = 1)$   
 $\langle \text{proof} \rangle$

**lemma** *ereal-add-diff-cancel*:

**fixes**  $a b :: \text{ereal}$   
**shows**  $|b| \neq \infty \implies (a + b) - b = a$   
 $\langle \text{proof} \rangle$

**lemma** *add-top*:

**fixes**  $x :: 'a :: \{\text{order-top, ordered-comm-monoid-add}\}$   
**shows**  $0 \leq x \implies x + \text{top} = \text{top}$   
 $\langle \text{proof} \rangle$

**lemma** *top-add*:

**fixes**  $x :: 'a :: \{\text{order-top, ordered-comm-monoid-add}\}$   
**shows**  $0 \leq x \implies \text{top} + x = \text{top}$   
 $\langle \text{proof} \rangle$

**lemma** *le-lfp*:  $\text{mono } f \implies x \leq \text{lfp } f \implies f x \leq \text{lfp } f$   
 $\langle \text{proof} \rangle$

**lemma** *lfp-transfer*:

**assumes**  $\alpha$ : *sup-continuous*  $\alpha$  **and**  $f$ : *sup-continuous*  $f$  **and**  $mg$ : *mono*  $g$   
**assumes**  $\text{bot}$ :  $\alpha \text{ bot} \leq \text{lfp } g$  **and**  $\text{eq}$ :  $\bigwedge x. x \leq \text{lfp } f \implies \alpha (f x) = g (\alpha x)$   
**shows**  $\alpha (\text{lfp } f) = \text{lfp } g$   
 $\langle \text{proof} \rangle$

**lemma** *sup-continuous-applyD*: *sup-continuous*  $f \implies \text{sup-continuous } (\lambda x. f x h)$   
 $\langle \text{proof} \rangle$

**lemma** *sup-continuous-SUP*[*order-continuous-intros*]:

**fixes**  $M :: - \Rightarrow - \Rightarrow 'a :: \text{complete-lattice}$   
**assumes**  $M$ :  $\bigwedge i. i \in I \implies \text{sup-continuous } (M i)$



**shows** *sup-continuous* ( $SUP\ i \in I. M\ i$ )  
 ⟨*proof*⟩

**lemma** *sup-continuous-apply-SUP*[*order-continuous-intros*]:

**fixes**  $M :: - \Rightarrow - \Rightarrow 'a::\text{complete-lattice}$

**shows**  $(\bigwedge i. i \in I \Rightarrow \text{sup-continuous } (M\ i)) \Rightarrow \text{sup-continuous } (\lambda x. SUP\ i \in I. M\ i\ x)$

⟨*proof*⟩

**lemma** *sup-continuous-lfp'*[*order-continuous-intros*]:

**assumes** 1: *sup-continuous*  $f$

**assumes** 2:  $\bigwedge g. \text{sup-continuous } g \Rightarrow \text{sup-continuous } (f\ g)$

**shows** *sup-continuous* ( $\text{lfp } f$ )

⟨*proof*⟩

**lemma** *sup-continuous-lfp''*[*order-continuous-intros*]:

**assumes** 1:  $\bigwedge s. \text{sup-continuous } (f\ s)$

**assumes** 2:  $\bigwedge g. \text{sup-continuous } g \Rightarrow \text{sup-continuous } (\lambda s. f\ s\ (g\ s))$

**shows** *sup-continuous*  $(\lambda x. \text{lfp } (f\ x))$

⟨*proof*⟩

**lemma** *mono-INF-fun*:

$(\bigwedge x\ y. \text{mono } (F\ x\ y)) \Rightarrow \text{mono } (\lambda z\ x. INF\ y \in X\ x. F\ x\ y\ z :: 'a :: \text{complete-lattice})$

⟨*proof*⟩

**lemma** *continuous-on-cmult-ereal*:

$|c::\text{ereal}| \neq \infty \Rightarrow \text{continuous-on } A\ f \Rightarrow \text{continuous-on } A\ (\lambda x. c * f\ x)$

⟨*proof*⟩

**lemma** *real-of-nat-Sup*:

**assumes**  $A \neq \{\}$  *bdd-above*  $A$

**shows** *of-nat* ( $\text{Sup } A$ ) =  $(SUP\ a \in A. \text{of-nat } a :: \text{real})$

⟨*proof*⟩

**lemma** (*in complete-lattice*) *SUP-sup-const1*:

$I \neq \{\} \Rightarrow (SUP\ i \in I. \text{sup } c\ (f\ i)) = \text{sup } c\ (SUP\ i \in I. f\ i)$

⟨*proof*⟩

**lemma** (*in complete-lattice*) *SUP-sup-const2*:

$I \neq \{\} \Rightarrow (SUP\ i \in I. \text{sup } (f\ i)\ c) = \text{sup } (SUP\ i \in I. f\ i)\ c$

⟨*proof*⟩

**lemma** *one-less-of-natD*:

**assumes**  $(1::'a::\text{linordered-semidom}) < \text{of-nat } n$  **shows**  $1 < n$

⟨*proof*⟩

## 40.1 Defining the extended non-negative reals

Basic definitions and type class setup

```
typedef ennreal = {x :: ereal. 0 ≤ x}
morphisms enn2ereal e2ennreal'
  ⟨proof⟩
```

```
definition e2ennreal x = e2ennreal' (max 0 x)
```

```
lemma enn2ereal-range: e2ennreal ‘{0..} = UNIV
  ⟨proof⟩
```

```
lemma type-definition-ennreal': type-definition enn2ereal e2ennreal {x. 0 ≤ x}
  ⟨proof⟩
```

```
setup-lifting type-definition-ennreal'
```

```
declare [[coercion e2ennreal]]
```

```
instantiation ennreal :: complete-linorder
begin
```

```
lift-definition top-ennreal :: ennreal is top ⟨proof⟩
```

```
lift-definition bot-ennreal :: ennreal is 0 ⟨proof⟩
```

```
lift-definition sup-ennreal :: ennreal ⇒ ennreal ⇒ ennreal is sup ⟨proof⟩
```

```
lift-definition inf-ennreal :: ennreal ⇒ ennreal ⇒ ennreal is inf ⟨proof⟩
```

```
lift-definition Inf-ennreal :: ennreal set ⇒ ennreal is Inf
  ⟨proof⟩
```

```
lift-definition Sup-ennreal :: ennreal set ⇒ ennreal is sup 0 ∘ Sup
  ⟨proof⟩
```

```
lift-definition less-eq-ennreal :: ennreal ⇒ ennreal ⇒ bool is (≤) ⟨proof⟩
```

```
lift-definition less-ennreal :: ennreal ⇒ ennreal ⇒ bool is (<) ⟨proof⟩
```

```
instance
  ⟨proof⟩
```

```
end
```

```
lemma pcr-ennreal-enn2ereal[simp]: pcr-ennreal (enn2ereal x) x
  ⟨proof⟩
```

```
lemma rel-fun-eq-pcr-ennreal: rel-fun (=) pcr-ennreal f g ↔ f = enn2ereal ∘ g
  ⟨proof⟩
```

```
instantiation ennreal :: infinity
begin
```

```

definition infinity-ennreal :: ennreal
  where [simp]:  $\infty = (top::ennreal)$ 

instance  $\langle proof \rangle$ 

end

instantiation ennreal :: {semiring-1-no-zero-divisors, comm-semiring-1}
begin

lift-definition one-ennreal :: ennreal is 1  $\langle proof \rangle$ 
lift-definition zero-ennreal :: ennreal is 0  $\langle proof \rangle$ 
lift-definition plus-ennreal :: ennreal  $\Rightarrow$  ennreal  $\Rightarrow$  ennreal is (+)  $\langle proof \rangle$ 
lift-definition times-ennreal :: ennreal  $\Rightarrow$  ennreal  $\Rightarrow$  ennreal is (*)  $\langle proof \rangle$ 

instance
   $\langle proof \rangle$ 

end

instantiation ennreal :: minus
begin

lift-definition minus-ennreal :: ennreal  $\Rightarrow$  ennreal  $\Rightarrow$  ennreal is  $\lambda a b. max\ 0\ (a - b)$ 
   $\langle proof \rangle$ 

instance  $\langle proof \rangle$ 

end

instance ennreal :: numeral  $\langle proof \rangle$ 

instantiation ennreal :: inverse
begin

lift-definition inverse-ennreal :: ennreal  $\Rightarrow$  ennreal is inverse
   $\langle proof \rangle$ 

definition divide-ennreal :: ennreal  $\Rightarrow$  ennreal  $\Rightarrow$  ennreal
  where  $x\ div\ y = x * inverse\ (y :: ennreal)$ 

instance  $\langle proof \rangle$ 

end

lemma ennreal-zero-less-one:  $0 < (1::ennreal)$  — TODO: remove
   $\langle proof \rangle$ 

```

**instance** *ennreal* :: *dioid*

*<proof>*

**instance** *ennreal* :: *ordered-comm-semiring*

*<proof>*

**instance** *ennreal* :: *linordered-nonzero-semiring*

*<proof>*

**instance** *ennreal* :: *strict-ordered-ab-semigroup-add*

*<proof>*

**declare** [[*coercion of-nat* :: *nat*  $\Rightarrow$  *ennreal*]]

**lemma** *e2ennreal-neg*:  $x \leq 0 \implies e2ennreal\ x = 0$

*<proof>*

**lemma** *e2ennreal-mono*:  $x \leq y \implies e2ennreal\ x \leq e2ennreal\ y$

*<proof>*

**lemma** *enn2ereal-nonneg[simp]*:  $0 \leq enn2ereal\ x$

*<proof>*

**lemma** *ereal-ennreal-cases*:

**obtains** *b* **where**  $0 \leq a$   $a = enn2ereal\ b$  |  $a < 0$

*<proof>*

**lemma** *rel-fun-liminf[transfer-rule]*: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal*  
*liminf* *liminf*

*<proof>*

**lemma** *rel-fun-limsup[transfer-rule]*: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal*  
*limsup* *limsup*

*<proof>*

**lemma** *sum-enn2ereal[simp]*:  $(\bigwedge i. i \in I \implies 0 \leq f\ i) \implies (\sum i \in I. enn2ereal\ (f\ i))$   
 $= enn2ereal\ (sum\ f\ I)$

*<proof>*

**lemma** *transfer-e2ennreal-sum [transfer-rule]*:

*rel-fun* (*rel-fun* (=) *pcr-ennreal*) (*rel-fun* (=) *pcr-ennreal*) *sum* *sum*

*<proof>*

**lemma** *enn2ereal-of-nat[simp]*:  $enn2ereal\ (of-nat\ n) = ereal\ n$

*<proof>*

**lemma** *enn2ereal-numeral[simp]*:  $enn2ereal\ (numeral\ a) = numeral\ a$

*<proof>*

**lemma** *transfer-numeral[transfer-rule]*: *pcr-ennreal* (numeral *a*) (numeral *a*)  
 ⟨*proof*⟩

## 40.2 Cancellation simprocs

**lemma** *ennreal-add-left-cancel*:  $a + b = a + c \longleftrightarrow a = (\infty::ennreal) \vee b = c$   
 ⟨*proof*⟩

**lemma** *ennreal-add-left-cancel-le*:  $a + b \leq a + c \longleftrightarrow a = (\infty::ennreal) \vee b \leq c$   
 ⟨*proof*⟩

**lemma** *ereal-add-left-cancel-less*:

**fixes** *a b c* :: *ereal*

**shows**  $0 \leq a \implies 0 \leq b \implies a + b < a + c \longleftrightarrow a \neq \infty \wedge b < c$

⟨*proof*⟩

**lemma** *ennreal-add-left-cancel-less*:  $a + b < a + c \longleftrightarrow a \neq (\infty::ennreal) \wedge b < c$   
 ⟨*proof*⟩

⟨*ML*⟩

## 40.3 Order with top

**lemma** *ennreal-zero-less-top[simp]*:  $0 < (top::ennreal)$   
 ⟨*proof*⟩

**lemma** *ennreal-one-less-top[simp]*:  $1 < (top::ennreal)$   
 ⟨*proof*⟩

**lemma** *ennreal-zero-neq-top[simp]*:  $0 \neq (top::ennreal)$   
 ⟨*proof*⟩

**lemma** *ennreal-top-neq-zero[simp]*:  $(top::ennreal) \neq 0$   
 ⟨*proof*⟩

**lemma** *ennreal-top-neq-one[simp]*:  $top \neq (1::ennreal)$   
 ⟨*proof*⟩

**lemma** *ennreal-one-neq-top[simp]*:  $1 \neq (top::ennreal)$   
 ⟨*proof*⟩

**lemma** *ennreal-add-less-top[simp]*:

**fixes** *a b* :: *ennreal*

**shows**  $a + b < top \longleftrightarrow a < top \wedge b < top$

⟨*proof*⟩

**lemma** *ennreal-add-eq-top[simp]*:

**fixes** *a b* :: *ennreal*

**shows**  $a + b = top \longleftrightarrow a = top \vee b = top$

*<proof>*

**lemma** *ennreal-sum-less-top[simp]*:

**fixes**  $f :: 'a \Rightarrow \text{ennreal}$

**shows**  $\text{finite } I \implies (\sum i \in I. f\ i) < \text{top} \longleftrightarrow (\forall i \in I. f\ i < \text{top})$

*<proof>*

**lemma** *ennreal-sum-eq-top[simp]*:

**fixes**  $f :: 'a \Rightarrow \text{ennreal}$

**shows**  $\text{finite } I \implies (\sum i \in I. f\ i) = \text{top} \longleftrightarrow (\exists i \in I. f\ i = \text{top})$

*<proof>*

**lemma** *ennreal-mult-eq-top-iff*:

**fixes**  $a\ b :: \text{ennreal}$

**shows**  $a * b = \text{top} \longleftrightarrow (a = \text{top} \wedge b \neq 0) \vee (b = \text{top} \wedge a \neq 0)$

*<proof>*

**lemma** *ennreal-top-eq-mult-iff*:

**fixes**  $a\ b :: \text{ennreal}$

**shows**  $\text{top} = a * b \longleftrightarrow (a = \text{top} \wedge b \neq 0) \vee (b = \text{top} \wedge a \neq 0)$

*<proof>*

**lemma** *ennreal-mult-less-top*:

**fixes**  $a\ b :: \text{ennreal}$

**shows**  $a * b < \text{top} \longleftrightarrow (a = 0 \vee b = 0 \vee (a < \text{top} \wedge b < \text{top}))$

*<proof>*

**lemma** *top-power-ennreal*:  $\text{top}^{\wedge} n = (\text{if } n = 0 \text{ then } 1 \text{ else } \text{top} :: \text{ennreal})$

*<proof>*

**lemma** *ennreal-prod-eq-0[simp]*:

**fixes**  $f :: 'a \Rightarrow \text{ennreal}$

**shows**  $(\text{prod } f\ A = 0) = (\text{finite } A \wedge (\exists i \in A. f\ i = 0))$

*<proof>*

**lemma** *ennreal-prod-eq-top*:

**fixes**  $f :: 'a \Rightarrow \text{ennreal}$

**shows**  $(\prod i \in I. f\ i) = \text{top} \longleftrightarrow (\text{finite } I \wedge ((\forall i \in I. f\ i \neq 0) \wedge (\exists i \in I. f\ i = \text{top})))$

*<proof>*

**lemma** *ennreal-top-mult*:  $\text{top} * a = (\text{if } a = 0 \text{ then } 0 \text{ else } \text{top} :: \text{ennreal})$

*<proof>*

**lemma** *ennreal-mult-top*:  $a * \text{top} = (\text{if } a = 0 \text{ then } 0 \text{ else } \text{top} :: \text{ennreal})$

*<proof>*

**lemma** *enn2ereal-eq-top-iff[simp]*:  $\text{enn2ereal } x = \infty \longleftrightarrow x = \text{top}$

*<proof>*

**lemma** *enn2ereal-top[simp]*:  $\text{enn2ereal } \text{top} = \infty$   
 ⟨*proof*⟩

**lemma** *e2ennreal-infty[simp]*:  $e2ennreal \ \infty = \text{top}$   
 ⟨*proof*⟩

**lemma** *ennreal-top-minus[simp]*:  $\text{top} - x = (\text{top}::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *minus-top-ennreal*:  $x - \text{top} = (\text{if } x = \text{top} \text{ then } \text{top} \text{ else } 0::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *bot-ennreal*:  $\text{bot} = (0::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *ennreal-of-nat-neq-top[simp]*:  $\text{of-nat } i \neq (\text{top}::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *numeral-eq-of-nat*:  $(\text{numeral } a::\text{ennreal}) = \text{of-nat } (\text{numeral } a)$   
 ⟨*proof*⟩

**lemma** *of-nat-less-top*:  $\text{of-nat } i < (\text{top}::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *top-neq-numeral[simp]*:  $\text{top} \neq (\text{numeral } i::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *ennreal-numeral-less-top[simp]*:  $\text{numeral } i < (\text{top}::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *ennreal-add-bot[simp]*:  $\text{bot} + x = (x::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *add-top-right-ennreal [simp]*:  $x + \text{top} = (\text{top}::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *add-top-left-ennreal [simp]*:  $\text{top} + x = (\text{top}::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *ennreal-top-mult-left [simp]*:  $x \neq 0 \implies x * \text{top} = (\text{top}::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *ennreal-top-mult-right [simp]*:  $x \neq 0 \implies \text{top} * x = (\text{top}::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *power-top-ennreal [simp]*:  $n > 0 \implies \text{top} \wedge n = (\text{top}::\text{ennreal})$   
 ⟨*proof*⟩

**lemma** *power-eq-top-ennreal-iff*:  $x \wedge n = \text{top} \iff x = (\text{top} :: \text{ennreal}) \wedge n > 0$   
 ⟨*proof*⟩

**lemma** *ennreal-mult-le-mult-iff*:  $c \neq 0 \implies c \neq \text{top} \implies c * a \leq c * b \iff a \leq$   
 ( $b :: \text{ennreal}$ )  
**including** *ennreal.lifting*  
 ⟨*proof*⟩

**lemma** *power-mono-ennreal*:  $x \leq y \implies x \wedge n \leq (y \wedge n :: \text{ennreal})$   
 ⟨*proof*⟩

**instance** *ennreal :: semiring-char-0*  
 ⟨*proof*⟩

#### 40.4 Arithmetic

**lemma** *ennreal-minus-zero[simp]*:  $a - (0 :: \text{ennreal}) = a$   
 ⟨*proof*⟩

**lemma** *ennreal-add-diff-cancel-right[simp]*:  
**fixes**  $x y z :: \text{ennreal}$  **shows**  $y \neq \text{top} \implies (x + y) - y = x$   
 ⟨*proof*⟩

**lemma** *ennreal-add-diff-cancel-left[simp]*:  
**fixes**  $x y z :: \text{ennreal}$  **shows**  $y \neq \text{top} \implies (y + x) - y = x$   
 ⟨*proof*⟩

**lemma**  
**fixes**  $a b :: \text{ennreal}$   
**shows**  $a - b = 0 \implies a \leq b$   
 ⟨*proof*⟩

**lemma** *ennreal-minus-cancel*:  
**fixes**  $a b c :: \text{ennreal}$   
**shows**  $c \neq \text{top} \implies a \leq c \implies b \leq c \implies c - a = c - b \implies a = b$   
 ⟨*proof*⟩

**lemma** *sup-const-add-ennreal*:  
**fixes**  $a b c :: \text{ennreal}$   
**shows**  $\text{sup } (c + a) (c + b) = c + \text{sup } a b$   
 ⟨*proof*⟩

**lemma** *ennreal-diff-add-assoc*:  
**fixes**  $a b c :: \text{ennreal}$   
**shows**  $a \leq b \implies c + b - a = c + (b - a)$   
 ⟨*proof*⟩

**lemma** *mult-divide-eq-ennreal*:  
**fixes**  $a b :: \text{ennreal}$



**shows**  $b \neq 0 \implies b \neq \text{top} \implies (a * b) / b = a$   
 ⟨proof⟩

**lemma** *divide-mult-eq*:  $a \neq 0 \implies a \neq \infty \implies x * a / (b * a) = x / (b::\text{ennreal})$   
 ⟨proof⟩

**lemma** *ennreal-mult-divide-eq*:  
**fixes**  $a b :: \text{ennreal}$   
**shows**  $b \neq 0 \implies b \neq \text{top} \implies (a * b) / b = a$   
 ⟨proof⟩

**lemma** *ennreal-add-diff-cancel*:  
**fixes**  $a b :: \text{ennreal}$   
**shows**  $b \neq \infty \implies (a + b) - b = a$   
 ⟨proof⟩

**lemma** *ennreal-minus-eq-0*:  
 $a - b = 0 \implies a \leq (b::\text{ennreal})$   
 ⟨proof⟩

**lemma** *ennreal-mono-minus-cancel*:  
**fixes**  $a b c :: \text{ennreal}$   
**shows**  $a - b \leq a - c \implies a < \text{top} \implies b \leq a \implies c \leq a \implies c \leq b$   
 ⟨proof⟩

**lemma** *ennreal-mono-minus*:  
**fixes**  $a b c :: \text{ennreal}$   
**shows**  $c \leq b \implies a - b \leq a - c$   
 ⟨proof⟩

**lemma** *ennreal-minus-pos-iff*:  
**fixes**  $a b :: \text{ennreal}$   
**shows**  $a < \text{top} \vee b < \text{top} \implies 0 < a - b \implies b < a$   
 ⟨proof⟩

**lemma** *ennreal-inverse-top[simp]*:  $\text{inverse top} = (0::\text{ennreal})$   
 ⟨proof⟩

**lemma** *ennreal-inverse-zero[simp]*:  $\text{inverse } 0 = (\text{top}::\text{ennreal})$   
 ⟨proof⟩

**lemma** *ennreal-top-divide*:  $\text{top} / (x::\text{ennreal}) = (\text{if } x = \text{top} \text{ then } 0 \text{ else top})$   
 ⟨proof⟩

**lemma** *ennreal-zero-divide[simp]*:  $0 / (x::\text{ennreal}) = 0$   
 ⟨proof⟩

**lemma** *ennreal-divide-zero[simp]*:  $x / (0::\text{ennreal}) = (\text{if } x = 0 \text{ then } 0 \text{ else top})$   
 ⟨proof⟩

**lemma** *ennreal-divide-top[simp]*:  $x / (\text{top}::\text{ennreal}) = 0$   
 ⟨proof⟩

**lemma** *ennreal-times-divide*:  $a * (b / c) = a * b / (c::\text{ennreal})$   
 ⟨proof⟩

**lemma** *ennreal-zero-less-divide*:  $0 < a / b \longleftrightarrow (0 < a \wedge b < (\text{top}::\text{ennreal}))$   
 ⟨proof⟩

**lemma** *add-divide-distrib-ennreal*:  $(a + b) / c = a / c + b / (c :: \text{ennreal})$   
 ⟨proof⟩

**lemma** *divide-right-mono-ennreal*:  
 fixes  $a b c :: \text{ennreal}$   
 shows  $a \leq b \implies a / c \leq b / c$   
 ⟨proof⟩

**lemma** *ennreal-mult-strict-right-mono*:  $(a::\text{ennreal}) < c \implies 0 < b \implies b < \text{top} \implies a * b < c * b$   
 ⟨proof⟩

**lemma** *ennreal-indicator-less[simp]*:  
 indicator  $A x \leq (\text{indicator } B x::\text{ennreal}) \longleftrightarrow (x \in A \longrightarrow x \in B)$   
 ⟨proof⟩

**lemma** *ennreal-inverse-positive*:  $0 < \text{inverse } x \longleftrightarrow (x::\text{ennreal}) \neq \text{top}$   
 ⟨proof⟩

**lemma** *ennreal-inverse-mult'*:  $((0 < b \vee a < \text{top}) \wedge (0 < a \vee b < \text{top})) \implies \text{inverse } (a * b::\text{ennreal}) = \text{inverse } a * \text{inverse } b$   
 ⟨proof⟩

**lemma** *ennreal-inverse-mult*:  $a < \text{top} \implies b < \text{top} \implies \text{inverse } (a * b::\text{ennreal}) = \text{inverse } a * \text{inverse } b$   
 ⟨proof⟩

**lemma** *ennreal-inverse-1[simp]*:  $\text{inverse } (1::\text{ennreal}) = 1$   
 ⟨proof⟩

**lemma** *ennreal-inverse-eq-0-iff[simp]*:  $\text{inverse } (a::\text{ennreal}) = 0 \longleftrightarrow a = \text{top}$   
 ⟨proof⟩

**lemma** *ennreal-inverse-eq-top-iff[simp]*:  $\text{inverse } (a::\text{ennreal}) = \text{top} \longleftrightarrow a = 0$   
 ⟨proof⟩

**lemma** *ennreal-divide-eq-0-iff[simp]*:  $(a::\text{ennreal}) / b = 0 \longleftrightarrow (a = 0 \vee b = \text{top})$   
 ⟨proof⟩

**lemma** *ennreal-divide-eq-top-iff*:  $(a::ennreal) / b = top \iff ((a \neq 0 \wedge b = 0) \vee (a = top \wedge b \neq top))$

*<proof>*

**lemma** *one-divide-one-divide-ennreal[simp]*:  $1 / (1 / c) = (c::ennreal)$

**including** *ennreal.lifting*

*<proof>*

**lemma** *ennreal-mult-left-cong*:

$((a::ennreal) \neq 0 \implies b = c) \implies a * b = a * c$

*<proof>*

**lemma** *ennreal-mult-right-cong*:

$((a::ennreal) \neq 0 \implies b = c) \implies b * a = c * a$

*<proof>*

**lemma** *ennreal-zero-less-mult-iff*:  $0 < a * b \iff 0 < a \wedge 0 < (b::ennreal)$

*<proof>*

**lemma** *less-diff-eq-ennreal*:

**fixes**  $a b c :: ennreal$

**shows**  $b < top \vee c < top \implies a < b - c \iff a + c < b$

*<proof>*

**lemma** *diff-add-cancel-ennreal*:

**fixes**  $a b :: ennreal$  **shows**  $a \leq b \implies b - a + a = b$

*<proof>*

**lemma** *ennreal-diff-self[simp]*:  $a \neq top \implies a - a = (0::ennreal)$

*<proof>*

**lemma** *ennreal-minus-mono*:

**fixes**  $a b c :: ennreal$

**shows**  $a \leq c \implies d \leq b \implies a - b \leq c - d$

*<proof>*

**lemma** *ennreal-minus-eq-top[simp]*:  $a - (b::ennreal) = top \iff a = top$

*<proof>*

**lemma** *ennreal-divide-self[simp]*:  $a \neq 0 \implies a < top \implies a / a = (1::ennreal)$

*<proof>*

## 40.5 Coercion from *real* to *ennreal*

**lift-definition** *ennreal* :: *real*  $\Rightarrow$  *ennreal* **is**  $sup\ 0 \circ ereal$

*<proof>*

**declare**  $[[coercion\ ennreal]]$

**lemma** *ennreal-cong*:  $x = y \implies \text{ennreal } x = \text{ennreal } y$   
 ⟨*proof*⟩

**lemma** *ennreal-cases*[*cases type: ennreal*]:  
**fixes**  $x :: \text{ennreal}$   
**obtains** (*real*)  $r :: \text{real}$  **where**  $0 \leq r$   $x = \text{ennreal } r \mid (\text{top}) x = \text{top}$   
 ⟨*proof*⟩

**lemmas** *ennreal2-cases* = *ennreal-cases*[*case-product ennreal-cases*]  
**lemmas** *ennreal3-cases* = *ennreal-cases*[*case-product ennreal2-cases*]

**lemma** *ennreal-neq-top*[*simp*]:  $\text{ennreal } r \neq \text{top}$   
 ⟨*proof*⟩

**lemma** *top-neq-ennreal*[*simp*]:  $\text{top} \neq \text{ennreal } r$   
 ⟨*proof*⟩

**lemma** *ennreal-less-top*[*simp*]:  $\text{ennreal } x < \text{top}$   
 ⟨*proof*⟩

**lemma** *ennreal-neg*:  $x \leq 0 \implies \text{ennreal } x = 0$   
 ⟨*proof*⟩

**lemma** *ennreal-inj*[*simp*]:  
 $0 \leq a \implies 0 \leq b \implies \text{ennreal } a = \text{ennreal } b \longleftrightarrow a = b$   
 ⟨*proof*⟩

**lemma** *ennreal-le-iff*[*simp*]:  $0 \leq y \implies \text{ennreal } x \leq \text{ennreal } y \longleftrightarrow x \leq y$   
 ⟨*proof*⟩

**lemma** *le-ennreal-iff*:  $0 \leq r \implies x \leq \text{ennreal } r \longleftrightarrow (\exists q \geq 0. x = \text{ennreal } q \wedge q \leq r)$   
 ⟨*proof*⟩

**lemma** *ennreal-less-iff*:  $0 \leq r \implies \text{ennreal } r < \text{ennreal } q \longleftrightarrow r < q$   
 ⟨*proof*⟩

**lemma** *ennreal-eq-zero-iff*[*simp*]:  $0 \leq x \implies \text{ennreal } x = 0 \longleftrightarrow x = 0$   
 ⟨*proof*⟩

**lemma** *ennreal-less-zero-iff*[*simp*]:  $0 < \text{ennreal } x \longleftrightarrow 0 < x$   
 ⟨*proof*⟩

**lemma** *ennreal-lessI*:  $0 < q \implies r < q \implies \text{ennreal } r < \text{ennreal } q$   
 ⟨*proof*⟩

**lemma** *ennreal-leI*:  $x \leq y \implies \text{ennreal } x \leq \text{ennreal } y$   
 ⟨*proof*⟩

**lemma** *enn2ereal-ennreal[simp]*:  $0 \leq x \implies \text{enn2ereal} (\text{ennreal } x) = x$   
 ⟨proof⟩

**lemma** *e2ennreal-enn2ereal[simp]*:  $e2ennreal (\text{enn2ereal } x) = x$   
 ⟨proof⟩

**lemma** *enn2ereal-e2ennreal*:  $x \geq 0 \implies \text{enn2ereal} (e2ennreal x) = x$   
 ⟨proof⟩

**lemma** *e2ennreal-ereal [simp]*:  $e2ennreal (\text{ereal } x) = \text{ennreal } x$   
 ⟨proof⟩

**lemma** *ennreal-0[simp]*:  $\text{ennreal } 0 = 0$   
 ⟨proof⟩

**lemma** *ennreal-1[simp]*:  $\text{ennreal } 1 = 1$   
 ⟨proof⟩

**lemma** *ennreal-eq-0-iff*:  $\text{ennreal } x = 0 \longleftrightarrow x \leq 0$   
 ⟨proof⟩

**lemma** *ennreal-le-iff2*:  $\text{ennreal } x \leq \text{ennreal } y \longleftrightarrow ((0 \leq y \wedge x \leq y) \vee (x \leq 0 \wedge y \leq 0))$   
 ⟨proof⟩

**lemma** *ennreal-eq-1[simp]*:  $\text{ennreal } x = 1 \longleftrightarrow x = 1$   
 ⟨proof⟩

**lemma** *ennreal-le-1[simp]*:  $\text{ennreal } x \leq 1 \longleftrightarrow x \leq 1$   
 ⟨proof⟩

**lemma** *ennreal-ge-1[simp]*:  $\text{ennreal } x \geq 1 \longleftrightarrow x \geq 1$   
 ⟨proof⟩

**lemma** *one-less-ennreal[simp]*:  $1 < \text{ennreal } x \longleftrightarrow 1 < x$   
 ⟨proof⟩

**lemma** *ennreal-plus[simp]*:  
 $0 \leq a \implies 0 \leq b \implies \text{ennreal} (a + b) = \text{ennreal } a + \text{ennreal } b$   
 ⟨proof⟩

**lemma** *add-mono-ennreal*:  $x < \text{ennreal } y \implies x' < \text{ennreal } y' \implies x + x' < \text{ennreal} (y + y')$   
 ⟨proof⟩

**lemma** *sum-ennreal[simp]*:  $(\bigwedge i. i \in I \implies 0 \leq f i) \implies (\sum i \in I. \text{ennreal} (f i)) = \text{ennreal} (\text{sum } f I)$   
 ⟨proof⟩

**lemma** *sum-list-ennreal*[simp]:

**assumes**  $\bigwedge x. x \in \text{set } xs \implies f x \geq 0$

**shows**  $\text{sum-list } (\text{map } (\lambda x. \text{ennreal } (f x)) \text{ } xs) = \text{ennreal } (\text{sum-list } (\text{map } f \text{ } xs))$

*<proof>*

**lemma** *ennreal-of-nat-eq-real-of-nat*:  $\text{of-nat } i = \text{ennreal } (\text{of-nat } i)$

*<proof>*

**lemma** *of-nat-le-ennreal-iff*[simp]:  $0 \leq r \implies \text{of-nat } i \leq \text{ennreal } r \iff \text{of-nat } i \leq r$

*<proof>*

**lemma** *ennreal-le-of-nat-iff*[simp]:  $\text{ennreal } r \leq \text{of-nat } i \iff r \leq \text{of-nat } i$

*<proof>*

**lemma** *ennreal-indicator*:  $\text{ennreal } (\text{indicator } A \text{ } x) = \text{indicator } A \text{ } x$

*<proof>*

**lemma** *ennreal-numeral*[simp]:  $\text{ennreal } (\text{numeral } n) = \text{numeral } n$

*<proof>*

**lemma** *ennreal-less-numeral-iff* [simp]:  $\text{ennreal } n < \text{numeral } w \iff n < \text{numeral } w$

*<proof>*

**lemma** *numeral-less-ennreal-iff* [simp]:  $\text{numeral } w < \text{ennreal } n \iff \text{numeral } w < n$

*<proof>*

**lemma** *numeral-le-ennreal-iff* [simp]:  $\text{numeral } n \leq \text{ennreal } m \iff \text{numeral } n \leq m$

*<proof>*

**lemma** *min-ennreal*:  $0 \leq x \implies 0 \leq y \implies \text{min } (\text{ennreal } x) (\text{ennreal } y) = \text{ennreal } (\text{min } x \text{ } y)$

*<proof>*

**lemma** *ennreal-half*[simp]:  $\text{ennreal } (1/2) = \text{inverse } 2$

*<proof>*

**lemma** *ennreal-minus*:  $0 \leq q \implies \text{ennreal } r - \text{ennreal } q = \text{ennreal } (r - q)$

*<proof>*

**lemma** *ennreal-minus-top*[simp]:  $\text{ennreal } a - \text{top} = 0$

*<proof>*

**lemma** *e2ennreal-enn2ereal-diff* [simp]:

$e2ennreal(\text{enn2ereal } x - \text{enn2ereal } y) = x - y$  **for**  $x \text{ } y$

*<proof>*

**lemma** *ennreal-mult*:  $0 \leq a \implies 0 \leq b \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$

*<proof>*

**lemma** *ennreal-mult'*:  $0 \leq a \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$

*<proof>*

**lemma** *indicator-mult-ennreal*:  $\text{indicator } A \ x * \text{ennreal } r = \text{ennreal } (\text{indicator } A \ x * r)$

*<proof>*

**lemma** *ennreal-mult''*:  $0 \leq b \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$

*<proof>*

**lemma** *numeral-mult-ennreal*:  $0 \leq x \implies \text{numeral } b * \text{ennreal } x = \text{ennreal } (\text{numeral } b * x)$

*<proof>*

**lemma** *ennreal-power*:  $0 \leq r \implies \text{ennreal } r \wedge n = \text{ennreal } (r \wedge n)$

*<proof>*

**lemma** *power-eq-top-ennreal*:  $x \wedge n = \text{top} \iff (n \neq 0 \wedge (x :: \text{ennreal}) = \text{top})$

*<proof>*

**lemma** *inverse-ennreal*:  $0 < r \implies \text{inverse } (\text{ennreal } r) = \text{ennreal } (\text{inverse } r)$

*<proof>*

**lemma** *divide-ennreal*:  $0 \leq r \implies 0 < q \implies \text{ennreal } r / \text{ennreal } q = \text{ennreal } (r / q)$

*<proof>*

**lemma** *ennreal-inverse-power*:  $\text{inverse } (x \wedge n :: \text{ennreal}) = \text{inverse } x \wedge n$

*<proof>*

**lemma** *power-divide-distrib-ennreal* [*algebra-simps*]:

$(x / y) \wedge n = x \wedge n / (y \wedge n :: \text{ennreal})$

*<proof>*

**lemma** *ennreal-divide-numeral*:  $0 \leq x \implies \text{ennreal } x / \text{numeral } b = \text{ennreal } (x / \text{numeral } b)$

*<proof>*

**lemma** *prod-ennreal*:  $(\bigwedge i. i \in A \implies 0 \leq f i) \implies (\prod i \in A. \text{ennreal } (f i)) = \text{ennreal } (\text{prod } f A)$

*<proof>*

**lemma** *prod-mono-ennreal*:

**assumes**  $\bigwedge x. x \in A \implies f x \leq (g x :: \text{ennreal})$

**shows**  $\text{prod } f A \leq \text{prod } g A$

*<proof>*

**lemma** *mult-right-ennreal-cancel*:  $a * \text{ennreal } c = b * \text{ennreal } c \iff (a = b \vee c \leq 0)$   
*<proof>*

**lemma** *ennreal-le-epsilon*:  
 $(\bigwedge e::\text{real}. y < \text{top} \implies 0 < e \implies x \leq y + \text{ennreal } e) \implies x \leq y$   
*<proof>*

**lemma** *ennreal-rat-dense*:  
**fixes**  $x y :: \text{ennreal}$   
**shows**  $x < y \implies \exists r::\text{rat}. x < \text{real-of-rat } r \wedge \text{real-of-rat } r < y$   
*<proof>*

**lemma** *ennreal-Ex-less-of-nat*:  $(x::\text{ennreal}) < \text{top} \implies \exists n. x < \text{of-nat } n$   
*<proof>*

## 40.6 Coercion from *ennreal* to *real*

**definition** *enn2real*  $x = \text{real-of-ereal } (\text{enn2ereal } x)$

**lemma** *enn2real-nonneg[simp]*:  $0 \leq \text{enn2real } x$   
*<proof>*

**lemma** *enn2real-mono*:  $a \leq b \implies b < \text{top} \implies \text{enn2real } a \leq \text{enn2real } b$   
*<proof>*

**lemma** *enn2real-of-nat[simp]*:  $\text{enn2real } (\text{of-nat } n) = n$   
*<proof>*

**lemma** *enn2real-ennreal[simp]*:  $0 \leq r \implies \text{enn2real } (\text{ennreal } r) = r$   
*<proof>*

**lemma** *ennreal-enn2real[simp]*:  $r < \text{top} \implies \text{ennreal } (\text{enn2real } r) = r$   
*<proof>*

**lemma** *real-of-ereal-enn2ereal[simp]*:  $\text{real-of-ereal } (\text{enn2ereal } x) = \text{enn2real } x$   
*<proof>*

**lemma** *enn2real-top[simp]*:  $\text{enn2real } \text{top} = 0$   
*<proof>*

**lemma** *enn2real-0[simp]*:  $\text{enn2real } 0 = 0$   
*<proof>*

**lemma** *enn2real-1[simp]*:  $\text{enn2real } 1 = 1$   
*<proof>*



**lemma** *enn2real-numeral[simp]*:  $\text{enn2real } (\text{numeral } n) = (\text{numeral } n)$   
 ⟨*proof*⟩

**lemma** *enn2real-mult*:  $\text{enn2real } (a * b) = \text{enn2real } a * \text{enn2real } b$   
 ⟨*proof*⟩

**lemma** *enn2real-leI*:  $0 \leq B \implies x \leq \text{ennreal } B \implies \text{enn2real } x \leq B$   
 ⟨*proof*⟩

**lemma** *enn2real-positive-iff*:  $0 < \text{enn2real } x \longleftrightarrow (0 < x \wedge x < \text{top})$   
 ⟨*proof*⟩

**lemma** *enn2real-eq-posreal-iff[simp]*:  $c > 0 \implies \text{enn2real } x = c \longleftrightarrow x = c$   
 ⟨*proof*⟩

**lemma** *ennreal-enn2real-if*:  $\text{ennreal } (\text{enn2real } r) = (\text{if } r = \text{top} \text{ then } 0 \text{ else } r)$   
 ⟨*proof*⟩

## 40.7 Coercion from *enat* to *ennreal*

**definition** *ennreal-of-enat* ::  $\text{enat} \Rightarrow \text{ennreal}$

**where**

*ennreal-of-enat*  $n = (\text{case } n \text{ of } \infty \Rightarrow \text{top} \mid \text{enat } n \Rightarrow \text{of-nat } n)$

**declare** [[*coercion* *ennreal-of-enat*]]

**declare** [[*coercion* *of-nat* ::  $\text{nat} \Rightarrow \text{ennreal}$ ]]

**lemma** *ennreal-of-enat-infty[simp]*:  $\text{ennreal-of-enat } \infty = \infty$   
 ⟨*proof*⟩

**lemma** *ennreal-of-enat-enat[simp]*:  $\text{ennreal-of-enat } (\text{enat } n) = \text{of-nat } n$   
 ⟨*proof*⟩

**lemma** *ennreal-of-enat-0[simp]*:  $\text{ennreal-of-enat } 0 = 0$   
 ⟨*proof*⟩

**lemma** *ennreal-of-enat-1[simp]*:  $\text{ennreal-of-enat } 1 = 1$   
 ⟨*proof*⟩

**lemma** *ennreal-top-neq-of-nat[simp]*:  $(\text{top}::\text{ennreal}) \neq \text{of-nat } i$   
 ⟨*proof*⟩

**lemma** *ennreal-of-enat-inj[simp]*:  $\text{ennreal-of-enat } i = \text{ennreal-of-enat } j \longleftrightarrow i = j$   
 ⟨*proof*⟩

**lemma** *ennreal-of-enat-le-iff[simp]*:  $\text{ennreal-of-enat } m \leq \text{ennreal-of-enat } n \longleftrightarrow m \leq n$   
 ⟨*proof*⟩

**lemma** *of-nat-less-ennreal-of-nat[simp]*:  $\text{of-nat } n \leq \text{ennreal-of-enat } x \longleftrightarrow \text{of-nat } n \leq x$   
 ⟨proof⟩

**lemma** *ennreal-of-enat-Sup*:  $\text{ennreal-of-enat } (\text{Sup } X) = (\text{SUP } x \in X. \text{ennreal-of-enat } x)$   
 ⟨proof⟩

**lemma** *ennreal-of-enat-eSuc[simp]*:  $\text{ennreal-of-enat } (\text{eSuc } x) = 1 + \text{ennreal-of-enat } x$   
 ⟨proof⟩

**lemma** *ennreal-of-enat-plus[simp]*:  $\langle \text{ennreal-of-enat } (a+b) = \text{ennreal-of-enat } a + \text{ennreal-of-enat } b \rangle$   
 ⟨proof⟩

**lemma** *sum-ennreal-of-enat[simp]*:  $(\sum i \in I. \text{ennreal-of-enat } (f i)) = \text{ennreal-of-enat } (\text{sum } f I)$   
 ⟨proof⟩

## 40.8 Topology on *ennreal*

**lemma** *enn2ereal-Iio*:  $\text{enn2ereal } -' \{..<a\} = (\text{if } 0 \leq a \text{ then } \{..< \text{e2ennreal } a\} \text{ else } \{\})$   
 ⟨proof⟩

**lemma** *enn2ereal-Ioi*:  $\text{enn2ereal } -' \{a <..\} = (\text{if } 0 \leq a \text{ then } \{\text{e2ennreal } a <..\} \text{ else } \text{UNIV})$   
 ⟨proof⟩

**instantiation** *ennreal* :: *linear-continuum-topology*  
**begin**

**definition** *open-ennreal* :: *ennreal set*  $\Rightarrow$  *bool*  
**where** (*open* :: *ennreal set*  $\Rightarrow$  *bool*) = *generate-topology* (*range lessThan*  $\cup$  *range greaterThan*)

**instance**  
 ⟨proof⟩

**end**

**lemma** *continuous-on-e2ennreal*: *continuous-on* *A* *e2ennreal*  
 ⟨proof⟩

**lemma** *continuous-at-e2ennreal*: *continuous* (*at* *x* *within* *A*) *e2ennreal*  
 ⟨proof⟩

**lemma** *continuous-on-enn2ereal*: *continuous-on UNIV enn2ereal*  
 ⟨proof⟩

**lemma** *continuous-at-enn2ereal*: *continuous (at x within A) enn2ereal*  
 ⟨proof⟩

**lemma** *sup-continuous-e2ennreal*[*order-continuous-intros*]:  
**assumes** *f*: *sup-continuous f* **shows** *sup-continuous* ( $\lambda x. e2ennreal (f x)$ )  
 ⟨proof⟩

**lemma** *sup-continuous-enn2ereal*[*order-continuous-intros*]:  
**assumes** *f*: *sup-continuous f* **shows** *sup-continuous* ( $\lambda x. enn2ereal (f x)$ )  
 ⟨proof⟩

**lemma** *sup-continuous-mult-left-ennreal'*:  
**fixes** *c* :: *ennreal*  
**shows** *sup-continuous* ( $\lambda x. c * x$ )  
 ⟨proof⟩

**lemma** *sup-continuous-mult-left-ennreal*[*order-continuous-intros*]:  
*sup-continuous f*  $\implies$  *sup-continuous* ( $\lambda x. c * f x$  :: *ennreal*)  
 ⟨proof⟩

**lemma** *sup-continuous-mult-right-ennreal*[*order-continuous-intros*]:  
*sup-continuous f*  $\implies$  *sup-continuous* ( $\lambda x. f x * c$  :: *ennreal*)  
 ⟨proof⟩

**lemma** *sup-continuous-divide-ennreal*[*order-continuous-intros*]:  
**fixes** *f g* :: '*a*::*complete-lattice*  $\Rightarrow$  *ennreal*  
**shows** *sup-continuous f*  $\implies$  *sup-continuous* ( $\lambda x. f x / c$ )  
 ⟨proof⟩

**lemma** *transfer-enn2ereal-continuous-on* [*transfer-rule*]:  
*rel-fun* (=) (*rel-fun* (*rel-fun* (=) *pcr-ennreal*) (=)) *continuous-on continuous-on*  
 ⟨proof⟩

**lemma** *transfer-sup-continuous*[*transfer-rule*]:  
(*rel-fun* (*rel-fun* (=) *pcr-ennreal*) (=)) *sup-continuous sup-continuous*  
 ⟨proof⟩

**lemma** *continuous-on-ennreal*[*tendsto-intros*]:  
*continuous-on A f*  $\implies$  *continuous-on A* ( $\lambda x. ennreal (f x)$ )  
 ⟨proof⟩

**lemma** *tendsto-ennrealD*:  
**assumes** *lim*: ( $\lambda x. ennreal (f x)$ )  $\longrightarrow$  *ennreal x* *F*  
**assumes** \*:  $\forall_F x$  in *F*.  $0 \leq f x$  **and**  $x: 0 \leq x$   
**shows** ( $f \longrightarrow x$ ) *F*

*<proof>*

**lemma** *tendsto-ennreal-iff [simp]*:

$\langle (\lambda x. \text{ennreal } (f x)) \longrightarrow \text{ennreal } x \rangle F \longleftrightarrow \langle f \longrightarrow x \rangle F$  **(is**  $\langle ?P \longleftrightarrow ?Q \rangle$   
**if**  $\langle \forall_F x \text{ in } F. 0 \leq f x \rangle \langle 0 \leq x \rangle$

*<proof>*

**lemma** *tendsto-enn2ereal-iff [simp]*:  $\langle (\lambda i. \text{enn2ereal } (f i)) \longrightarrow \text{enn2ereal } x \rangle F \longleftrightarrow$   
 $\langle f \longrightarrow x \rangle F$

*<proof>*

**lemma** *ennreal-tendsto-0-iff*:  $(\bigwedge n. f n \geq 0) \implies \langle (\lambda n. \text{ennreal } (f n)) \longrightarrow 0 \rangle$   
 $\longleftrightarrow \langle f \longrightarrow 0 \rangle$

*<proof>*

**lemma** *continuous-on-add-ennreal*:

**fixes**  $f g :: 'a::\text{topological-space} \Rightarrow \text{ennreal}$

**shows** *continuous-on*  $A f \implies \text{continuous-on } A g \implies \text{continuous-on } A (\lambda x. f x$   
 $+ g x)$

*<proof>*

**lemma** *continuous-on-inverse-ennreal [continuous-intros]*:

**fixes**  $f :: 'a::\text{topological-space} \Rightarrow \text{ennreal}$

**shows** *continuous-on*  $A f \implies \text{continuous-on } A (\lambda x. \text{inverse } (f x))$

*<proof>*

**instance** *ennreal* :: *topological-comm-monoid-add*

*<proof>*

**lemma** *sup-continuous-add-ennreal [order-continuous-intros]*:

**fixes**  $f g :: 'a::\text{complete-lattice} \Rightarrow \text{ennreal}$

**shows** *sup-continuous*  $f \implies \text{sup-continuous } g \implies \text{sup-continuous } (\lambda x. f x + g$   
 $x)$

*<proof>*

**lemma** *ennreal-suminf-lessD*:  $(\sum i. f i :: \text{ennreal}) < x \implies f i < x$

*<proof>*

**lemma** *sums-ennreal [simp]*:  $(\bigwedge i. 0 \leq f i) \implies 0 \leq x \implies (\lambda i. \text{ennreal } (f i)) \text{ sums}$   
 $\text{ennreal } x \longleftrightarrow f \text{ sums } x$

*<proof>*

**lemma** *summable-suminf-not-top*:  $(\bigwedge i. 0 \leq f i) \implies (\sum i. \text{ennreal } (f i)) \neq \text{top} \implies$   
 $\text{summable } f$

*<proof>*

**lemma** *suminf-ennreal [simp]*:

$(\bigwedge i. 0 \leq f i) \implies (\sum i. \text{ennreal } (f i)) \neq \text{top} \implies (\sum i. \text{ennreal } (f i)) = \text{ennreal}$   
 $(\sum i. f i)$

*<proof>*

**lemma** *sums-enn2ereal[simp]*:  $(\lambda i. \text{enn2ereal } (f i)) \text{ sums } \text{enn2ereal } x \longleftrightarrow f \text{ sums } x$   
*<proof>*

**lemma** *suminf-enn2ereal[simp]*:  $(\sum i. \text{enn2ereal } (f i)) = \text{enn2ereal } (\text{suminf } f)$   
*<proof>*

**lemma** *transfer-e2ennreal-suminf [transfer-rule]*: *rel-fun* (*rel-fun* (=) *pcr-ennreal*)  
*pcr-ennreal suminf suminf*  
*<proof>*

**lemma** *ennreal-suminf-cmult[simp]*:  $(\sum i. r * f i) = r * (\sum i. f i::\text{ennreal})$   
*<proof>*

**lemma** *ennreal-suminf-multc[simp]*:  $(\sum i. f i * r) = (\sum i. f i::\text{ennreal}) * r$   
*<proof>*

**lemma** *ennreal-suminf-divide[simp]*:  $(\sum i. f i / r) = (\sum i. f i::\text{ennreal}) / r$   
*<proof>*

**lemma** *ennreal-suminf-neq-top*: *summable*  $f \implies (\bigwedge i. 0 \leq f i) \implies (\sum i. \text{ennreal } (f i)) \neq \text{top}$   
*<proof>*

**lemma** *suminf-ennreal-eq*:  
 $(\bigwedge i. 0 \leq f i) \implies f \text{ sums } x \implies (\sum i. \text{ennreal } (f i)) = \text{ennreal } x$   
*<proof>*

**lemma** *ennreal-suminf-bound-add*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{ennreal}$   
**shows**  $(\bigwedge N. (\sum n < N. f n) + y \leq x) \implies \text{suminf } f + y \leq x$   
*<proof>*

**lemma** *ennreal-suminf-SUP-eq-directed*:  
**fixes**  $f :: 'a \Rightarrow \text{nat} \Rightarrow \text{ennreal}$   
**assumes**  $*$ :  $\bigwedge N i j. i \in I \implies j \in I \implies \text{finite } N \implies \exists k \in I. \forall n \in N. f i n \leq f k n \wedge f j n \leq f k n$   
**shows**  $(\sum n. \text{SUP } i \in I. f i n) = (\text{SUP } i \in I. \sum n. f i n)$   
*<proof>*

**lemma** *INF-ennreal-add-const*:  
**fixes**  $f g :: \text{nat} \Rightarrow \text{ennreal}$   
**shows**  $(\text{INF } i. f i + c) = (\text{INF } i. f i) + c$   
*<proof>*

**lemma** *INF-ennreal-const-add*:  
**fixes**  $f g :: \text{nat} \Rightarrow \text{ennreal}$

**shows**  $(\text{INF } i. c + f i) = c + (\text{INF } i. f i)$   
 ⟨proof⟩

**lemma** *SUP-mult-left-ennreal*:  $c * (\text{SUP } i \in I. f i) = (\text{SUP } i \in I. c * f i :: \text{ennreal})$   
 ⟨proof⟩

**lemma** *SUP-mult-right-ennreal*:  $(\text{SUP } i \in I. f i) * c = (\text{SUP } i \in I. f i * c :: \text{ennreal})$   
 ⟨proof⟩

**lemma** *SUP-divide-ennreal*:  $(\text{SUP } i \in I. f i) / c = (\text{SUP } i \in I. f i / c :: \text{ennreal})$   
 ⟨proof⟩

**lemma** *ennreal-SUP-of-nat-eq-top*:  $(\text{SUP } x. \text{of-nat } x :: \text{ennreal}) = \text{top}$   
 ⟨proof⟩

**lemma** *ennreal-SUP-eq-top*:  
**fixes**  $f :: 'a \Rightarrow \text{ennreal}$   
**assumes**  $\bigwedge n. \exists i \in I. \text{of-nat } n \leq f i$   
**shows**  $(\text{SUP } i \in I. f i) = \text{top}$   
 ⟨proof⟩

**lemma** *ennreal-INF-const-minus*:  
**fixes**  $f :: 'a \Rightarrow \text{ennreal}$   
**shows**  $I \neq \{\} \implies (\text{SUP } x \in I. c - f x) = c - (\text{INF } x \in I. f x)$   
 ⟨proof⟩

**lemma** *of-nat-Sup-ennreal*:  
**assumes**  $A \neq \{\}$  *bdd-above*  $A$   
**shows**  $\text{of-nat } (\text{Sup } A) = (\text{SUP } a \in A. \text{of-nat } a :: \text{ennreal})$   
 ⟨proof⟩

**lemma** *ennreal-tendsto-const-minus*:  
**fixes**  $g :: 'a \Rightarrow \text{ennreal}$   
**assumes**  $ae: \forall_F x \text{ in } F. g x \leq c$   
**assumes**  $g: ((\lambda x. c - g x) \longrightarrow 0) F$   
**shows**  $(g \longrightarrow c) F$   
 ⟨proof⟩

**lemma** *ennreal-SUP-add*:  
**fixes**  $f g :: \text{nat} \Rightarrow \text{ennreal}$   
**shows**  $\text{incseq } f \implies \text{incseq } g \implies (\text{SUP } i. f i + g i) = \text{Sup } (f \text{ ‘ UNIV}) + \text{Sup } (g \text{ ‘ UNIV})$   
 ⟨proof⟩

**lemma** *ennreal-SUP-sum*:  
**fixes**  $f :: 'a \Rightarrow \text{nat} \Rightarrow \text{ennreal}$   
**shows**  $(\bigwedge i. i \in I \implies \text{incseq } (f i)) \implies (\text{SUP } n. \sum i \in I. f i n) = (\sum i \in I. \text{SUP } n. f i n)$   
 ⟨proof⟩

**lemma** *ennreal-liminf-minus*:

**fixes**  $f :: \text{nat} \Rightarrow \text{ennreal}$

**shows**  $(\bigwedge n. f\ n \leq c) \implies \text{liminf } (\lambda n. c - f\ n) = c - \text{limsup } f$

*<proof>*

**lemma** *ennreal-continuous-on-cmult*:

$(c :: \text{ennreal}) < \text{top} \implies \text{continuous-on } A\ f \implies \text{continuous-on } A\ (\lambda x. c * f\ x)$

*<proof>*

**lemma** *ennreal-tendsto-cmult*:

$(c :: \text{ennreal}) < \text{top} \implies (f \longrightarrow x)\ F \implies ((\lambda x. c * f\ x) \longrightarrow c * x)\ F$

*<proof>*

**lemma** *tendsto-ennrealI*[*intro, simp, tendsto-intros*]:

$(f \longrightarrow x)\ F \implies ((\lambda x. \text{ennreal } (f\ x)) \longrightarrow \text{ennreal } x)\ F$

*<proof>*

**lemma** *tendsto-enn2erealI* [*tendsto-intros*]:

**assumes**  $(f \longrightarrow l)\ F$

**shows**  $((\lambda i. \text{enn2ereal}(f\ i)) \longrightarrow \text{enn2ereal } l)\ F$

*<proof>*

**lemma** *tendsto-e2ennrealI* [*tendsto-intros*]:

**assumes**  $(f \longrightarrow l)\ F$

**shows**  $((\lambda i. \text{e2ennreal}(f\ i)) \longrightarrow \text{e2ennreal } l)\ F$

*<proof>*

**lemma** *ennreal-suminf-minus*:

**fixes**  $f\ g :: \text{nat} \Rightarrow \text{ennreal}$

**shows**  $(\bigwedge i. g\ i \leq f\ i) \implies \text{suminf } f \neq \text{top} \implies \text{suminf } g \neq \text{top} \implies (\sum i. f\ i - g\ i) = \text{suminf } f - \text{suminf } g$

*<proof>*

**lemma** *ennreal-Sup-countable-SUP*:

$A \neq \{\} \implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{incseq } f \wedge \text{range } f \subseteq A \wedge \text{Sup } A = (\text{SUP } i. f\ i)$

*<proof>*

**lemma** *ennreal-Inf-countable-INF*:

$A \neq \{\} \implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{decseq } f \wedge \text{range } f \subseteq A \wedge \text{Inf } A = (\text{INF } i. f\ i)$

*<proof>*

**lemma** *ennreal-SUP-countable-SUP*:

$A \neq \{\} \implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{range } f \subseteq g\ A \wedge \text{Sup } (g\ A) = \text{Sup } (f\ UNIV)$

*<proof>*

**lemma** *of-nat-tendsto-top-ennreal*:  $(\lambda n :: \text{nat}. \text{of-nat } n :: \text{ennreal}) \longrightarrow \text{top}$

*<proof>*

**lemma** *SUP-sup-continuous-ennreal*:

**fixes**  $f :: \text{ennreal} \Rightarrow 'a :: \text{complete-lattice}$

**assumes**  $f: \text{sup-continuous } f$  **and**  $I \neq \{\}$

**shows**  $(\text{SUP } i \in I. f (g i)) = f (\text{SUP } i \in I. g i)$

*<proof>*

**lemma** *ennreal-suminf-SUP-eq*:

**fixes**  $f :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ennreal}$

**shows**  $(\bigwedge i. \text{incseq } (\lambda n. f n i)) \implies (\sum i. \text{SUP } n. f n i) = (\text{SUP } n. \sum i. f n i)$

*<proof>*

**lemma** *ennreal-SUP-add-left*:

**fixes**  $c :: \text{ennreal}$

**shows**  $I \neq \{\} \implies (\text{SUP } i \in I. f i + c) = (\text{SUP } i \in I. f i) + c$

*<proof>*

**lemma** *ennreal-SUP-const-minus*:

**fixes**  $f :: 'a \Rightarrow \text{ennreal}$

**shows**  $I \neq \{\} \implies c < \text{top} \implies (\text{INF } x \in I. c - f x) = c - (\text{SUP } x \in I. f x)$

*<proof>*

**lemma** *isCont-ennreal[simp]*: *<isCont ennreal x>*

*<proof>*

**lemma** *isCont-ennreal-of-enat[simp]*: *<isCont ennreal-of-enat x>*

*<proof>*

## 40.9 Approximation lemmas

**lemma** *INF-approx-ennreal*:

**fixes**  $x :: \text{ennreal}$  **and**  $e :: \text{real}$

**assumes**  $e > 0$

**assumes**  $x = (\text{INF } i \in A. f i)$

**assumes**  $x \neq \infty$

**shows**  $\exists i \in A. f i < x + e$

*<proof>*

**lemma** *SUP-approx-ennreal*:

**fixes**  $x :: \text{ennreal}$  **and**  $e :: \text{real}$

**assumes**  $e > 0$   $A \neq \{\}$

**assumes**  $\text{SUP}: x = (\text{SUP } i \in A. f i)$

**assumes**  $x \neq \infty$

**shows**  $\exists i \in A. x < f i + e$

*<proof>*

**lemma** *ennreal-approx-SUP*:

**fixes**  $x :: \text{ennreal}$



**assumes** *f-bound*:  $\bigwedge i. i \in A \implies f i \leq x$   
**assumes** *approx*:  $\bigwedge e. (e::real) > 0 \implies \exists i \in A. x \leq f i + e$   
**shows**  $x = (\text{SUP } i \in A. f i)$   
*<proof>*

**lemma** *ennreal-approx-INF*:  
**fixes**  $x::ennreal$   
**assumes** *f-bound*:  $\bigwedge i. i \in A \implies x \leq f i$   
**assumes** *approx*:  $\bigwedge e. (e::real) > 0 \implies \exists i \in A. f i \leq x + e$   
**shows**  $x = (\text{INF } i \in A. f i)$   
*<proof>*

**lemma** *ennreal-approx-unit*:  
 $(\bigwedge a::ennreal. 0 < a \implies a < 1 \implies a * z \leq y) \implies z \leq y$   
*<proof>*

**lemma** *suminf-ennreal2*:  
 $(\bigwedge i. 0 \leq f i) \implies \text{summable } f \implies (\sum i. \text{ennreal } (f i)) = \text{ennreal } (\sum i. f i)$   
*<proof>*

**lemma** *less-top-ennreal*:  $x < \text{top} \longleftrightarrow (\exists r \geq 0. x = \text{ennreal } r)$   
*<proof>*

**lemma** *enn2real-less-iff[simp]*:  $x < \text{top} \implies \text{enn2real } x < c \longleftrightarrow x < c$   
*<proof>*

**lemma** *enn2real-le-iff[simp]*:  $\llbracket x < \text{top}; c > 0 \rrbracket \implies \text{enn2real } x \leq c \longleftrightarrow x \leq c$   
*<proof>*

**lemma** *enn2real-less*:  
**assumes**  $\text{enn2real } e < r \text{ } e \neq \text{top}$  **shows**  $e < \text{ennreal } r$   
*<proof>*

**lemma** *enn2real-le*:  
**assumes**  $\text{enn2real } e \leq r \text{ } e \neq \text{top}$  **shows**  $e \leq \text{ennreal } r$   
*<proof>*

**lemma** *tendsto-top-iff-ennreal*:  
**fixes**  $f :: 'a \Rightarrow \text{ennreal}$   
**shows**  $(f \longrightarrow \text{top}) F \longleftrightarrow (\forall l \geq 0. \text{eventually } (\lambda x. \text{ennreal } l < f x) F)$   
*<proof>*

**lemma** *ennreal-tendsto-top-eq-at-top*:  
 $((\lambda z. \text{ennreal } (f z)) \longrightarrow \text{top}) F \longleftrightarrow (\text{LIM } z F. f z :> \text{at-top})$   
*<proof>*

**lemma** *tendsto-0-if-Limsup-eq-0-ennreal*:  
**fixes**  $f :: - \Rightarrow \text{ennreal}$   
**shows**  $\text{Limsup } F f = 0 \implies (f \longrightarrow 0) F$

*<proof>*

**lemma** *diff-le-self-ennreal[simp]*:  $a - b \leq (a::ennreal)$   
*<proof>*

**lemma** *ennreal-ineq-diff-add*:  $b \leq a \implies a = b + (a - b::ennreal)$   
*<proof>*

**lemma** *ennreal-mult-strict-left-mono*:  $(a::ennreal) < c \implies 0 < b \implies b < top \implies b * a < b * c$   
*<proof>*

**lemma** *ennreal-between*:  $0 < e \implies 0 < x \implies x < top \implies x - e < (x::ennreal)$   
*<proof>*

**lemma** *minus-less-iff-ennreal*:  $b < top \implies b \leq a \implies a - b < c \iff a < c + (b::ennreal)$   
*<proof>*

**lemma** *tendsto-zero-ennreal*:  
**assumes** *ev*:  $\bigwedge r. 0 < r \implies \forall_F x \text{ in } F. f x < ennreal r$   
**shows**  $(f \longrightarrow 0) F$   
*<proof>*

**lifting-update** *ennreal.lifting*  
**lifting-forget** *ennreal.lifting*

#### 40.10 *ennreal* theorems

**lemma** *neg-top-trans*: **fixes**  $x y :: ennreal$  **shows**  $\llbracket y \neq top; x \leq y \rrbracket \implies x \neq top$   
*<proof>*

**lemma** *diff-diff-ennreal*: **fixes**  $a b :: ennreal$  **shows**  $a \leq b \implies b \neq \infty \implies b - (b - a) = a$   
*<proof>*

**lemma** *ennreal-less-one-iff[simp]*:  $ennreal x < 1 \iff x < 1$   
*<proof>*

**lemma** *SUP-const-minus-ennreal*:  
**fixes**  $f :: 'a \Rightarrow ennreal$  **shows**  $I \neq \{\} \implies (SUP x \in I. c - f x) = c - (INF x \in I. f x)$   
**including** *ennreal.lifting*  
*<proof>*

**lemma** *zero-minus-ennreal[simp]*:  $0 - (a::ennreal) = 0$   
**including** *ennreal.lifting*  
*<proof>*

**lemma** *diff-diff-commute-ennreal*:

**fixes**  $a\ b\ c :: \text{ennreal}$  **shows**  $a - b - c = a - c - b$   
 ⟨proof⟩

**lemma** *diff-gr0-ennreal*:  $b < (a::\text{ennreal}) \implies 0 < a - b$

**including** *ennreal.lifting* ⟨proof⟩

**lemma** *divide-le-posI-ennreal*:

**fixes**  $x\ y\ z :: \text{ennreal}$   
**shows**  $x > 0 \implies z \leq x * y \implies z / x \leq y$   
 ⟨proof⟩

**lemma** *add-diff-eq-ennreal*:

**fixes**  $x\ y\ z :: \text{ennreal}$   
**shows**  $z \leq y \implies x + (y - z) = x + y - z$   
 ⟨proof⟩

**lemma** *add-diff-inverse-ennreal*:

**fixes**  $x\ y :: \text{ennreal}$  **shows**  $x \leq y \implies x + (y - x) = y$   
 ⟨proof⟩

**lemma** *add-diff-eq-iff-ennreal[simp]*:

**fixes**  $x\ y :: \text{ennreal}$  **shows**  $x + (y - x) = y \longleftrightarrow x \leq y$   
 ⟨proof⟩

**lemma** *add-diff-le-ennreal*:  $a + b - c \leq a + (b - c::\text{ennreal})$

⟨proof⟩

**lemma** *diff-eq-0-ennreal*:  $a < \text{top} \implies a \leq b \implies a - b = (0::\text{ennreal})$

⟨proof⟩

**lemma** *diff-diff-ennreal'*: **fixes**  $x\ y\ z :: \text{ennreal}$  **shows**  $z \leq y \implies y - z \leq x \implies$   
 $x - (y - z) = x + z - y$

⟨proof⟩

**lemma** *diff-diff-ennreal''*: **fixes**  $x\ y\ z :: \text{ennreal}$

**shows**  $z \leq y \implies x - (y - z) = (\text{if } y - z \leq x \text{ then } x + z - y \text{ else } 0)$   
 ⟨proof⟩

**lemma** *power-less-top-ennreal*: **fixes**  $x :: \text{ennreal}$  **shows**  $x \wedge n < \text{top} \longleftrightarrow x < \text{top}$   
 $\vee n = 0$

⟨proof⟩

**lemma** *ennreal-divide-times*:  $(a / b) * c = a * (c / b :: \text{ennreal})$

⟨proof⟩

**lemma** *diff-less-top-ennreal*:  $a - b < \text{top} \longleftrightarrow a < (\text{top} :: \text{ennreal})$

⟨proof⟩

**lemma** *divide-less-ennreal*:  $b \neq 0 \implies b < \text{top} \implies a / b < c \iff a < (c * b :: \text{ennreal})$

*<proof>*

**lemma** *one-less-numeral[simp]*:  $1 < (\text{numeral } n :: \text{ennreal}) \iff (\text{num.One} < n)$

*<proof>*

**lemma** *divide-eq-1-ennreal*:  $a / b = (1 :: \text{ennreal}) \iff (b \neq \text{top} \wedge b \neq 0 \wedge b = a)$

*<proof>*

**lemma** *ennreal-mult-cancel-left*:  $(a * b = a * c) = (a = \text{top} \wedge b \neq 0 \wedge c \neq 0 \vee a = 0 \vee b = (c :: \text{ennreal}))$

*<proof>*

**lemma** *ennreal-minus-if*:  $\text{ennreal } a - \text{ennreal } b = \text{ennreal } ( \text{if } 0 \leq b \text{ then } ( \text{if } b \leq a \text{ then } a - b \text{ else } 0) \text{ else } a )$

*<proof>*

**lemma** *ennreal-plus-if*:  $\text{ennreal } a + \text{ennreal } b = \text{ennreal } ( \text{if } 0 \leq a \text{ then } ( \text{if } 0 \leq b \text{ then } a + b \text{ else } a) \text{ else } b )$

*<proof>*

**lemma** *ennreal-diff-le-mono-left*:  $a \leq b \implies a - c \leq (b :: \text{ennreal})$

*<proof>*

**lemma** *ennreal-minus-le-iff*:  $a - b \leq c \iff (a \leq b + (c :: \text{ennreal}) \wedge (a = \text{top} \wedge b = \text{top} \implies c = \text{top}))$

*<proof>*

**lemma** *ennreal-le-minus-iff*:  $a \leq b - c \iff (a + c \leq (b :: \text{ennreal}) \vee (a = 0 \wedge b \leq c))$

*<proof>*

**lemma** *diff-add-eq-diff-diff-swap-ennreal*:  $x - (y + z :: \text{ennreal}) = x - y - z$

*<proof>*

**lemma** *diff-add-assoc2-ennreal*:  $b \leq a \implies (a - b + c :: \text{ennreal}) = a + c - b$

*<proof>*

**lemma** *diff-gt-0-iff-gt-ennreal*:  $0 < a - b \iff (a = \text{top} \wedge b = \text{top} \vee b < (a :: \text{ennreal}))$

*<proof>*

**lemma** *diff-eq-0-iff-ennreal*:  $(a - b :: \text{ennreal}) = 0 \iff (a < \text{top} \wedge a \leq b)$

*<proof>*

**lemma** *add-diff-self-ennreal*:  $a + (b - a :: \text{ennreal}) = ( \text{if } a \leq b \text{ then } b \text{ else } a )$

*<proof>*

**lemma** *diff-add-self-ennreal*:  $(b - a + a :: \text{ennreal}) = ( \text{if } a \leq b \text{ then } b \text{ else } a )$

*<proof>*

**lemma** *ennreal-minus-cancel-iff*:

**fixes**  $a\ b\ c :: \text{ennreal}$

**shows**  $a - b = a - c \iff (b = c \vee (a \leq b \wedge a \leq c) \vee a = \text{top})$

*<proof>*

The next lemma is wrong for  $a = \text{top}$ , for  $b = c = 1$  for instance.

**lemma** *ennreal-right-diff-distrib*:

**fixes**  $a\ b\ c :: \text{ennreal}$

**assumes**  $a \neq \text{top}$

**shows**  $a * (b - c) = a * b - a * c$

*<proof>*

**lemma** *SUP-diff-ennreal*:

$c < \text{top} \implies (\text{SUP } i \in I. f\ i - c :: \text{ennreal}) = (\text{SUP } i \in I. f\ i) - c$

*<proof>*

**lemma** *ennreal-SUP-add-right*:

**fixes**  $c :: \text{ennreal}$  **shows**  $I \neq \{\}$   $\implies c + (\text{SUP } i \in I. f\ i) = (\text{SUP } i \in I. c + f\ i)$

*<proof>*

**lemma** *SUP-add-directed-ennreal*:

**fixes**  $f\ g :: - \Rightarrow \text{ennreal}$

**assumes** *directed*:  $\bigwedge i\ j. i \in I \implies j \in I \implies \exists k \in I. f\ i + g\ j \leq f\ k + g\ k$

**shows**  $(\text{SUP } i \in I. f\ i + g\ i) = (\text{SUP } i \in I. f\ i) + (\text{SUP } i \in I. g\ i)$

*<proof>*

**lemma** *enn2real-eq-0-iff*:  $\text{enn2real } x = 0 \iff x = 0 \vee x = \text{top}$

*<proof>*

**lemma** *continuous-on-diff-ennreal*:

$\text{continuous-on } A\ f \implies \text{continuous-on } A\ g \implies (\bigwedge x. x \in A \implies f\ x \neq \text{top}) \implies$   
 $(\bigwedge x. x \in A \implies g\ x \neq \text{top}) \implies \text{continuous-on } A\ (\lambda z. f\ z - g\ z :: \text{ennreal})$

**including** *ennreal.lifting*

*<proof>*

**lemma** *tendsto-diff-ennreal*:

$(f \longrightarrow x)\ F \implies (g \longrightarrow y)\ F \implies x \neq \text{top} \implies y \neq \text{top} \implies ((\lambda z. f\ z - g\ z :: \text{ennreal}) \longrightarrow x - y)\ F$

*<proof>*

**declare** *lim-real-of-ereal* [*tendsto-intros*]

**lemma** *tendsto-enn2real* [*tendsto-intros*]:

**assumes**  $(u \longrightarrow \text{ennreal } l)\ F\ l \geq 0$

**shows**  $((\lambda n. \text{enn2real } (u\ n)) \longrightarrow l)\ F$

*<proof>*

end

## 41 Logarithm of Natural Numbers

**theory** *Log-Nat*  
**imports** *Complex-Main*  
**begin**

### 41.1 Preliminaries

**lemma** *divide-nat-diff-div-nat-less-one*:  
 $\text{real } x / \text{real } b - \text{real } (x \text{ div } b) < 1$  **for**  $x \ b :: \text{nat}$   
 ⟨*proof*⟩

### 41.2 Floorlog

**definition** *floorlog* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$   
**where**  $\text{floorlog } b \ a = (\text{if } a > 0 \wedge b > 1 \text{ then } \text{nat } \lfloor \log b \ a \rfloor + 1 \text{ else } 0)$

**lemma** *floorlog-mono*:  $x \leq y \implies \text{floorlog } b \ x \leq \text{floorlog } b \ y$   
 ⟨*proof*⟩

**lemma** *floorlog-bounds*:  
 $b \wedge (\text{floorlog } b \ x - 1) \leq x \wedge x < b \wedge (\text{floorlog } b \ x)$  **if**  $x > 0 \ b > 1$   
 ⟨*proof*⟩

**lemma** *floorlog-power* [*simp*]:  
 $\text{floorlog } b \ (a * b \wedge c) = \text{floorlog } b \ a + c$  **if**  $a > 0 \ b > 1$   
 ⟨*proof*⟩

**lemma** *floor-log-add-eqI*:  
 $\lfloor \log b \ (a + r) \rfloor = \lfloor \log b \ a \rfloor$  **if**  $b > 1 \ a \geq 1 \ 0 \leq r \ r < 1$   
**for**  $a \ b :: \text{nat}$  **and**  $r :: \text{real}$   
 ⟨*proof*⟩

**lemma** *floor-log-div*:  
 $\lfloor \log b \ x \rfloor = \lfloor \log b \ (x \text{ div } b) \rfloor + 1$  **if**  $b > 1 \ x > 0 \ x \text{ div } b > 0$   
**for**  $b \ x :: \text{nat}$   
 ⟨*proof*⟩

**lemma** *compute-floorlog* [*code*]:  
 $\text{floorlog } b \ x = (\text{if } x > 0 \wedge b > 1 \text{ then } \text{floorlog } b \ (x \text{ div } b) + 1 \text{ else } 0)$   
 ⟨*proof*⟩

**lemma** *floor-log-eq-if*:  
 $\lfloor \log b \ x \rfloor = \lfloor \log b \ y \rfloor$  **if**  $x \text{ div } b = y \text{ div } b \ b > 1 \ x > 0 \ x \text{ div } b \geq 1$   
**for**  $b \ x \ y :: \text{nat}$   
 ⟨*proof*⟩

**lemma** *floorlog-eq-if*:

$\text{floorlog } b \ x = \text{floorlog } b \ y$  **if**  $x \text{ div } b = y \text{ div } b$   $b > 1$   $x > 0$   $x \text{ div } b \geq 1$   
**for**  $b \ x \ y :: \text{nat}$   
 ⟨*proof*⟩

**lemma** *floorlog-leD*:

$\text{floorlog } b \ x \leq w \implies b > 1 \implies x < b \wedge w$   
 ⟨*proof*⟩

**lemma** *floorlog-leI*:

$x < b \wedge w \implies 0 \leq w \implies b > 1 \implies \text{floorlog } b \ x \leq w$   
 ⟨*proof*⟩

**lemma** *floorlog-eq-zero-iff*:

$\text{floorlog } b \ x = 0 \iff b \leq 1 \vee x \leq 0$   
 ⟨*proof*⟩

**lemma** *floorlog-le-iff*:

$\text{floorlog } b \ x \leq w \iff b \leq 1 \vee b > 1 \wedge 0 \leq w \wedge x < b \wedge w$   
 ⟨*proof*⟩

**lemma** *floorlog-ge-SucI*:

$\text{Suc } w \leq \text{floorlog } b \ x$  **if**  $b \wedge w \leq x$   $b > 1$   
 ⟨*proof*⟩

**lemma** *floorlog-geI*:

$w \leq \text{floorlog } b \ x$  **if**  $b \wedge (w - 1) \leq x$   $b > 1$   
 ⟨*proof*⟩

**lemma** *floorlog-geD*:

$b \wedge (w - 1) \leq x$  **if**  $w \leq \text{floorlog } b \ x$   $w > 0$   
 ⟨*proof*⟩

### 41.3

**definition** *ceillog2* ::  $\text{nat} \Rightarrow \text{nat}$  **where**

$\text{ceillog2 } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \text{nat } \lceil \log 2 (\text{real } n) \rceil)$

**lemma** *ceillog2-0* [*simp*]:  $\text{ceillog2 } 0 = 0$

**and** *ceillog2-Suc-0* [*simp*]:  $\text{ceillog2 } (\text{Suc } 0) = 0$

**and** *ceillog2-2* [*simp*]:  $\text{ceillog2 } 2 = 1$

⟨*proof*⟩

**lemma** *ceillog2-le1-eq-0* [*simp*]:  $n \leq 1 \implies \text{ceillog2 } n = 0$

⟨*proof*⟩

**lemma** *ceillog2-2-power* [*simp*]:  $\text{ceillog2 } (2 \wedge n) = n$

⟨*proof*⟩

**lemma** *ceillog2-ge-log*:

**assumes**  $n > 0$

**shows**  $\text{real } (\text{ceillog2 } n) \geq \log 2 (\text{real } n)$

*<proof>*

**lemma** *ceillog2-less-log*:

**assumes**  $n > 0$

**shows**  $\text{real } (\text{ceillog2 } n) < \log 2 (\text{real } n) + 1$

*<proof>*

**lemma** *ceillog2-le-iff*:

**assumes**  $n > 0$

**shows**  $\text{ceillog2 } n \leq l \iff n \leq 2^l$

*<proof>*

**lemma** *ceillog2-ge-iff*:

**assumes**  $n > 0$

**shows**  $\text{ceillog2 } n \geq l \iff 2^l < 2 * n$

*<proof>*

**lemma** *le-two-power-ceillog2*:  $n \leq 2^{\text{ceillog2 } n}$

*<proof>*

**lemma** *two-power-ceillog2-gt*:

**assumes**  $n > 0$

**shows**  $2 * n > 2^{\text{ceillog2 } n}$

*<proof>*

**lemma** *ceillog2-eqI*:

**assumes**  $n \leq 2^l & 2^l < 2 * n$

**shows**  $\text{ceillog2 } n = l$

*<proof>*

**lemma** *ceillog2-rec-even*:

**assumes**  $k > 0$

**shows**  $\text{ceillog2 } (2 * k) = \text{Suc } (\text{ceillog2 } k)$

*<proof>*

**lemma** *ceillog2-mono*:

**assumes**  $m \leq n$

**shows**  $\text{ceillog2 } m \leq \text{ceillog2 } n$

*<proof>*

**lemma** *ceillog2-rec-odd*:

**assumes**  $k > 0$

**shows**  $\text{ceillog2 } (\text{Suc } (2 * k)) = \text{Suc } (\text{ceillog2 } (\text{Suc } k))$

*<proof>*



**lemma** *ceillog2-rec*:

$ceillog2\ n = (if\ n \leq 1\ then\ 0\ else\ 1 + ceillog2\ ((n + 1)\ div\ 2))$   
 ⟨proof⟩

**lemma** *funpow-div2-ceillog2-le-1*:

$((\lambda n. (n + 1)\ div\ 2) \ \sim\sim\ ceillog2\ n)\ n \leq 1$   
 ⟨proof⟩

**fun** *ceillog2-aux* ::  $nat \Rightarrow nat \Rightarrow nat$  **where**

$ceillog2-aux\ acc\ n = (if\ n \leq 1\ then\ acc\ else\ ceillog2-aux\ (acc + 1)\ ((n + 1)\ div\ 2))$

**lemmas** [*simp del*] = *ceillog2-aux.simps*

**lemma** *ceillog2-aux-correct*:  $ceillog2-aux\ acc\ n = ceillog2\ n + acc$

⟨proof⟩

**lemma** *ceillog2-code* [*code*]:  $ceillog2\ n = ceillog2-aux\ 0\ n$

⟨proof⟩

#### 41.4 Bitlen

**definition** *bitlen* ::  $int \Rightarrow int$

**where**  $bitlen\ a = floorlog\ 2\ (nat\ a)$

**lemma** *bitlen-alt-def*:

$bitlen\ a = (if\ a > 0\ then\ \lfloor \log\ 2\ a \rfloor + 1\ else\ 0)$   
 ⟨proof⟩

**lemma** *bitlen-zero* [*simp*]:

$bitlen\ 0 = 0$   
 ⟨proof⟩

**lemma** *bitlen-nonneg*:

$0 \leq bitlen\ x$   
 ⟨proof⟩

**lemma** *bitlen-bounds*:

$2^{\wedge nat\ (bitlen\ x - 1)} \leq x \wedge x < 2^{\wedge nat\ (bitlen\ x)}\ \mathbf{if}\ x > 0$   
 ⟨proof⟩

**lemma** *bitlen-pow2* [*simp*]:

$bitlen\ (b * 2^{\wedge c}) = bitlen\ b + c\ \mathbf{if}\ b > 0$   
 ⟨proof⟩

**lemma** *compute-bitlen* [*code*]:

$\text{bitlen } x = (\text{if } x > 0 \text{ then } \text{bitlen } (x \text{ div } 2) + 1 \text{ else } 0)$   
 ⟨proof⟩

**lemma** *bitlen-eq-zero-iff*:  
 $\text{bitlen } x = 0 \iff x \leq 0$   
 ⟨proof⟩

**lemma** *bitlen-div*:  
 $1 \leq \text{real-of-int } m / 2^{\text{nat } (\text{bitlen } m - 1)}$   
**and**  $\text{real-of-int } m / 2^{\text{nat } (\text{bitlen } m - 1)} < 2 \text{ if } 0 < m$   
 ⟨proof⟩

**lemma** *bitlen-le-iff-floorlog*:  
 $\text{bitlen } x \leq w \iff w \geq 0 \wedge \text{floorlog } 2 (\text{nat } x) \leq \text{nat } w$   
 ⟨proof⟩

**lemma** *bitlen-le-iff-power*:  
 $\text{bitlen } x \leq w \iff w \geq 0 \wedge x < 2^{\text{nat } w}$   
 ⟨proof⟩

**lemma** *less-power-nat-iff-bitlen*:  
 $x < 2^w \iff \text{bitlen } (\text{int } x) \leq w$   
 ⟨proof⟩

**lemma** *bitlen-ge-iff-power*:  
 $w \leq \text{bitlen } x \iff w \leq 0 \vee 2^{\text{nat } w - 1} \leq x$   
 ⟨proof⟩

**lemma** *bitlen-twopow-add-eq*:  
 $\text{bitlen } (2^w + b) = w + 1 \text{ if } 0 \leq b < 2^w$   
 ⟨proof⟩

end

## 42 Various algebraic structures combined with a lattice

**theory** *Lattice-Algebras*  
**imports** *Complex-Main*  
**begin**

**class** *semilattice-inf-ab-group-add* = *ordered-ab-group-add* + *semilattice-inf*  
**begin**

**lemma** *add-inf-distrib-left*:  $a + \text{inf } b \ c = \text{inf } (a + b) \ (a + c)$  (**is** ?L=?R)  
 ⟨proof⟩

**lemma** *add-inf-distrib-right*:  $\text{inf } a \ b + c = \text{inf } (a + c) \ (b + c)$

```

    <proof>

end

class semilattice-sup-ab-group-add = ordered-ab-group-add + semilattice-sup
begin

lemma add-sup-distrib-left: a + sup b c = sup (a + b) (a + c) (is ?L = ?R)
<proof>

lemma add-sup-distrib-right: sup a b + c = sup (a + c) (b + c)
<proof>

end

class lattice-ab-group-add = ordered-ab-group-add + lattice
begin

subclass semilattice-inf-ab-group-add <proof>
subclass semilattice-sup-ab-group-add <proof>

lemmas add-sup-inf-distrib =
    add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

lemma inf-eq-neg-sup: inf a b = - sup (- a) (- b)
<proof>

lemma sup-eq-neg-inf: sup a b = - inf (- a) (- b)
<proof>

lemma neg-inf-eq-sup: - inf a b = sup (- a) (- b)
<proof>

lemma diff-inf-eq-sup: a - inf b c = a + sup (- b) (- c)
<proof>

lemma neg-sup-eq-inf: - sup a b = inf (- a) (- b)
<proof>

lemma diff-sup-eq-inf: a - sup b c = a + inf (- b) (- c)
<proof>

lemma add-eq-inf-sup: a + b = sup a b + inf a b
<proof>

```

## 42.1 Positive Part, Negative Part, Absolute Value

```

definition nprt :: 'a ⇒ 'a
  where nprt x = inf x 0

```

**definition**  $pprt :: 'a \Rightarrow 'a$

**where**  $pprt\ x = sup\ x\ 0$

**lemma**  $pprt\ neg$ :  $pprt\ (-\ x) = -\ nprt\ x$   
 $\langle proof \rangle$

**lemma**  $nprr\ neg$ :  $nprr\ (-\ x) = -\ pprt\ x$   
 $\langle proof \rangle$

**lemma**  $prts$ :  $a = pprt\ a + nprr\ a$   
 $\langle proof \rangle$

**lemma**  $zero\ le\ pprt[simp]$ :  $0 \leq pprt\ a$   
 $\langle proof \rangle$

**lemma**  $nprr\ le\ zero[simp]$ :  $nprr\ a \leq 0$   
 $\langle proof \rangle$

**lemma**  $le\ eq\ neg$ :  $a \leq -\ b \longleftrightarrow a + b \leq 0$   
**(is ?lhs = ?rhs)**  
 $\langle proof \rangle$

**lemma**  $pprt\ 0[simp]$ :  $pprt\ 0 = 0$   $\langle proof \rangle$

**lemma**  $nprr\ 0[simp]$ :  $nprr\ 0 = 0$   $\langle proof \rangle$

**lemma**  $pprt\ eq\ id\ [simp, no-atp]$ :  $0 \leq x \implies pprt\ x = x$   
 $\langle proof \rangle$

**lemma**  $nprr\ eq\ id\ [simp, no-atp]$ :  $x \leq 0 \implies nprr\ x = x$   
 $\langle proof \rangle$

**lemma**  $pprt\ eq\ 0\ [simp, no-atp]$ :  $x \leq 0 \implies pprt\ x = 0$   
 $\langle proof \rangle$

**lemma**  $nprr\ eq\ 0\ [simp, no-atp]$ :  $0 \leq x \implies nprr\ x = 0$   
 $\langle proof \rangle$

**lemma**  $sup\ 0\ imp\ 0$ :

**assumes**  $sup\ a\ (-\ a) = 0$

**shows**  $a = 0$

$\langle proof \rangle$

**lemma**  $inf\ 0\ imp\ 0$ :  $inf\ a\ (-\ a) = 0 \implies a = 0$   
 $\langle proof \rangle$

**lemma**  $inf\ 0\ eq\ 0\ [simp]$ :  $inf\ a\ (-\ a) = 0 \longleftrightarrow a = 0$   
 $\langle proof \rangle$

**lemma** *sup-0-eq-0* [*simp*]:  $\text{sup } a \ (- \ a) = 0 \longleftrightarrow a = 0$   
 ⟨*proof*⟩

**lemma** *zero-le-double-add-iff-zero-le-single-add* [*simp*]:  $0 \leq a + a \longleftrightarrow 0 \leq a$   
 (is ?lhs  $\longleftrightarrow$  ?rhs)  
 ⟨*proof*⟩

**lemma** *double-zero* [*simp*]:  $a + a = 0 \longleftrightarrow a = 0$   
 ⟨*proof*⟩

**lemma** *zero-less-double-add-iff-zero-less-single-add* [*simp*]:  $0 < a + a \longleftrightarrow 0 < a$   
 ⟨*proof*⟩

**lemma** *double-add-le-zero-iff-single-add-le-zero* [*simp*]:  $a + a \leq 0 \longleftrightarrow a \leq 0$   
 ⟨*proof*⟩

**lemma** *double-add-less-zero-iff-single-less-zero* [*simp*]:  $a + a < 0 \longleftrightarrow a < 0$   
 ⟨*proof*⟩

**declare** *neg-inf-eq-sup* [*simp*]  
 and *neg-sup-eq-inf* [*simp*]  
 and *diff-inf-eq-sup* [*simp*]  
 and *diff-sup-eq-inf* [*simp*]

**lemma** *le-minus-self-iff*:  $a \leq - \ a \longleftrightarrow a \leq 0$   
 ⟨*proof*⟩

**lemma** *minus-le-self-iff*:  $- \ a \leq a \longleftrightarrow 0 \leq a$   
 ⟨*proof*⟩

**lemma** *zero-le-iff-zero-nprt*:  $0 \leq a \longleftrightarrow \text{nprt } a = 0$   
 ⟨*proof*⟩

**lemma** *le-zero-iff-zero-pprt*:  $a \leq 0 \longleftrightarrow \text{pprt } a = 0$   
 ⟨*proof*⟩

**lemma** *le-zero-iff-pprt-id*:  $0 \leq a \longleftrightarrow \text{pprt } a = a$   
 ⟨*proof*⟩

**lemma** *zero-le-iff-nprt-id*:  $a \leq 0 \longleftrightarrow \text{nprt } a = a$   
 ⟨*proof*⟩

**lemma** *pprt-mono* [*simp*, *no-atp*]:  $a \leq b \implies \text{pprt } a \leq \text{pprt } b$   
 ⟨*proof*⟩

**lemma** *nprt-mono* [*simp*, *no-atp*]:  $a \leq b \implies \text{nprt } a \leq \text{nprt } b$   
 ⟨*proof*⟩

**end**

**lemmas** *add-sup-inf-distrib* =  
*add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

**class** *lattice-ab-group-add-abs* = *lattice-ab-group-add* + *abs* +  
**assumes** *abs-lattice*:  $|a| = \text{sup } a \ (- a)$   
**begin**

**lemma** *abs-prts*:  $|a| = \text{pprt } a \ - \ \text{nppt } a$   
 $\langle \text{proof} \rangle$

**subclass** *ordered-ab-group-add-abs*  
 $\langle \text{proof} \rangle$

**end**

**lemma** *sup-eq-if*:  
**fixes**  $a :: 'a :: \{\text{lattice-ab-group-add}, \text{linorder}\}$   
**shows**  $\text{sup } a \ (- a) = (\text{if } a < 0 \text{ then } - a \ \text{else } a)$   
 $\langle \text{proof} \rangle$

**lemma** *abs-if-lattice*:  
**fixes**  $a :: 'a :: \{\text{lattice-ab-group-add-abs}, \text{linorder}\}$   
**shows**  $|a| = (\text{if } a < 0 \text{ then } - a \ \text{else } a)$   
 $\langle \text{proof} \rangle$

**lemma** *estimate-by-abs*:  
**fixes**  $a \ b \ c :: 'a :: \text{lattice-ab-group-add-abs}$   
**assumes**  $a + b \leq c$   
**shows**  $a \leq c + |b|$   
 $\langle \text{proof} \rangle$

**class** *lattice-ring* = *ordered-ring* + *lattice-ab-group-add-abs*  
**begin**

**subclass** *semilattice-inf-ab-group-add*  $\langle \text{proof} \rangle$   
**subclass** *semilattice-sup-ab-group-add*  $\langle \text{proof} \rangle$

**end**

**lemma** *abs-le-mult*:  
**fixes**  $a \ b :: 'a :: \text{lattice-ring}$   
**shows**  $|a * b| \leq |a| * |b|$   
 $\langle \text{proof} \rangle$

**instance** *lattice-ring*  $\subseteq$  *ordered-ring-abs*  
 $\langle \text{proof} \rangle$

```

lemma mult-le-prts:
  fixes a b :: 'a::lattice-ring
  assumes a1 ≤ a
    and a ≤ a2
    and b1 ≤ b
    and b ≤ b2
  shows a * b ≤
    pprt a2 * pprt b2 + pprt a1 * nprt b2 + nprt a2 * pprt b1 + nprt a1 * nprt
b1
  ⟨proof⟩

lemma mult-ge-prts:
  fixes a b :: 'a::lattice-ring
  assumes a1 ≤ a
    and a ≤ a2
    and b1 ≤ b
    and b ≤ b2
  shows a * b ≥
    nprt a1 * pprt b2 + nprt a2 * nprt b2 + pprt a1 * pprt b1 + pprt a2 * nprt
b1
  ⟨proof⟩

instance int :: lattice-ring
  ⟨proof⟩

instance real :: lattice-ring
  ⟨proof⟩

end

```

## 43 Floating-Point Numbers

```

theory Float
imports Log-Nat Lattice-Algebras
begin

definition float = {m * 2 powr e | (m :: int) (e :: int). True}

typedef float = float
  morphisms real-of-float float-of
  ⟨proof⟩

setup-lifting type-definition-float

declare real-of-float [code-unfold]

lemmas float-of-inject[simp]

declare [[coercion real-of-float :: float ⇒ real]]

```

**lemma** *real-of-float-eq*:  $f1 = f2 \iff \text{real-of-float } f1 = \text{real-of-float } f2$  **for**  $f1\ f2 :: \text{float}$   
 ⟨*proof*⟩

**declare** *real-of-float-inverse*[simp] *float-of-inverse* [simp]  
**declare** *real-of-float* [simp]

### 43.1 Real operations preserving the representation as floating point number

**lemma** *floatI*:  $m * 2^{\text{powr } e} = x \implies x \in \text{float}$  **for**  $m\ e :: \text{int}$   
 ⟨*proof*⟩

**lemma** *zero-float*[simp]:  $0 \in \text{float}$   
 ⟨*proof*⟩

**lemma** *one-float*[simp]:  $1 \in \text{float}$   
 ⟨*proof*⟩

**lemma** *numeral-float*[simp]:  $\text{numeral } i \in \text{float}$   
 ⟨*proof*⟩

**lemma** *neg-numeral-float*[simp]:  $-\text{numeral } i \in \text{float}$   
 ⟨*proof*⟩

**lemma** *real-of-int-float*[simp]:  $\text{real-of-int } x \in \text{float}$  **for**  $x :: \text{int}$   
 ⟨*proof*⟩

**lemma** *real-of-nat-float*[simp]:  $\text{real } x \in \text{float}$  **for**  $x :: \text{nat}$   
 ⟨*proof*⟩

**lemma** *two-powr-int-float*[simp]:  $2^{\text{powr } ( \text{real-of-int } i )} \in \text{float}$  **for**  $i :: \text{int}$   
 ⟨*proof*⟩

**lemma** *two-powr-nat-float*[simp]:  $2^{\text{powr } ( \text{real } i )} \in \text{float}$  **for**  $i :: \text{nat}$   
 ⟨*proof*⟩

**lemma** *two-powr-minus-int-float*[simp]:  $2^{\text{powr } - ( \text{real-of-int } i )} \in \text{float}$  **for**  $i :: \text{int}$   
 ⟨*proof*⟩

**lemma** *two-powr-minus-nat-float*[simp]:  $2^{\text{powr } - ( \text{real } i )} \in \text{float}$  **for**  $i :: \text{nat}$   
 ⟨*proof*⟩

**lemma** *two-powr-numeral-float*[simp]:  $2^{\text{powr } \text{numeral } i} \in \text{float}$   
 ⟨*proof*⟩

**lemma** *two-powr-neg-numeral-float*[simp]:  $2^{\text{powr } - \text{numeral } i} \in \text{float}$   
 ⟨*proof*⟩



**lemma** *two-pow-float[simp]*:  $2^n \in \text{float}$   
 ⟨proof⟩

**lemma** *plus-float[simp]*:  $r \in \text{float} \implies p \in \text{float} \implies r + p \in \text{float}$   
 ⟨proof⟩

**lemma** *uminus-float[simp]*:  $x \in \text{float} \implies -x \in \text{float}$   
 ⟨proof⟩

**lemma** *times-float[simp]*:  $x \in \text{float} \implies y \in \text{float} \implies x * y \in \text{float}$   
 ⟨proof⟩

**lemma** *minus-float[simp]*:  $x \in \text{float} \implies y \in \text{float} \implies x - y \in \text{float}$   
 ⟨proof⟩

**lemma** *abs-float[simp]*:  $x \in \text{float} \implies |x| \in \text{float}$   
 ⟨proof⟩

**lemma** *sgn-of-float[simp]*:  $x \in \text{float} \implies \text{sgn } x \in \text{float}$   
 ⟨proof⟩

**lemma** *div-power-2-float[simp]*:  $x \in \text{float} \implies x / 2^d \in \text{float}$   
 ⟨proof⟩

**lemma** *div-power-2-int-float[simp]*:  $x \in \text{float} \implies x / (2::\text{int})^d \in \text{float}$   
 ⟨proof⟩

**lemma** *div-numeral-Bit0-float[simp]*:  
**assumes**  $x / \text{numeral } n \in \text{float}$   
**shows**  $x / (\text{numeral } (\text{Num.Bit0 } n)) \in \text{float}$   
 ⟨proof⟩

**lemma** *div-neg-numeral-Bit0-float[simp]*:  
**assumes**  $x / \text{numeral } n \in \text{float}$   
**shows**  $x / (- \text{numeral } (\text{Num.Bit0 } n)) \in \text{float}$   
 ⟨proof⟩

**lemma** *power-float[simp]*:  
**assumes**  $a \in \text{float}$   
**shows**  $a^b \in \text{float}$   
 ⟨proof⟩

**lift-definition** *Float* ::  $\text{int} \Rightarrow \text{int} \Rightarrow \text{float}$  **is**  $\lambda(m::\text{int}) (e::\text{int}). m * 2^{\text{powr } e}$   
 ⟨proof⟩

**declare** *Float.rep-eq[simp]*

**code-datatype** *Float*

**lemma** *compute-real-of-float*[code]:  
*real-of-float* (Float  $m\ e$ ) = (if  $e \geq 0$  then  $m * 2^{\text{nat } e}$  else  $m / 2^{\text{nat } (-e)}$ )  
 ⟨proof⟩

### 43.2 Arithmetic operations on floating point numbers

**instantiation** *float* :: {ring-1, linorder, linordered-ring, linordered-idom, numeral, equal}  
**begin**

**lift-definition** *zero-float* :: float is 0 ⟨proof⟩  
**declare** *zero-float.rep-eq*[simp]

**lift-definition** *one-float* :: float is 1 ⟨proof⟩  
**declare** *one-float.rep-eq*[simp]

**lift-definition** *plus-float* :: float  $\Rightarrow$  float  $\Rightarrow$  float is (+) ⟨proof⟩  
**declare** *plus-float.rep-eq*[simp]

**lift-definition** *times-float* :: float  $\Rightarrow$  float  $\Rightarrow$  float is (\*) ⟨proof⟩  
**declare** *times-float.rep-eq*[simp]

**lift-definition** *minus-float* :: float  $\Rightarrow$  float  $\Rightarrow$  float is (-) ⟨proof⟩  
**declare** *minus-float.rep-eq*[simp]

**lift-definition** *uminus-float* :: float  $\Rightarrow$  float is uminus ⟨proof⟩  
**declare** *uminus-float.rep-eq*[simp]

**lift-definition** *abs-float* :: float  $\Rightarrow$  float is abs ⟨proof⟩  
**declare** *abs-float.rep-eq*[simp]

**lift-definition** *sgn-float* :: float  $\Rightarrow$  float is sgn ⟨proof⟩  
**declare** *sgn-float.rep-eq*[simp]

**lift-definition** *equal-float* :: float  $\Rightarrow$  float  $\Rightarrow$  bool is (=) :: real  $\Rightarrow$  real  $\Rightarrow$  bool  
 ⟨proof⟩

**lift-definition** *less-eq-float* :: float  $\Rightarrow$  float  $\Rightarrow$  bool is ( $\leq$ ) ⟨proof⟩  
**declare** *less-eq-float.rep-eq*[simp]

**lift-definition** *less-float* :: float  $\Rightarrow$  float  $\Rightarrow$  bool is (<) ⟨proof⟩  
**declare** *less-float.rep-eq*[simp]

**instance**  
 ⟨proof⟩

**end**

**lemma** *real-of-float* [simp]: *real-of-float* (of-nat  $n$ ) = of-nat  $n$

*<proof>*

**lemma** *real-of-float-of-int-eq [simp]: real-of-float (of-int z) = of-int z*  
*<proof>*

**lemma** *Float-0-eq-0[simp]: Float 0 e = 0*  
*<proof>*

**lemma** *real-of-float-power[simp]: real-of-float (f<sup>n</sup>) = real-of-float f<sup>n</sup> for f :: float*  
*<proof>*

**lemma** *real-of-float-min: real-of-float (min x y) = min (real-of-float x) (real-of-float y)*  
**and** *real-of-float-max: real-of-float (max x y) = max (real-of-float x) (real-of-float y)*  
**for** *x y :: float*  
*<proof>*

**instance** *float :: unbounded-dense-linorder*  
*<proof>*

**instantiation** *float :: lattice-ab-group-add*  
**begin**

**definition** *inf-float :: float ⇒ float ⇒ float*  
**where** *inf-float a b = min a b*

**definition** *sup-float :: float ⇒ float ⇒ float*  
**where** *sup-float a b = max a b*

**instance**  
*<proof>*

**end**

**lemma** *float-numeral[simp]: real-of-float (numeral x :: float) = numeral x*  
*<proof>*

**lemma** *transfer-numeral [transfer-rule]:*  
*rel-fun (=) pcr-float (numeral :: - ⇒ real) (numeral :: - ⇒ float)*  
*<proof>*

**lemma** *float-neg-numeral[simp]: real-of-float (- numeral x :: float) = - numeral x*  
*<proof>*

**lemma** *transfer-neg-numeral [transfer-rule]:*  
*rel-fun (=) pcr-float (- numeral :: - ⇒ real) (- numeral :: - ⇒ float)*  
*<proof>*

**lemma** *float-of-numeral*: numeral  $k = \text{float-of } (\text{numeral } k)$   
**and** *float-of-neg-numeral*:  $-\text{numeral } k = \text{float-of } (-\text{numeral } k)$   
 ⟨*proof*⟩

### 43.3 Quickcheck

**instantiation** *float* :: *exhaustive*  
**begin**

**definition** *exhaustive-float* **where**  
*exhaustive-float*  $f$   $d =$   
*Quickcheck-Exhaustive.exhaustive* ( $\lambda x.$  *Quickcheck-Exhaustive.exhaustive* ( $\lambda y.$   $f$   
 (*Float*  $x$   $y$ ))  $d$ )  $d$

**instance** ⟨*proof*⟩

**end**

**context**  
**includes** *term-syntax*  
**begin**

**definition** [*code-unfold*]:  
*valtermify-float*  $x$   $y = \text{Code-Evaluation.valtermify } \text{Float } \{\cdot\} x \{\cdot\} y$

**end**

**instantiation** *float* :: *full-exhaustive*  
**begin**

**definition**  
*full-exhaustive-float*  $f$   $d =$   
*Quickcheck-Exhaustive.full-exhaustive*  
 ( $\lambda x.$  *Quickcheck-Exhaustive.full-exhaustive* ( $\lambda y.$   $f$  (*valtermify-float*  $x$   $y$ ))  $d$ )  $d$

**instance** ⟨*proof*⟩

**end**

**instantiation** *float* :: *random*  
**begin**

**definition** *Quickcheck-Random.random*  $i =$   
*scomp* (*Quickcheck-Random.random* ( $2^{\wedge} \text{nat-of-natural } i$ ))  
 ( $\lambda \text{man. scomp } (\text{Quickcheck-Random.random } i) (\lambda \text{exp. Pair } (\text{valtermify-float}$   
*man*  $\text{exp}))$ )

**instance** ⟨*proof*⟩

end

#### 43.4 Represent floats as unique mantissa and exponent

**lemma** *int-induct-abs*[*case-names less*]:

**fixes**  $j :: int$   
**assumes**  $H: \bigwedge n. (\bigwedge i. |i| < |n| \implies P i) \implies P n$   
**shows**  $P j$   
*<proof>*

**lemma** *int-cancel-factors*:

**fixes**  $n :: int$   
**assumes**  $1 < r$   
**shows**  $n = 0 \vee (\exists k i. n = k * r ^ i \wedge \neg r \text{ dvd } k)$   
*<proof>*

**lemma** *mult-powr-eq-mult-powr-iff-asym*:

**fixes**  $m1 m2 e1 e2 :: int$   
**assumes**  $m1: \neg 2 \text{ dvd } m1$   
**and**  $e1 \leq e2$   
**shows**  $m1 * 2 \text{ powr } e1 = m2 * 2 \text{ powr } e2 \iff m1 = m2 \wedge e1 = e2$   
*(is ?lhs  $\iff$  ?rhs)*  
*<proof>*

**lemma** *mult-powr-eq-mult-powr-iff*:

$\neg 2 \text{ dvd } m1 \implies \neg 2 \text{ dvd } m2 \implies m1 * 2 \text{ powr } e1 = m2 * 2 \text{ powr } e2 \iff m1 = m2 \wedge e1 = e2$   
**for**  $m1 m2 e1 e2 :: int$   
*<proof>*

**lemma** *floatE-normed*:

**assumes**  $x: x \in \text{float}$   
**obtains** *(zero)*  $x = 0$   
 | *(powr)*  $m e :: int$  **where**  $x = m * 2 \text{ powr } e \wedge \neg 2 \text{ dvd } m \wedge x \neq 0$   
*<proof>*

**lemma** *float-normed-cases*:

**fixes**  $f :: float$   
**obtains** *(zero)*  $f = 0$   
 | *(powr)*  $m e :: int$  **where**  $\text{real-of-float } f = m * 2 \text{ powr } e \wedge \neg 2 \text{ dvd } m \wedge f \neq 0$   
*<proof>*

**definition** *mantissa*  $:: float \Rightarrow int$

**where**  $\text{mantissa } f =$   
 $\text{fst } (\text{SOME } p::int \times int. (f = 0 \wedge \text{fst } p = 0 \wedge \text{snd } p = 0) \vee$   
 $(f \neq 0 \wedge \text{real-of-float } f = \text{real-of-int } (\text{fst } p) * 2 \text{ powr } \text{real-of-int } (\text{snd } p) \wedge \neg$   
 $2 \text{ dvd } \text{fst } p))$

**definition** *exponent* :: float  $\Rightarrow$  int

**where** *exponent* *f* =

*snd* (SOME *p*:int  $\times$  int. (*f* = 0  $\wedge$  *fst* *p* = 0  $\wedge$  *snd* *p* = 0)  $\vee$   
 (*f*  $\neq$  0  $\wedge$  *real-of-float* *f* = *real-of-int* (*fst* *p*) \* 2 *powr* *real-of-int* (*snd* *p*)  $\wedge$   $\neg$   
 2 *dvd* *fst* *p*))

**lemma** *exponent-0[simp]*: *exponent* 0 = 0 (is ?E)

**and** *mantissa-0[simp]*: *mantissa* 0 = 0 (is ?M)

*<proof>*

**lemma** *mantissa-exponent*: *real-of-float* *f* = *mantissa* *f* \* 2 *powr* *exponent* *f* (is ?E)

**and** *mantissa-not-dvd*: *f*  $\neq$  0  $\implies$   $\neg$  2 *dvd* *mantissa* *f* (is -  $\implies$  ?D)

*<proof>*

**lemma** *mantissa-noteq-0*: *f*  $\neq$  0  $\implies$  *mantissa* *f*  $\neq$  0

*<proof>*

**lemma** *mantissa-eq-zero-iff*: *mantissa* *x* = 0  $\longleftrightarrow$  *x* = 0

(is ?lhs  $\longleftrightarrow$  ?rhs)

*<proof>*

**lemma** *mantissa-pos-iff*: 0 < *mantissa* *x*  $\longleftrightarrow$  0 < *x*

*<proof>*

**lemma** *mantissa-nonneg-iff*: 0  $\leq$  *mantissa* *x*  $\longleftrightarrow$  0  $\leq$  *x*

*<proof>*

**lemma** *mantissa-neg-iff*: 0 > *mantissa* *x*  $\longleftrightarrow$  0 > *x*

*<proof>*

**lemma**

**fixes** *m e* :: int

**defines** *f*  $\equiv$  *float-of* (*m* \* 2 *powr* *e*)

**assumes** *dvd*:  $\neg$  2 *dvd* *m*

**shows** *mantissa-float*: *mantissa* *f* = *m* (is ?M)

**and** *exponent-float*: *m*  $\neq$  0  $\implies$  *exponent* *f* = *e* (is -  $\implies$  ?E)

*<proof>*

### 43.5 Compute arithmetic operations

**lemma** *Float-mantissa-exponent*: *Float* (*mantissa* *f*) (*exponent* *f*) = *f*

*<proof>*

**lemma** *Float-cases* [*cases type*: float]:

**fixes** *f* :: float

**obtains** (*Float*) *m e* :: int **where** *f* = *Float* *m e*

*<proof>*

**lemma** *denormalize-shift*:

**assumes** *f-def*:  $f = \text{Float } m \ e$

**and** *not-0*:  $f \neq 0$

**obtains** *i* **where**  $m = \text{mantissa } f * 2^i \ e = \text{exponent } f - i$

*<proof>*

**context**

**begin**

**qualified lemma** *compute-float-zero*[*code-unfold, code*]:  $0 = \text{Float } 0 \ 0$

*<proof>* **lemma** *compute-float-one*[*code-unfold, code*]:  $1 = \text{Float } 1 \ 0$

*<proof>*

**lift-definition** *normfloat* :: *float*  $\Rightarrow$  *float* **is**  $\lambda x. x$  *<proof>*

**lemma** *normfloat-id*[*simp*]: *normfloat*  $x = x$  *<proof>* **lemma** *compute-normfloat*[*code*]:

*normfloat* (*Float*  $m \ e$ ) =

(if  $m \bmod 2 = 0 \wedge m \neq 0$  then *normfloat* (*Float* ( $m \text{ div } 2$ ) ( $e + 1$ ))

else if  $m = 0$  then 0 else *Float*  $m \ e$ )

*<proof>* **lemma** *compute-float-numeral*[*code-abbrev*]: *Float* (*numeral*  $k$ ) 0 = *numeral*  $k$

*<proof>* **lemma** *compute-float-neg-numeral*[*code-abbrev*]: *Float* ( $- \text{numeral } k$ ) 0 =  $- \text{numeral } k$

*<proof>* **lemma** *compute-float-uminus*[*code*]:  $- \text{Float } m1 \ e1 = \text{Float } (- m1) \ e1$

*<proof>* **lemma** *compute-float-times*[*code*]: *Float*  $m1 \ e1 * \text{Float } m2 \ e2 = \text{Float } (m1 * m2) (e1 + e2)$

*<proof>* **lemma** *compute-float-plus*[*code*]:

*Float*  $m1 \ e1 + \text{Float } m2 \ e2 =$

(if  $m1 = 0$  then *Float*  $m2 \ e2$

else if  $m2 = 0$  then *Float*  $m1 \ e1$

else if  $e1 \leq e2$  then *Float* ( $m1 + m2 * 2^{\text{nat } (e2 - e1)}$ )  $e1$

else *Float* ( $m2 + m1 * 2^{\text{nat } (e1 - e2)}$ )  $e2$ )

*<proof>* **lemma** *compute-float-minus*[*code*]:  $f - g = f + (-g)$  **for**  $f \ g :: \text{float}$

*<proof>* **lemma** *compute-float-sgn*[*code*]:

*sgn* (*Float*  $m1 \ e1$ ) = (if  $0 < m1$  then 1 else if  $m1 < 0$  then  $-1$  else 0)

*<proof>*

**lift-definition** *is-float-pos* :: *float*  $\Rightarrow$  *bool* **is**  $(<) \ 0 :: \text{real} \Rightarrow \text{bool}$  *<proof>* **lemma**

*compute-is-float-pos*[*code*]: *is-float-pos* (*Float*  $m \ e$ )  $\longleftrightarrow 0 < m$

*<proof>*

**lift-definition** *is-float-nonneg* :: *float*  $\Rightarrow$  *bool* **is**  $(\leq) \ 0 :: \text{real} \Rightarrow \text{bool}$  *<proof>*

**lemma** *compute-is-float-nonneg*[*code*]: *is-float-nonneg* (*Float*  $m \ e$ )  $\longleftrightarrow 0 \leq m$

*<proof>*

**lift-definition** *is-float-zero* :: *float*  $\Rightarrow$  *bool* **is**  $(=) \ 0 :: \text{real} \Rightarrow \text{bool}$  *<proof>* **lemma**

*compute-is-float-zero*[*code*]: *is-float-zero* (*Float*  $m \ e$ )  $\longleftrightarrow 0 = m$

*<proof>* **lemma** *compute-float-abs*[*code*]:  $|\text{Float } m \ e| = \text{Float } |m| \ e$

*<proof>* **lemma** *compute-float-eq*[*code*]: *equal-class.equal*  $f \ g = \text{is-float-zero } (f - g)$

*<proof>*

**end**

### 43.6 Lemmas for types *real*, *nat*, *int*

**lemmas** *real-of-ints* =  
*of-int-add*  
*of-int-minus*  
*of-int-diff*  
*of-int-mult*  
*of-int-power*  
*of-int-numeral of-int-neg-numeral*

**lemmas** *int-of-reals* = *real-of-ints[symmetric]*

### 43.7 Rounding Real Numbers

**definition** *round-down* :: *int*  $\Rightarrow$  *real*  $\Rightarrow$  *real*  
**where** *round-down prec x* =  $\lfloor x * 2^{\text{powr } \text{prec}} \rfloor * 2^{\text{powr } -\text{prec}}$

**definition** *round-up* :: *int*  $\Rightarrow$  *real*  $\Rightarrow$  *real*  
**where** *round-up prec x* =  $\lceil x * 2^{\text{powr } \text{prec}} \rceil * 2^{\text{powr } -\text{prec}}$

**lemma** *round-down-float[simp]*: *round-down prec x*  $\in$  *float*  
*<proof>*

**lemma** *round-up-float[simp]*: *round-up prec x*  $\in$  *float*  
*<proof>*

**lemma** *round-up*:  $x \leq \text{round-up } \text{prec } x$   
*<proof>*

**lemma** *round-down*: *round-down prec x*  $\leq x$   
*<proof>*

**lemma** *round-up-0[simp]*: *round-up p 0* = 0  
*<proof>*

**lemma** *round-down-0[simp]*: *round-down p 0* = 0  
*<proof>*

**lemma** *round-up-diff-round-down*: *round-up prec x* - *round-down prec x*  $\leq 2^{\text{powr } -\text{prec}}$   
*<proof>*

**lemma** *round-down-shift*: *round-down p (x \* 2<sup>powr k</sup>)* =  $2^{\text{powr } k} * \text{round-down } (p + k) x$   
*<proof>*



**lemma** *round-up-shift*:  $\text{round-up } p (x * 2 \text{ powr } k) = 2 \text{ powr } k * \text{round-up } (p + k) x$

*<proof>*

**lemma** *round-up-uminus-eq*:  $\text{round-up } p (-x) = - \text{round-down } p x$   
**and** *round-down-uminus-eq*:  $\text{round-down } p (-x) = - \text{round-up } p x$

*<proof>*

**lemma** *round-up-mono*:  $x \leq y \implies \text{round-up } p x \leq \text{round-up } p y$

*<proof>*

**lemma** *round-up-le1*:

**assumes**  $x \leq 1 \text{ prec} \geq 0$

**shows**  $\text{round-up } \text{prec } x \leq 1$

*<proof>*

**lemma** *round-up-less1*:

**assumes**  $x < 1 / 2 \text{ p} > 0$

**shows**  $\text{round-up } p x < 1$

*<proof>*

**lemma** *round-down-ge1*:

**assumes**  $x: x \geq 1$

**assumes** *prec*:  $p \geq - \log 2 x$

**shows**  $1 \leq \text{round-down } p x$

*<proof>*

**lemma** *round-up-le0*:  $x \leq 0 \implies \text{round-up } p x \leq 0$

*<proof>*

## 43.8 Rounding Floats

**definition** *div-twopow* ::  $\text{int} \Rightarrow \text{nat} \Rightarrow \text{int}$

**where** [*simp*]:  $\text{div-twopow } x n = x \text{ div } (2 \wedge n)$

**definition** *mod-twopow* ::  $\text{int} \Rightarrow \text{nat} \Rightarrow \text{int}$

**where** [*simp*]:  $\text{mod-twopow } x n = x \text{ mod } (2 \wedge n)$

**lemma** *compute-div-twopow*[*code*]:

$\text{div-twopow } x n = (\text{if } x = 0 \vee x = -1 \vee n = 0 \text{ then } x \text{ else } \text{div-twopow } (x \text{ div } 2) (n - 1))$

*<proof>*

**lemma** *compute-mod-twopow*[*code*]:

$\text{mod-twopow } x n = (\text{if } n = 0 \text{ then } 0 \text{ else } x \text{ mod } 2 + 2 * \text{mod-twopow } (x \text{ div } 2) (n - 1))$

*<proof>*

**lift-definition** *float-up* ::  $\text{int} \Rightarrow \text{float} \Rightarrow \text{float}$  **is** *round-up* *<proof>*

**declare** *float-up.rep-eq*[simp]

**lemma** *round-up-correct*:  $\text{round-up } e \ f - f \in \{0..2 \text{ powr } -e\}$   
 ⟨proof⟩

**lemma** *float-up-correct*:  $\text{real-of-float } (\text{float-up } e \ f) - \text{real-of-float } f \in \{0..2 \text{ powr } -e\}$   
 ⟨proof⟩

**lift-definition** *float-down* :: *int*  $\Rightarrow$  *float*  $\Rightarrow$  *float* **is** *round-down* ⟨proof⟩  
**declare** *float-down.rep-eq*[simp]

**lemma** *round-down-correct*:  $f - (\text{round-down } e \ f) \in \{0..2 \text{ powr } -e\}$   
 ⟨proof⟩

**lemma** *float-down-correct*:  $\text{real-of-float } f - \text{real-of-float } (\text{float-down } e \ f) \in \{0..2 \text{ powr } -e\}$   
 ⟨proof⟩

**context**  
**begin**

**qualified lemma** *compute-float-down*[code]:

*float-down* *p* (*Float* *m* *e*) =  
 (if  $p + e < 0$  then *Float* (*div-two*pow *m* (*nat*  $-(p + e)$ )))  $(-p)$  else *Float* *m* *e*)  
 ⟨proof⟩

**lemma** *abs-round-down-le*:  $|f - (\text{round-down } e \ f)| \leq 2 \text{ powr } -e$   
 ⟨proof⟩

**lemma** *abs-round-up-le*:  $|f - (\text{round-up } e \ f)| \leq 2 \text{ powr } -e$   
 ⟨proof⟩

**lemma** *round-down-nonneg*:  $0 \leq s \Longrightarrow 0 \leq \text{round-down } p \ s$   
 ⟨proof⟩

**lemma** *ceil-divide-floor-conv*:

**assumes**  $b \neq 0$

**shows**  $\lceil \text{real-of-int } a / \text{real-of-int } b \rceil =$

$(\text{if } b \text{ dvd } a \text{ then } a \text{ div } b \text{ else } \lfloor \text{real-of-int } a / \text{real-of-int } b \rfloor + 1)$

⟨proof⟩ **lemma** *compute-float-up*[code]:  $\text{float-up } p \ x = - \text{float-down } p \ (-x)$   
 ⟨proof⟩

**end**

**lemma** *bitlen-Float*:

**fixes** *m* *e*

**defines** [*THEN meta-eq-to-obj-eq*]:  $f \equiv \text{Float } m \ e$

**shows**  $\text{bitlen } |mantissa\ f| + \text{exponent } f = (\text{if } m = 0 \text{ then } 0 \text{ else } \text{bitlen } |m| + e)$   
 ⟨proof⟩

**lemma** *float-gt1-scale*:

**assumes**  $1 \leq \text{Float } m\ e$

**shows**  $0 \leq e + (\text{bitlen } m - 1)$

⟨proof⟩

### 43.9 Truncating Real Numbers

**definition** *truncate-down::nat  $\Rightarrow$  real  $\Rightarrow$  real*

**where**  $\text{truncate-down } prec\ x = \text{round-down } (prec - \lfloor \log 2 |x| \rfloor)\ x$

**lemma** *truncate-down*:  $\text{truncate-down } prec\ x \leq x$

⟨proof⟩

**lemma** *truncate-down-le*:  $x \leq y \implies \text{truncate-down } prec\ x \leq y$

⟨proof⟩

**lemma** *truncate-down-zero[simp]*:  $\text{truncate-down } prec\ 0 = 0$

⟨proof⟩

**lemma** *truncate-down-float[simp]*:  $\text{truncate-down } p\ x \in \text{float}$

⟨proof⟩

**definition** *truncate-up::nat  $\Rightarrow$  real  $\Rightarrow$  real*

**where**  $\text{truncate-up } prec\ x = \text{round-up } (prec - \lfloor \log 2 |x| \rfloor)\ x$

**lemma** *truncate-up*:  $x \leq \text{truncate-up } prec\ x$

⟨proof⟩

**lemma** *truncate-up-le*:  $x \leq y \implies x \leq \text{truncate-up } prec\ y$

⟨proof⟩

**lemma** *truncate-up-zero[simp]*:  $\text{truncate-up } prec\ 0 = 0$

⟨proof⟩

**lemma** *truncate-up-uminus-eq*:  $\text{truncate-up } prec\ (-x) = - \text{truncate-down } prec\ x$

**and** *truncate-down-uminus-eq*:  $\text{truncate-down } prec\ (-x) = - \text{truncate-up } prec\ x$

⟨proof⟩

**lemma** *truncate-up-float[simp]*:  $\text{truncate-up } p\ x \in \text{float}$

⟨proof⟩

**lemma** *mult-powr-eq*:  $0 < b \implies b \neq 1 \implies 0 < x \implies x * b \text{ powr } y = b \text{ powr } (y + \log b\ x)$

⟨proof⟩

**lemma** *truncate-down-pos*:

**assumes**  $x > 0$   
**shows**  $\text{truncate-down } p \ x > 0$   
 ⟨*proof*⟩

**lemma** *truncate-down-nonneg*:  $0 \leq y \implies 0 \leq \text{truncate-down } p \ y$   
 ⟨*proof*⟩

**lemma** *truncate-down-ge1*:  $1 \leq x \implies 1 \leq \text{truncate-down } p \ x$   
 ⟨*proof*⟩

**lemma** *truncate-up-nonpos*:  $x \leq 0 \implies \text{truncate-up } p \ x \leq 0$   
 ⟨*proof*⟩

**lemma** *truncate-up-le1*:  
**assumes**  $x \leq 1$   
**shows**  $\text{truncate-up } p \ x \leq 1$   
 ⟨*proof*⟩

**lemma** *truncate-down-shift-int*:  
 $\text{truncate-down } p \ (x * 2 \text{ powr } \text{real-of-int } k) = \text{truncate-down } p \ x * 2 \text{ powr } k$   
 ⟨*proof*⟩

**lemma** *truncate-down-shift-nat*:  $\text{truncate-down } p \ (x * 2 \text{ powr } \text{real } k) = \text{truncate-down } p \ x * 2 \text{ powr } k$   
 ⟨*proof*⟩

**lemma** *truncate-up-shift-int*:  $\text{truncate-up } p \ (x * 2 \text{ powr } \text{real-of-int } k) = \text{truncate-up } p \ x * 2 \text{ powr } k$   
 ⟨*proof*⟩

**lemma** *truncate-up-shift-nat*:  $\text{truncate-up } p \ (x * 2 \text{ powr } \text{real } k) = \text{truncate-up } p \ x * 2 \text{ powr } k$   
 ⟨*proof*⟩

### 43.10 Truncating Floats

**lift-definition** *float-round-up* ::  $\text{nat} \Rightarrow \text{float} \Rightarrow \text{float}$  **is** *truncate-up*  
 ⟨*proof*⟩

**lemma** *float-round-up*:  $\text{real-of-float } x \leq \text{real-of-float } (\text{float-round-up } p \ x)$   
 ⟨*proof*⟩

**lemma** *float-round-up-zero[simp]*:  $\text{float-round-up } p \ 0 = 0$   
 ⟨*proof*⟩

**lift-definition** *float-round-down* ::  $\text{nat} \Rightarrow \text{float} \Rightarrow \text{float}$  **is** *truncate-down*  
 ⟨*proof*⟩

**lemma** *float-round-down*:  $\text{real-of-float } (\text{float-round-down } p \ x) \leq \text{real-of-float } x$

*<proof>*

**lemma** *float-round-down-zero[simp]: float-round-down prec 0 = 0*  
*<proof>*

**lemmas** *float-round-up-le = order-trans[OF - float-round-up]*  
**and** *float-round-down-le = order-trans[OF float-round-down]*

**lemma** *minus-float-round-up-eq: - float-round-up prec x = float-round-down prec*  
*(- x)*  
**and** *minus-float-round-down-eq: - float-round-down prec x = float-round-up prec*  
*(- x)*  
*<proof>*

**context**  
**begin**

**qualified lemma** *compute-float-round-down[code]:*  
*float-round-down prec (Float m e) =*  
*(let d = bitlen |m| - int prec - 1 in*  
*if 0 < d then Float (div-twopow m (nat d)) (e + d)*  
*else Float m e)*  
*<proof>* **lemma** *compute-float-round-up[code]:*  
*float-round-up prec x = - float-round-down prec (-x)*  
*<proof>*

**end**

**lemma** *truncate-up-nonneg-mono:*  
**assumes**  $0 \leq x \ x \leq y$   
**shows** *truncate-up prec x ≤ truncate-up prec y*  
*<proof>*

**lemma** *truncate-up-switch-sign-mono:*  
**assumes**  $x \leq 0 \ 0 \leq y$   
**shows** *truncate-up prec x ≤ truncate-up prec y*  
*<proof>*

**lemma** *truncate-down-switch-sign-mono:*  
**assumes**  $x \leq 0$   
**and**  $0 \leq y$   
**and**  $x \leq y$   
**shows** *truncate-down prec x ≤ truncate-down prec y*  
*<proof>*

**lemma** *truncate-down-nonneg-mono:*  
**assumes**  $0 \leq x \ x \leq y$   
**shows** *truncate-down prec x ≤ truncate-down prec y*  
*<proof>*

**lemma** *truncate-down-eq-truncate-up*:  $\text{truncate-down } p \ x = - \text{truncate-up } p \ (-x)$   
**and** *truncate-up-eq-truncate-down*:  $\text{truncate-up } p \ x = - \text{truncate-down } p \ (-x)$   
 ⟨proof⟩

**lemma** *truncate-down-mono*:  $x \leq y \implies \text{truncate-down } p \ x \leq \text{truncate-down } p \ y$   
 ⟨proof⟩

**lemma** *truncate-up-mono*:  $x \leq y \implies \text{truncate-up } p \ x \leq \text{truncate-up } p \ y$   
 ⟨proof⟩

**lemma** *truncate-up-nonneg*:  $0 \leq \text{truncate-up } p \ x$  **if**  $0 \leq x$   
 ⟨proof⟩

**lemma** *truncate-up-pos*:  $0 < \text{truncate-up } p \ x$  **if**  $0 < x$   
 ⟨proof⟩

**lemma** *truncate-up-less-zero-iff[simp]*:  $\text{truncate-up } p \ x < 0 \longleftrightarrow x < 0$   
 ⟨proof⟩

**lemma** *truncate-up-nonneg-iff[simp]*:  $\text{truncate-up } p \ x \geq 0 \longleftrightarrow x \geq 0$   
 ⟨proof⟩

**lemma** *truncate-down-less-zero-iff[simp]*:  $\text{truncate-down } p \ x < 0 \longleftrightarrow x < 0$   
 ⟨proof⟩

**lemma** *truncate-down-nonneg-iff[simp]*:  $\text{truncate-down } p \ x \geq 0 \longleftrightarrow x \geq 0$   
 ⟨proof⟩

**lemma** *truncate-down-eq-zero-iff[simp]*:  $\text{truncate-down } p \ x = 0 \longleftrightarrow x = 0$   
 ⟨proof⟩

**lemma** *truncate-up-eq-zero-iff[simp]*:  $\text{truncate-up } p \ x = 0 \longleftrightarrow x = 0$   
 ⟨proof⟩

### 43.11 Approximation of positive rationals

**lemma** *div-mult-twopow-eq*:  $a \ \text{div} \ ((2::\text{nat}) \wedge^n) \ \text{div} \ b = a \ \text{div} \ (b * 2 \wedge^n)$  **for**  $a \ b :: \text{nat}$   
 ⟨proof⟩

**lemma** *real-div-nat-eq-floor-of-divide*:  $a \ \text{div} \ b = \text{real-of-int} \ [a / b]$  **for**  $a \ b :: \text{nat}$   
 ⟨proof⟩

**definition** *rat-precision*  $\text{prec } x \ y =$   
 (let  $d = \text{bitlen } x - \text{bitlen } y$   
 in  $\text{int } \text{prec} - d + (\text{if } \text{Float } (\text{abs } x) \ 0 < \text{Float } (\text{abs } y) \ d \text{ then } 1 \text{ else } 0)$ )

**lemma** *floor-log-divide-eq*:

**assumes**  $i > 0 \ j > 0 \ p > 1$   
**shows**  $\lfloor \log p (i / j) \rfloor = \text{floor} (\log p i) - \text{floor} (\log p j) -$   
 (if  $i \geq j * p^{\text{power} (\text{floor} (\log p i) - \text{floor} (\log p j))}$  then 0 else 1)  
 ⟨proof⟩

**lemma** *truncate-down-rat-precision*:

*truncate-down prec (real x / real y) = round-down (rat-precision prec x y) (real x / real y)*

**and** *truncate-up-rat-precision*:

*truncate-up prec (real x / real y) = round-up (rat-precision prec x y) (real x / real y)*

⟨proof⟩

**lift-definition** *lapprox-posrat* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{float}$

**is**  $\lambda \text{prec} (x::\text{nat}) (y::\text{nat}). \text{truncate-down prec} (x / y)$

⟨proof⟩

**context**

**begin**

**qualified lemma** *compute-lapprox-posrat*[code]:

*lapprox-posrat prec x y =*

(let

$l = \text{rat-precision prec} x y;$

$d = \text{if } 0 \leq l \text{ then } x * 2^{\text{nat } l} \text{ div } y \text{ else } x \text{ div } 2^{\text{nat } (- l)} \text{ div } y$

*in normfloat (Float d (- l))*)

⟨proof⟩

**end**

**lift-definition** *rapprox-posrat* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{float}$

**is**  $\lambda \text{prec} (x::\text{nat}) (y::\text{nat}). \text{truncate-up prec} (x / y)$

⟨proof⟩

**context**

**begin**

**qualified lemma** *compute-rapprox-posrat*[code]:

**fixes** *prec x y*

**defines**  $l \equiv \text{rat-precision prec} x y$

**shows** *rapprox-posrat prec x y =*

(let

$l = l;$

$(r, s) = \text{if } 0 \leq l \text{ then } (x * 2^{\text{nat } l}, y) \text{ else } (x, y * 2^{\text{nat } (- l)});$

$d = r \text{ div } s;$

$m = r \text{ mod } s$

*in normfloat (Float (d + (if  $m = 0 \vee y = 0$  then 0 else 1)) (- l))*)

⟨proof⟩

**end**

**lemma** *rat-precision-pos*:

assumes  $0 \leq x$   
 and  $0 < y$   
 and  $2 * x < y$   
 shows *rat-precision*  $n$  (*int*  $x$ ) (*int*  $y$ )  $> 0$   
 ⟨*proof*⟩

**lemma** *rapprox-posrat-less1*:

$0 \leq x \implies 0 < y \implies 2 * x < y \implies \text{real-of-float } (\text{rapprox-posrat } n \ x \ y) < 1$   
 ⟨*proof*⟩

**lift-definition** *lapprox-rat* :: *nat*  $\Rightarrow$  *int*  $\Rightarrow$  *int*  $\Rightarrow$  *float* **is**

$\lambda \text{prec } (x::\text{int}) \ (y::\text{int}). \text{truncate-down } \text{prec } (x / y)$   
 ⟨*proof*⟩

**context**

**begin**

**qualified lemma** *compute-lapprox-rat*[*code*]:

*lapprox-rat* *prec*  $x$   $y$  =  
 (if  $y = 0$  then  $0$   
 else if  $0 \leq x$  then  
 (if  $0 < y$  then *lapprox-posrat* *prec* (*nat*  $x$ ) (*nat*  $y$ )  
 else  $-$  (*rapprox-posrat* *prec* (*nat*  $x$ ) (*nat*  $(-y)$ )))  
 else  
 (if  $0 < y$   
 then  $-$  (*rapprox-posrat* *prec* (*nat*  $(-x)$ ) (*nat*  $y$ ))  
 else *lapprox-posrat* *prec* (*nat*  $(-x)$ ) (*nat*  $(-y)$ )))  
 ⟨*proof*⟩

**lift-definition** *rapprox-rat* :: *nat*  $\Rightarrow$  *int*  $\Rightarrow$  *int*  $\Rightarrow$  *float* **is**

$\lambda \text{prec } (x::\text{int}) \ (y::\text{int}). \text{truncate-up } \text{prec } (x / y)$   
 ⟨*proof*⟩

**lemma** *rapprox-rat* = *rapprox-posrat*

⟨*proof*⟩

**lemma** *lapprox-rat* = *lapprox-posrat*

⟨*proof*⟩ **lemma** *compute-rapprox-rat*[*code*]:

*rapprox-rat* *prec*  $x$   $y$  =  $-$  *lapprox-rat* *prec*  $(-x)$   $y$

⟨*proof*⟩ **lemma** *compute-truncate-down*[*code*]:

*truncate-down*  $p$  (*Ratreal*  $r$ ) = (let ( $a, b$ ) = *quotient-of*  $r$  in *lapprox-rat*  $p$   $a$   $b$ )

⟨*proof*⟩ **lemma** *compute-truncate-up*[*code*]:

*truncate-up*  $p$  (*Ratreal*  $r$ ) = (let ( $a, b$ ) = *quotient-of*  $r$  in *rapprox-rat*  $p$   $a$   $b$ )

⟨*proof*⟩

**end**



### 43.12 Division

**definition** *real-divl* *prec a b = truncate-down prec (a / b)*

**definition** *real-divr* *prec a b = truncate-up prec (a / b)*

**lift-definition** *float-divl* *:: nat ⇒ float ⇒ float ⇒ float is real-divl*  
 ⟨*proof*⟩

**context**  
**begin**

**qualified lemma** *compute-float-divl*[*code*]:  
*float-divl prec (Float m1 s1) (Float m2 s2) = lapprox-rat prec m1 m2 \* Float 1*  
*(s1 - s2)*  
 ⟨*proof*⟩

**lift-definition** *float-divr* *:: nat ⇒ float ⇒ float ⇒ float is real-divr*  
 ⟨*proof*⟩ **lemma** *compute-float-divr*[*code*]:  
*float-divr prec x y = - float-divl prec (-x) y*  
 ⟨*proof*⟩

**end**

### 43.13 Approximate Addition

**definition** *plus-down* *prec x y = truncate-down prec (x + y)*

**definition** *plus-up* *prec x y = truncate-up prec (x + y)*

**lemma** *float-plus-down-float*[*intro, simp*]: *x ∈ float ⇒ y ∈ float ⇒ plus-down p*  
*x y ∈ float*  
 ⟨*proof*⟩

**lemma** *float-plus-up-float*[*intro, simp*]: *x ∈ float ⇒ y ∈ float ⇒ plus-up p*  
*x y ∈ float*  
 ⟨*proof*⟩

**lift-definition** *float-plus-down* *:: nat ⇒ float ⇒ float ⇒ float is plus-down* ⟨*proof*⟩

**lift-definition** *float-plus-up* *:: nat ⇒ float ⇒ float ⇒ float is plus-up* ⟨*proof*⟩

**lemma** *plus-down*: *plus-down prec x y ≤ x + y*  
**and** *plus-up*: *x + y ≤ plus-up prec x y*  
 ⟨*proof*⟩

**lemma** *float-plus-down*: *real-of-float (float-plus-down prec x y) ≤ x + y*  
**and** *float-plus-up*: *x + y ≤ real-of-float (float-plus-up prec x y)*  
 ⟨*proof*⟩

**lemmas** *plus-down-le* = *order-trans*[*OF plus-down*]  
**and** *plus-up-le* = *order-trans*[*OF - plus-up*]  
**and** *float-plus-down-le* = *order-trans*[*OF float-plus-down*]  
**and** *float-plus-up-le* = *order-trans*[*OF - float-plus-up*]

**lemma** *compute-plus-up*[*code*]: *plus-up* *p* *x* *y* = - *plus-down* *p* (-*x*) (-*y*)  
 ⟨*proof*⟩

**lemma** *truncate-down-log2-eqI*:  
**assumes**  $\lfloor \log 2 |x| \rfloor = \lfloor \log 2 |y| \rfloor$   
**assumes**  $\lfloor x * 2^{\text{powr } (p - \lfloor \log 2 |x| \rfloor)} \rfloor = \lfloor y * 2^{\text{powr } (p - \lfloor \log 2 |x| \rfloor)} \rfloor$   
**shows** *truncate-down* *p* *x* = *truncate-down* *p* *y*  
 ⟨*proof*⟩

**lemma** *sum-neq-zeroI*:  
 $|a| \geq k \implies |b| < k \implies a + b \neq 0$   
 $|a| > k \implies |b| \leq k \implies a + b \neq 0$   
**for** *a* *k* :: *real*  
 ⟨*proof*⟩

**lemma** *abs-real-le-2-powr-bitlen*[*simp*]:  $|real-of-int\ m2| < 2^{\text{powr } real-of-int\ (bitlen\ |m2|)}$   
 ⟨*proof*⟩

**lemma** *floor-sum-times-2-powr-sgn-eq*:  
**fixes** *ai* *p* *q* :: *int*  
**and** *a* *b* :: *real*  
**assumes**  $a * 2^{\text{powr } p} = ai$   
**and** *b-le-1*:  $|b * 2^{\text{powr } (p + 1)}| \leq 1$   
**and** *leq*:  $q \leq p$   
**shows**  $\lfloor (a + b) * 2^{\text{powr } q} \rfloor = \lfloor (2 * ai + \text{sgn } b) * 2^{\text{powr } (q - p - 1)} \rfloor$   
 ⟨*proof*⟩

**lemma** *log2-abs-int-add-less-half-sgn-eq*:  
**fixes** *ai* :: *int*  
**and** *b* :: *real*  
**assumes**  $|b| \leq 1/2$   
**and**  $ai \neq 0$   
**shows**  $\lfloor \log 2 |real-of-int\ ai + b| \rfloor = \lfloor \log 2 |ai + \text{sgn } b / 2| \rfloor$   
 ⟨*proof*⟩

**context**  
**begin**

**qualified lemma** *compute-far-float-plus-down*:  
**fixes** *m1* *e1* *m2* *e2* :: *int*  
**and** *p* :: *nat*  
**defines**  $k1 \equiv \text{Suc } p - \text{nat } (bitlen\ |m1|)$   
**assumes** *H*:  $bitlen\ |m2| \leq e1 - e2 - k1 - 2\ m1 \neq 0\ m2 \neq 0\ e1 \geq e2$

**shows** *float-plus-down*  $p$  (*Float*  $m1$   $e1$ ) (*Float*  $m2$   $e2$ ) =  
*float-round-down*  $p$  (*Float* ( $m1 * 2^{\wedge}(\text{Suc}(\text{Suc } k1)) + \text{sgn } m2$ ) ( $e1 - \text{int } k1 - 2$ ))  
 ⟨*proof*⟩

**lemma** *compute-float-plus-down-naive*[code]: *float-plus-down*  $p$   $x$   $y$  = *float-round-down*  $p$  ( $x + y$ )

⟨*proof*⟩ **lemma** *compute-float-plus-down*[code]:

**fixes**  $p::\text{nat}$  **and**  $m1$   $e1$   $m2$   $e2::\text{int}$

**shows** *float-plus-down*  $p$  (*Float*  $m1$   $e1$ ) (*Float*  $m2$   $e2$ ) =

(if  $m1 = 0$  then *float-round-down*  $p$  (*Float*  $m2$   $e2$ )

else if  $m2 = 0$  then *float-round-down*  $p$  (*Float*  $m1$   $e1$ )

else

(if  $e1 \geq e2$  then

(let  $k1 = \text{Suc } p - \text{nat}(\text{bitlen } |m1|)$  in

if  $\text{bitlen } |m2| > e1 - e2 - k1 - 2$

then *float-round-down*  $p$  ((*Float*  $m1$   $e1$ ) + (*Float*  $m2$   $e2$ ))

else *float-round-down*  $p$  (*Float* ( $m1 * 2^{\wedge}(\text{Suc}(\text{Suc } k1)) + \text{sgn } m2$ ) ( $e1 - \text{int } k1 - 2$ )))

else *float-plus-down*  $p$  (*Float*  $m2$   $e2$ ) (*Float*  $m1$   $e1$ )))

⟨*proof*⟩ **lemma** *compute-float-plus-up*[code]: *float-plus-up*  $p$   $x$   $y$  = - *float-plus-down*  $p$  ( $-x$ ) ( $-y$ )

⟨*proof*⟩

**lemma** *mantissa-zero*: *mantissa*  $0 = 0$

⟨*proof*⟩ **lemma** *compute-float-less*[code]:  $a < b \longleftrightarrow \text{is-float-pos}(\text{float-plus-down } 0 \text{ } b \text{ } (-a))$

⟨*proof*⟩ **lemma** *compute-float-le*[code]:  $a \leq b \longleftrightarrow \text{is-float-nonneg}(\text{float-plus-down } 0 \text{ } b \text{ } (-a))$

⟨*proof*⟩

**end**

**lemma** *plus-down-mono*: *plus-down*  $p$   $a$   $b \leq \text{plus-down } p$   $c$   $d$  **if**  $a + b \leq c + d$

⟨*proof*⟩

**lemma** *plus-up-mono*: *plus-up*  $p$   $a$   $b \leq \text{plus-up } p$   $c$   $d$  **if**  $a + b \leq c + d$

⟨*proof*⟩

#### 43.14 Approximate Multiplication

**lemma** *mult-mono-nonpos-nonneg*:  $a * b \leq c * d$

**if**  $a \leq c$   $a \leq 0$   $0 \leq d$   $d \leq b$  **for**  $a$   $b$   $c$   $d::'a::\text{ordered-ring}$

⟨*proof*⟩

**lemma** *mult-mono-nonneg-nonpos*:  $b * a \leq d * c$

**if**  $a \leq c$   $c \leq 0$   $0 \leq d$   $d \leq b$  **for**  $a$   $b$   $c$   $d::'a::\text{ordered-ring}$

⟨*proof*⟩

**lemma** *mult-mono-nonpos-nonpos*:  $a * b \leq c * d$   
**if**  $a \geq c$   $a \leq 0$   $b \geq d$   $d \leq 0$  **for**  $a$   $b$   $c$   $d$ ::*real*  
 ⟨*proof*⟩

**lemma** *mult-float-mono1*:  
**shows**  $a \leq b \implies ab \leq bb \implies$   
 $aa \leq a \implies$   
 $b \leq ba \implies$   
 $ac \leq ab \implies$   
 $bb \leq bc \implies$   
 $\text{plus-down prec (nprt aa * pprrt bc)}$   
 $(\text{plus-down prec (nprt ba * nprt bc)})$   
 $(\text{plus-down prec (pprrt aa * pprrt ac)})$   
 $(\text{pprrt ba * nprt ac}))$   
 $\leq \text{plus-down prec (nprt a * pprrt bb)}$   
 $(\text{plus-down prec (nprt b * nprt bb)})$   
 $(\text{plus-down prec (pprrt a * pprrt ab)})$   
 $(\text{pprrt b * nprt ab}))$   
 ⟨*proof*⟩

**lemma** *mult-float-mono2*:  
**shows**  $a \leq b \implies$   
 $ab \leq bb \implies$   
 $aa \leq a \implies$   
 $b \leq ba \implies$   
 $ac \leq ab \implies$   
 $bb \leq bc \implies$   
 $\text{plus-up prec (pprrt b * pprrt bb)}$   
 $(\text{plus-up prec (pprrt a * nprt bb)})$   
 $(\text{plus-up prec (nprt b * pprrt ab)})$   
 $(\text{nprt a * nprt ab}))$   
 $\leq \text{plus-up prec (pprrt ba * pprrt bc)}$   
 $(\text{plus-up prec (pprrt aa * nprt bc)})$   
 $(\text{plus-up prec (nprt ba * pprrt ac)})$   
 $(\text{nprt aa * nprt ac}))$   
 ⟨*proof*⟩

### 43.15 Approximate Power

**lemma** *div2-less-self[termination-simp]*:  $\text{odd } n \implies n \text{ div } 2 < n$  **for**  $n$  :: *nat*  
 ⟨*proof*⟩

**fun** *power-down* :: *nat*  $\Rightarrow$  *real*  $\Rightarrow$  *nat*  $\Rightarrow$  *real*

**where**

$\text{power-down } p \ x \ 0 = 1$   
 $|\ \text{power-down } p \ x \ (\text{Suc } n) =$   
 $(\text{if odd } n \text{ then truncate-down (Suc } p) ((\text{power-down } p \ x \ (\text{Suc } n \text{ div } 2))^2)$   
 $\text{else truncate-down (Suc } p) (x * \text{power-down } p \ x \ n))$

**fun** *power-up* :: *nat*  $\Rightarrow$  *real*  $\Rightarrow$  *nat*  $\Rightarrow$  *real*  
**where**

*power-up* *p* *x* 0 = 1  
| *power-up* *p* *x* (*Suc* *n*) =  
    (if odd *n* then *truncate-up* *p* ((*power-up* *p* *x* (*Suc* *n* *div* 2))<sup>2</sup>)  
    else *truncate-up* *p* (*x* \* *power-up* *p* *x* *n*))

**lift-definition** *power-up-fl* :: *nat*  $\Rightarrow$  *float*  $\Rightarrow$  *nat*  $\Rightarrow$  *float* **is** *power-up*  
⟨*proof*⟩

**lift-definition** *power-down-fl* :: *nat*  $\Rightarrow$  *float*  $\Rightarrow$  *nat*  $\Rightarrow$  *float* **is** *power-down*  
⟨*proof*⟩

**lemma** *power-float-transfer*[*transfer-rule*]:  
(*rel-fun* *pcr-float* (*rel-fun* (=) *pcr-float*)) ( $\curvearrowright$ ) ( $\curvearrowright$ )  
⟨*proof*⟩

**lemma** *compute-power-up-fl*[*code*]:

*power-up-fl* *p* *x* 0 = 1  
*power-up-fl* *p* *x* (*Suc* *n*) =  
    (if odd *n* then *float-round-up* *p* ((*power-up-fl* *p* *x* (*Suc* *n* *div* 2))<sup>2</sup>)  
    else *float-round-up* *p* (*x* \* *power-up-fl* *p* *x* *n*))

**and** *compute-power-down-fl*[*code*]:

*power-down-fl* *p* *x* 0 = 1  
*power-down-fl* *p* *x* (*Suc* *n*) =  
    (if odd *n* then *float-round-down* (*Suc* *p*) ((*power-down-fl* *p* *x* (*Suc* *n* *div* 2))<sup>2</sup>)  
    else *float-round-down* (*Suc* *p*) (*x* \* *power-down-fl* *p* *x* *n*))

⟨*proof*⟩

**lemma** *power-down-pos*: 0 < *x*  $\implies$  0 < *power-down* *p* *x* *n*  
⟨*proof*⟩

**lemma** *power-down-nonneg*: 0  $\leq$  *x*  $\implies$  0  $\leq$  *power-down* *p* *x* *n*  
⟨*proof*⟩

**lemma** *power-down*: 0  $\leq$  *x*  $\implies$  *power-down* *p* *x* *n*  $\leq$  *x* <sup>*n*</sup>  
⟨*proof*⟩

**lemma** *power-up*: 0  $\leq$  *x*  $\implies$  *x* <sup>*n*</sup>  $\leq$  *power-up* *p* *x* *n*  
⟨*proof*⟩

**lemmas** *power-up-le* = *order-trans*[*OF* - *power-up*]  
**and** *power-up-less* = *less-le-trans*[*OF* - *power-up*]  
**and** *power-down-le* = *order-trans*[*OF* *power-down*]

**lemma** *power-down-fl*: 0  $\leq$  *x*  $\implies$  *power-down-fl* *p* *x* *n*  $\leq$  *x* <sup>*n*</sup>  
⟨*proof*⟩

**lemma** *power-up-fl*: 0  $\leq$  *x*  $\implies$  *x* <sup>*n*</sup>  $\leq$  *power-up-fl* *p* *x* *n*

*<proof>*

**lemma** *real-power-up-fl: real-of-float (power-up-fl p x n) = power-up p x n*  
*<proof>*

**lemma** *real-power-down-fl: real-of-float (power-down-fl p x n) = power-down p x n*  
*<proof>*

**lemmas** [*simp del*] = *power-down.simps(2) power-up.simps(2)*

**lemmas** *power-down-simp = power-down.simps(2)*

**lemmas** *power-up-simp = power-up.simps(2)*

**lemma** *power-down-even-nonneg: even n  $\implies$  0  $\leq$  power-down p x n*  
*<proof>*

**lemma** *power-down-eq-zero-iff[simp]: power-down prec b n = 0  $\longleftrightarrow$  b = 0  $\wedge$  n  $\neq$  0*  
*<proof>*

**lemma** *power-down-nonneg-iff[simp]:*  
*power-down prec b n  $\geq$  0  $\longleftrightarrow$  even n  $\vee$  b  $\geq$  0*  
*<proof>*

**lemma** *power-down-neg-iff[simp]:*  
*power-down prec b n < 0  $\longleftrightarrow$*   
*b < 0  $\wedge$  odd n*  
*<proof>*

**lemma** *power-down-nonpos-iff[simp]:*  
**notes** [*simp del*] = *power-down-neg-iff power-down-eq-zero-iff*  
**shows** *power-down prec b n  $\leq$  0  $\longleftrightarrow$  b < 0  $\wedge$  odd n  $\vee$  b = 0  $\wedge$  n  $\neq$  0*  
*<proof>*

**lemma** *power-down-mono:*  
*power-down prec a n  $\leq$  power-down prec b n*  
**if** *((0  $\leq$  a  $\wedge$  a  $\leq$  b)  $\vee$  (odd n  $\wedge$  a  $\leq$  b)  $\vee$  (even n  $\wedge$  a  $\leq$  0  $\wedge$  b  $\leq$  a))*  
*<proof>*

**lemma** *power-up-even-nonneg: even n  $\implies$  0  $\leq$  power-up p x n*  
*<proof>*

**lemma** *power-up-eq-zero-iff[simp]: power-up prec b n = 0  $\longleftrightarrow$  b = 0  $\wedge$  n  $\neq$  0*  
*<proof>*

**lemma** *power-up-nonneg-iff[simp]:*  
*power-up prec b n  $\geq$  0  $\longleftrightarrow$  even n  $\vee$  b  $\geq$  0*  
*<proof>*

**lemma** *power-up-neg-iff*[simp]:  
 $power-up\ prec\ b\ n < 0 \iff b < 0 \wedge odd\ n$   
 ⟨proof⟩

**lemma** *power-up-nonpos-iff*[simp]:  
**notes** [simp del] = *power-up-neg-iff power-up-eq-zero-iff*  
**shows**  $power-up\ prec\ b\ n \leq 0 \iff b < 0 \wedge odd\ n \vee b = 0 \wedge n \neq 0$   
 ⟨proof⟩

**lemma** *power-up-mono*:  
 $power-up\ prec\ a\ n \leq power-up\ prec\ b\ n$   
**if**  $((0 \leq a \wedge a \leq b) \vee (odd\ n \wedge a \leq b) \vee (even\ n \wedge a \leq 0 \wedge b \leq a))$   
 ⟨proof⟩

### 43.16 Lemmas needed by Approximate

**lemma** *Float-num*[simp]:  
 $real-of-float\ (Float\ 1\ 0) = 1$   
 $real-of-float\ (Float\ 1\ 1) = 2$   
 $real-of-float\ (Float\ 1\ 2) = 4$   
 $real-of-float\ (Float\ 1\ (-1)) = 1/2$   
 $real-of-float\ (Float\ 1\ (-2)) = 1/4$   
 $real-of-float\ (Float\ 1\ (-3)) = 1/8$   
 $real-of-float\ (Float\ (-1)\ 0) = -1$   
 $real-of-float\ (Float\ (numeral\ n)\ 0) = numeral\ n$   
 $real-of-float\ (Float\ (-numeral\ n)\ 0) = -numeral\ n$   
 ⟨proof⟩

**lemma** *real-of-Float-int*[simp]:  $real-of-float\ (Float\ n\ 0) = real\ n$   
 ⟨proof⟩

**lemma** *float-zero*[simp]:  $real-of-float\ (Float\ 0\ e) = 0$   
 ⟨proof⟩

**lemma** *abs-div-2-less*:  $a \neq 0 \implies a \neq -1 \implies |(a::int)\ div\ 2| < |a|$   
 ⟨proof⟩

**lemma** *lapprox-rat*:  $real-of-float\ (lapprox-rat\ prec\ x\ y) \leq real-of-int\ x / real-of-int\ y$   
 ⟨proof⟩

**lemma** *mult-div-le*:  
**fixes**  $a\ b :: int$   
**assumes**  $b > 0$   
**shows**  $a \geq b * (a\ div\ b)$   
 ⟨proof⟩

**lemma** *lapprox-rat-nonneg*:

**assumes**  $0 \leq x$  **and**  $0 \leq y$   
**shows**  $0 \leq \text{real-of-float } (\text{lapprox-rat } n \ x \ y)$   
 ⟨proof⟩

**lemma** *rapprox-rat*:  $\text{real-of-int } x / \text{real-of-int } y \leq \text{real-of-float } (\text{rapprox-rat } \text{prec } x \ y)$   
 ⟨proof⟩

**lemma** *rapprox-rat-le1*:  
**assumes**  $0 \leq x$   $0 < y$   $x \leq y$   
**shows**  $\text{real-of-float } (\text{rapprox-rat } n \ x \ y) \leq 1$   
 ⟨proof⟩

**lemma** *rapprox-rat-nonneg-nonpos*:  $0 \leq x \implies y \leq 0 \implies \text{real-of-float } (\text{rapprox-rat } n \ x \ y) \leq 0$   
 ⟨proof⟩

**lemma** *rapprox-rat-nonpos-nonneg*:  $x \leq 0 \implies 0 \leq y \implies \text{real-of-float } (\text{rapprox-rat } n \ x \ y) \leq 0$   
 ⟨proof⟩

**lemma** *real-divl*:  $\text{real-divl } \text{prec } x \ y \leq x / y$   
 ⟨proof⟩

**lemma** *real-divr*:  $x / y \leq \text{real-divr } \text{prec } x \ y$   
 ⟨proof⟩

**lemma** *float-divl*:  $\text{real-of-float } (\text{float-divl } \text{prec } x \ y) \leq x / y$   
 ⟨proof⟩

**lemma** *real-divl-lower-bound*:  $0 \leq x \implies 0 \leq y \implies 0 \leq \text{real-divl } \text{prec } x \ y$   
 ⟨proof⟩

**lemma** *float-divl-lower-bound*:  $0 \leq x \implies 0 \leq y \implies 0 \leq \text{real-of-float } (\text{float-divl } \text{prec } x \ y)$   
 ⟨proof⟩

**lemma** *exponent-1*:  $\text{exponent } 1 = 0$   
 ⟨proof⟩

**lemma** *mantissa-1*:  $\text{mantissa } 1 = 1$   
 ⟨proof⟩

**lemma** *bitlen-1*:  $\text{bitlen } 1 = 1$   
 ⟨proof⟩

**lemma** *float-upper-bound*:  $x \leq 2^{\text{powr } (\text{bitlen } |\text{mantissa } x| + \text{exponent } x)}$   
 ⟨proof⟩



**lemma** *real-divl-pos-less1-bound*:

**assumes**  $0 < x \leq 1$

**shows**  $1 \leq \text{real-divl prec } 1 \ x$

*<proof>*

**lemma** *float-divl-pos-less1-bound*:

$0 < \text{real-of-float } x \implies \text{real-of-float } x \leq 1 \implies \text{prec} \geq 1 \implies$

$1 \leq \text{real-of-float } (\text{float-divl prec } 1 \ x)$

*<proof>*

**lemma** *float-divr*:  $\text{real-of-float } x / \text{real-of-float } y \leq \text{real-of-float } (\text{float-divr prec } x \ y)$

*<proof>*

**lemma** *real-divr-pos-less1-lower-bound*:

**assumes**  $0 < x$

**and**  $x \leq 1$

**shows**  $1 \leq \text{real-divr prec } 1 \ x$

*<proof>*

**lemma** *float-divr-pos-less1-lower-bound*:  $0 < x \implies x \leq 1 \implies 1 \leq \text{float-divr prec } 1 \ x$

*<proof>*

**lemma** *real-divr-nonpos-pos-upper-bound*:  $x \leq 0 \implies 0 \leq y \implies \text{real-divr prec } x \ y \leq 0$

*<proof>*

**lemma** *float-divr-nonpos-pos-upper-bound*:

$\text{real-of-float } x \leq 0 \implies 0 \leq \text{real-of-float } y \implies \text{real-of-float } (\text{float-divr prec } x \ y) \leq$

$0$

*<proof>*

**lemma** *real-divr-nonneg-neg-upper-bound*:  $0 \leq x \implies y \leq 0 \implies \text{real-divr prec } x \ y \leq 0$

*<proof>*

**lemma** *float-divr-nonneg-neg-upper-bound*:

$0 \leq \text{real-of-float } x \implies \text{real-of-float } y \leq 0 \implies \text{real-of-float } (\text{float-divr prec } x \ y) \leq$

$0$

*<proof>*

**lemma** *Float-le-zero-iff*:  $\text{Float } a \ b \leq 0 \iff a \leq 0$

*<proof>*

**lemma** *real-of-float-pprt[simp]*:

**fixes**  $a :: \text{float}$

**shows**  $\text{real-of-float } (\text{pprt } a) = \text{pprt } (\text{real-of-float } a)$

*<proof>*

```

lemma real-of-float-nprt[simp]:
  fixes a :: float
  shows real-of-float (nprt a) = nprt (real-of-float a)
  ⟨proof⟩

context
begin

lift-definition int-floor-fl :: float ⇒ int is floor ⟨proof⟩ lemma compute-int-floor-fl[code]:
  int-floor-fl (Float m e) = (if 0 ≤ e then m * 2 ^ nat e else m div (2 ^ (nat
  (-e))))
  ⟨proof⟩

lift-definition floor-fl :: float ⇒ float is  $\lambda x.$  real-of-int  $\lfloor x \rfloor$ 
  ⟨proof⟩ lemma compute-floor-fl[code]:
  floor-fl (Float m e) = (if 0 ≤ e then Float m e else Float (m div (2 ^ (nat (-e))))
  0)
  ⟨proof⟩

end

lemma floor-fl: real-of-float (floor-fl x) ≤ real-of-float x
  ⟨proof⟩

lemma int-floor-fl: real-of-int (int-floor-fl x) ≤ real-of-float x
  ⟨proof⟩

lemma floor-pos-exp: exponent (floor-fl x) ≥ 0
  ⟨proof⟩

lemma compute-mantissa[code]:
  mantissa (Float m e) =
  (if m = 0 then 0 else if 2 dvd m then mantissa (normfloat (Float m e)) else m)
  ⟨proof⟩

lemma compute-exponent[code]:
  exponent (Float m e) =
  (if m = 0 then 0 else if 2 dvd m then exponent (normfloat (Float m e)) else e)
  ⟨proof⟩

lifting-update Float.float.lifting
lifting-forget Float.float.lifting

end

```

## 44 Pointwise instantiation of functions to algebra type classes

```

theory Function-Algebras
imports Main
begin

    Pointwise operations
instantiation fun :: (type, plus) plus
begin

definition  $f + g = (\lambda x. f\ x + g\ x)$ 
instance  $\langle proof \rangle$ 

end

lemma plus-fun-apply [simp]:
     $(f + g)\ x = f\ x + g\ x$ 
     $\langle proof \rangle$ 

instantiation fun :: (type, zero) zero
begin

definition  $0 = (\lambda x. 0)$ 
instance  $\langle proof \rangle$ 

end

lemma zero-fun-apply [simp]:
     $0\ x = 0$ 
     $\langle proof \rangle$ 

instantiation fun :: (type, times) times
begin

definition  $f * g = (\lambda x. f\ x * g\ x)$ 
instance  $\langle proof \rangle$ 

end

lemma times-fun-apply [simp]:
     $(f * g)\ x = f\ x * g\ x$ 
     $\langle proof \rangle$ 

instantiation fun :: (type, one) one
begin

definition  $1 = (\lambda x. 1)$ 
instance  $\langle proof \rangle$ 

```

**end**

**lemma** *one-fun-apply* [*simp*]:

$1\ x = 1$

*<proof>*

Additive structures

**instance** *fun* :: (*type*, *semigroup-add*) *semigroup-add*

*<proof>*

**instance** *fun* :: (*type*, *cancel-semigroup-add*) *cancel-semigroup-add*

*<proof>*

**instance** *fun* :: (*type*, *ab-semigroup-add*) *ab-semigroup-add*

*<proof>*

**instance** *fun* :: (*type*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*

*<proof>*

**instance** *fun* :: (*type*, *monoid-add*) *monoid-add*

*<proof>*

**instance** *fun* :: (*type*, *comm-monoid-add*) *comm-monoid-add*

*<proof>*

**instance** *fun* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add* *<proof>*

**instance** *fun* :: (*type*, *group-add*) *group-add*

*<proof>*

**instance** *fun* :: (*type*, *ab-group-add*) *ab-group-add*

*<proof>*

Multiplicative structures

**instance** *fun* :: (*type*, *semigroup-mult*) *semigroup-mult*

*<proof>*

**instance** *fun* :: (*type*, *ab-semigroup-mult*) *ab-semigroup-mult*

*<proof>*

**instance** *fun* :: (*type*, *monoid-mult*) *monoid-mult*

*<proof>*

**instance** *fun* :: (*type*, *comm-monoid-mult*) *comm-monoid-mult*

*<proof>*

Misc

**instance** *fun* :: (*type*, *Rings.dvd*) *Rings.dvd* *<proof>*

```

instance fun :: (type, mult-zero) mult-zero
  ⟨proof⟩

instance fun :: (type, zero-neq-one) zero-neq-one
  ⟨proof⟩

  Ring structures

instance fun :: (type, semiring) semiring
  ⟨proof⟩

instance fun :: (type, comm-semiring) comm-semiring
  ⟨proof⟩

instance fun :: (type, semiring-0) semiring-0 ⟨proof⟩

instance fun :: (type, comm-semiring-0) comm-semiring-0 ⟨proof⟩

instance fun :: (type, semiring-0-cancel) semiring-0-cancel ⟨proof⟩

instance fun :: (type, comm-semiring-0-cancel) comm-semiring-0-cancel ⟨proof⟩

instance fun :: (type, semiring-1) semiring-1 ⟨proof⟩

lemma numeral-fun:
  ⟨numeral n = (λx::'a. numeral n)⟩
  ⟨proof⟩

lemma numeral-fun-apply [simp]:
  ⟨numeral n x = numeral n⟩
  ⟨proof⟩

lemma of-nat-fun: of-nat n = (λx::'a. of-nat n)
  ⟨proof⟩

lemma of-nat-fun-apply [simp]:
  of-nat n x = of-nat n
  ⟨proof⟩

instance fun :: (type, comm-semiring-1) comm-semiring-1 ⟨proof⟩

instance fun :: (type, semiring-1-cancel) semiring-1-cancel ⟨proof⟩

instance fun :: (type, comm-semiring-1-cancel) comm-semiring-1-cancel
  ⟨proof⟩

instance fun :: (type, semiring-char-0) semiring-char-0
  ⟨proof⟩

```

```

instance fun :: (type, ring) ring <proof>

instance fun :: (type, comm-ring) comm-ring <proof>

instance fun :: (type, ring-1) ring-1 <proof>

instance fun :: (type, comm-ring-1) comm-ring-1 <proof>

instance fun :: (type, ring-char-0) ring-char-0 <proof>

    Ordered structures

instance fun :: (type, ordered-ab-semigroup-add) ordered-ab-semigroup-add
    <proof>

instance fun :: (type, ordered-cancel-ab-semigroup-add) ordered-cancel-ab-semigroup-add
    <proof>

instance fun :: (type, ordered-ab-semigroup-add-imp-le) ordered-ab-semigroup-add-imp-le
    <proof>

instance fun :: (type, ordered-comm-monoid-add) ordered-comm-monoid-add <proof>

instance fun :: (type, ordered-cancel-comm-monoid-add) ordered-cancel-comm-monoid-add
    <proof>

instance fun :: (type, ordered-ab-group-add) ordered-ab-group-add <proof>

instance fun :: (type, ordered-semiring) ordered-semiring
    <proof>

instance fun :: (type, dioid) dioid
    <proof>

instance fun :: (type, ordered-comm-semiring) ordered-comm-semiring
    <proof>

instance fun :: (type, ordered-cancel-semiring) ordered-cancel-semiring <proof>

instance fun :: (type, ordered-cancel-comm-semiring) ordered-cancel-comm-semiring
    <proof>

instance fun :: (type, ordered-ring) ordered-ring <proof>

instance fun :: (type, ordered-comm-ring) ordered-comm-ring <proof>

lemmas func-plus = plus-fun-def
lemmas func-zero = zero-fun-def
lemmas func-times = times-fun-def

```

```

lemmas func-one = one-fun-def

end

```

## 45 Pointwise instantiation of functions to division

```

theory Function-Division
imports Function-Algebras
begin

```

### 45.1 Syntactic with division

```

instantiation fun :: (type, inverse) inverse
begin

```

```

definition inverse f = inverse ◦ f

```

```

definition f div g = ( $\lambda x. f\ x / g\ x$ )

```

```

instance  $\langle$ proof $\rangle$ 

```

```

end

```

```

lemma inverse-fun-apply [simp]:
  inverse f x = inverse (f x)
   $\langle$ proof $\rangle$ 

```

```

lemma divide-fun-apply [simp]:
  (f / g) x = f x / g x
   $\langle$ proof $\rangle$ 

```

Unfortunately, we cannot lift these operations to algebraic type classes for division: being different from the constant zero function  $f \neq 0$  is too weak as precondition. So we must introduce our own set of lemmas.

```

abbreviation zero-free :: ( $'b \Rightarrow 'a::field$ )  $\Rightarrow$  bool where
  zero-free f  $\equiv \neg (\exists x. f\ x = 0)$ 

```

```

lemma fun-left-inverse:
  fixes f ::  $'b \Rightarrow 'a::field$ 
  shows zero-free f  $\Longrightarrow$  inverse f * f = 1
   $\langle$ proof $\rangle$ 

```

```

lemma fun-right-inverse:
  fixes f ::  $'b \Rightarrow 'a::field$ 
  shows zero-free f  $\Longrightarrow$  f * inverse f = 1
   $\langle$ proof $\rangle$ 

```

```

lemma fun-divide-inverse:
  fixes f g ::  $'b \Rightarrow 'a::field$ 

```

**shows**  $f / g = f * \text{inverse } g$   
 ⟨proof⟩

Feel free to extend this.

Another possibility would be a reformulation of the division type classes to use a *zero-free* predicate rather than a direct  $a \neq 0$  condition.

**end**

## 46 Lexicographic order on functions

**theory** *Fun-Lexorder*  
**imports** *Main*  
**begin**

**definition** *less-fun* :: (*'a::linorder*  $\Rightarrow$  *'b::linorder*)  $\Rightarrow$  (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  *bool*

**where**

*less-fun* *f g*  $\longleftrightarrow$  ( $\exists k. f\ k < g\ k \wedge (\forall k' < k. f\ k' = g\ k')$ )

**lemma** *less-funI*:

**assumes**  $\exists k. f\ k < g\ k \wedge (\forall k' < k. f\ k' = g\ k')$

**shows** *less-fun* *f g*

⟨proof⟩

**lemma** *less-funE*:

**assumes** *less-fun* *f g*

**obtains** *k* **where**  $f\ k < g\ k$  **and**  $\bigwedge k'. k' < k \implies f\ k' = g\ k'$

⟨proof⟩

**lemma** *less-fun-asym*:

**assumes** *less-fun* *f g*

**shows**  $\neg \text{less-fun } g\ f$

⟨proof⟩

**lemma** *less-fun-irrefl*:

$\neg \text{less-fun } f\ f$

⟨proof⟩

**lemma** *less-fun-trans*:

**assumes** *less-fun* *f g* **and** *less-fun* *g h*

**shows** *less-fun* *f h*

⟨proof⟩

**lemma** *order-less-fun*:

*class.order* ( $\lambda f\ g. \text{less-fun } f\ g \vee f = g$ ) *less-fun*

⟨proof⟩

**lemma** *less-fun-trichotomy*:

**assumes** *finite*  $\{k. f\ k \neq g\ k\}$



**shows**  $less\text{-fun } f g \vee f = g \vee less\text{-fun } g f$   
 ⟨proof⟩

**end**

## 47 The going-to filter

**theory** *Going-To-Filter*  
**imports** *Complex-Main*  
**begin**

**definition** *going-to-within* :: ('a ⇒ 'b) ⇒ 'b filter ⇒ 'a set ⇒ 'a filter  
 (⟨⟨open-block notation=⟨mixfix going-to-within⟩⟩(-)/ going'-to (-)/ within (-)⟩  
 [1000,60,60] 60)  
**where**  $f \text{ going-to } F \text{ within } A = inf (filtercomap f F) (principal A)$

**abbreviation** *going-to* :: ('a ⇒ 'b) ⇒ 'b filter ⇒ 'a filter  
 (infix ⟨going'-to⟩ 60)  
**where**  $f \text{ going-to } F \equiv f \text{ going-to } F \text{ within } UNIV$

The *going-to* filter is, in a sense, the opposite of *filtermap*. It corresponds to the intuition of, given a function  $f : A \rightarrow B$  and a filter  $F$  on the range of  $B$ , looking at such values of  $x$  that  $f(x)$  approaches  $F$ . This can be written as  $f \text{ going-to } F$ .

A classic example is the *at-infinity* filter, which describes the neighbourhood of infinity (i. e. all values sufficiently far away from the zero). This can also be written as *norm going-to at-top*.

Additionally, the *going-to* filter can be restricted with an optional ‘within’ parameter. For instance, if one would want to consider the filter of complex numbers near infinity that do not lie on the negative real line, one could write *cmod going-to at-top within - complex-of-real ‘{..0}*.

A third, less mathematical example lies in the complexity analysis of algorithms. Suppose we wanted to say that an algorithm on lists takes  $O(n^2)$  time where  $n$  is the length of the input list. We can write this using the Landau symbols from the AFP, where the underlying filter is *length going-to sequentially*. If, on the other hand, we want to look the complexity of the algorithm on sorted lists, we could use the filter *length going-to sequentially within Collect sorted*.

**lemma** *going-to-def*:  $f \text{ going-to } F = filtercomap f F$   
 ⟨proof⟩

**lemma** *eventually-going-toI* [intro]:  
**assumes** *eventually P F*  
**shows** *eventually* ( $\lambda x. P (f x)$ ) ( $f \text{ going-to } F$ )  
 ⟨proof⟩

**lemma** *filterlim-going-toI-weak* [intro]:  $filterlim f F (f \text{ going-to } F \text{ within } A)$

*<proof>*

**lemma** *going-to-mono*:  $F \leq G \implies A \subseteq B \implies f \text{ going-to } F \text{ within } A \leq f \text{ going-to } G \text{ within } B$

*<proof>*

**lemma** *going-to-inf*:

$f \text{ going-to } (\inf F G) \text{ within } A = \inf (f \text{ going-to } F \text{ within } A) (f \text{ going-to } G \text{ within } A)$

*<proof>*

**lemma** *going-to-sup*:

$f \text{ going-to } (\sup F G) \text{ within } A \geq \sup (f \text{ going-to } F \text{ within } A) (f \text{ going-to } G \text{ within } A)$

*<proof>*

**lemma** *going-to-top [simp]*:  $f \text{ going-to top within } A = \text{principal } A$

*<proof>*

**lemma** *going-to-bot [simp]*:  $f \text{ going-to bot within } A = \text{bot}$

*<proof>*

**lemma** *going-to-principal*:

$f \text{ going-to principal } A \text{ within } B = \text{principal } (f \text{ - ' } A \cap B)$

*<proof>*

**lemma** *going-to-within-empty [simp]*:  $f \text{ going-to } F \text{ within } \{\} = \text{bot}$

*<proof>*

**lemma** *going-to-within-union [simp]*:

$f \text{ going-to } F \text{ within } (A \cup B) = \sup (f \text{ going-to } F \text{ within } A) (f \text{ going-to } F \text{ within } B)$

*<proof>*

**lemma** *eventually-going-to-at-top-linorder*:

**fixes**  $f :: 'a \Rightarrow 'b :: \text{linorder}$

**shows**  $\text{eventually } P (f \text{ going-to at-top within } A) \iff (\exists C. \forall x \in A. f x \geq C \longrightarrow P x)$

*<proof>*

**lemma** *eventually-going-to-at-bot-linorder*:

**fixes**  $f :: 'a \Rightarrow 'b :: \text{linorder}$

**shows**  $\text{eventually } P (f \text{ going-to at-bot within } A) \iff (\exists C. \forall x \in A. f x \leq C \longrightarrow P x)$

*<proof>*

**lemma** *eventually-going-to-at-top-dense*:

**fixes**  $f :: 'a \Rightarrow 'b :: \{\text{linorder, no-top}\}$

**shows**  $\text{eventually } P (f \text{ going-to at-top within } A) \iff (\exists C. \forall x \in A. f x > C \longrightarrow P x)$

$P x$   
 ⟨proof⟩

**lemma** *eventually-going-to-at-bot-dense:*

**fixes**  $f :: 'a \Rightarrow 'b :: \{\text{linorder, no-bot}\}$

**shows** *eventually*  $P$  (*f going-to at-bot within A*)  $\longleftrightarrow$   $(\exists C. \forall x \in A. f x < C \longrightarrow P x)$

⟨proof⟩

**lemma** *eventually-going-to-nhds:*

*eventually*  $P$  (*f going-to nhds a within A*)  $\longleftrightarrow$

$(\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in A. f x \in S \longrightarrow P x))$

⟨proof⟩

**lemma** *eventually-going-to-at:*

*eventually*  $P$  (*f going-to (at a within B) within A*)  $\longleftrightarrow$

$(\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in A. f x \in B \cap S - \{a\} \longrightarrow P x))$

⟨proof⟩

**lemma** *norm-going-to-at-top-eq: norm going-to at-top = at-infinity*

⟨proof⟩

**lemmas** *at-infinity-altdef = norm-going-to-at-top-eq [symmetric]*

end

## 48 Big sum and product over function bodies

**theory** *Groups-Big-Fun*

**imports**

*Main*

**begin**

### 48.1 Abstract product

**locale** *comm-monoid-fun = comm-monoid*

**begin**

**definition**  $G :: ('b \Rightarrow 'a) \Rightarrow 'a$

**where**

*expand-set*:  $G g = \text{comm-monoid-set.F } f \mathbf{1} g \{a. g a \neq \mathbf{1}\}$

**interpretation**  $F: \text{comm-monoid-set } f \mathbf{1}$

⟨proof⟩

**lemma** *expand-superset:*

**assumes** *finite A and*  $\{a. g a \neq \mathbf{1}\} \subseteq A$

**shows**  $G g = F.F g A$

⟨proof⟩

**lemma** *conditionalize*:

**assumes** *finite A*

**shows**  $F.F\ g\ A = G\ (\lambda a. \text{if } a \in A \text{ then } g\ a \text{ else } \mathbf{1})$

*<proof>*

**lemma** *neutral [simp]*:

$G\ (\lambda a. \mathbf{1}) = \mathbf{1}$

*<proof>*

**lemma** *update [simp]*:

**assumes** *finite {a. g a ≠ 1}*

**assumes**  $g\ a = \mathbf{1}$

**shows**  $G\ (g(a := b)) = b * G\ g$

*<proof>*

**lemma** *infinite [simp]*:

$\neg \text{finite } \{a. g\ a \neq \mathbf{1}\} \implies G\ g = \mathbf{1}$

*<proof>*

**lemma** *cong [cong]*:

**assumes**  $\bigwedge a. g\ a = h\ a$

**shows**  $G\ g = G\ h$

*<proof>*

**lemma** *not-neutral-obtains-not-neutral*:

**assumes**  $G\ g \neq \mathbf{1}$

**obtains** *a where g a ≠ 1*

*<proof>*

**lemma** *reindex-cong*:

**assumes** *bij l*

**assumes**  $g \circ l = h$

**shows**  $G\ g = G\ h$

*<proof>*

**lemma** *distrib*:

**assumes** *finite {a. g a ≠ 1} and finite {a. h a ≠ 1}*

**shows**  $G\ (\lambda a. g\ a * h\ a) = G\ g * G\ h$

*<proof>*

**lemma** *swap*:

**assumes** *finite C*

**assumes** *subset: {a. ∃ b. g a b ≠ 1} × {b. ∃ a. g a b ≠ 1} ⊆ C (is ?A × ?B ⊆ C)*

**shows**  $G\ (\lambda a. G\ (g\ a)) = G\ (\lambda b. G\ (\lambda a. g\ a\ b))$

*<proof>*

**lemma cartesian-product:**

**assumes** *finite C*

**assumes** *subset: {a.  $\exists b. g a b \neq \mathbf{1}$ }  $\times$  {b.  $\exists a. g a b \neq \mathbf{1}$ }  $\subseteq C$  (is ?A  $\times$  ?B  $\subseteq C$ )*

**shows**  $G (\lambda a. G (g a)) = G (\lambda(a, b). g a b)$

*<proof>*

**lemma cartesian-product2:**

**assumes** *fin: finite D*

**assumes** *subset: {(a, b).  $\exists c. g a b c \neq \mathbf{1}$ }  $\times$  {c.  $\exists a b. g a b c \neq \mathbf{1}$ }  $\subseteq D$  (is ?AB  $\times$  ?C  $\subseteq D$ )*

**shows**  $G (\lambda(a, b). G (g a b)) = G (\lambda(a, b, c). g a b c)$

*<proof>*

**lemma delta [simp]:**

$G (\lambda b. \text{if } b = a \text{ then } g b \text{ else } \mathbf{1}) = g a$

*<proof>*

**lemma delta' [simp]:**

$G (\lambda b. \text{if } a = b \text{ then } g b \text{ else } \mathbf{1}) = g a$

*<proof>*

**end**

## 48.2 Concrete sum

**context** *comm-monoid-add*

**begin**

**sublocale** *Sum-any: comm-monoid-fun plus 0*

**rewrites** *comm-monoid-set.F plus 0 = sum*

**defines** *Sum-any = Sum-any.G*

*<proof>*

**end**

**syntax** (ASCII)

*-Sum-any :: ptrn  $\Rightarrow$  'a  $\Rightarrow$  'a::comm-monoid-add* (*<(<indent=3 notation=<binder SUM>>SUM -.)> [0, 10] 10*)

**syntax**

*-Sum-any :: ptrn  $\Rightarrow$  'a  $\Rightarrow$  'a::comm-monoid-add* (*<(<indent=2 notation=<binder  $\sum$ >> $\sum$  -.)> [0, 10] 10*)

**syntax-consts**

*-Sum-any  $\equiv$  Sum-any*

**translations**

$\sum a. b \equiv \text{CONST Sum-any } (\lambda a. b)$

**lemma Sum-any-left-distrib:**

**fixes** *r :: 'a :: semiring-0*

**assumes** *finite* {*a. g a ≠ 0*}  
**shows** *Sum-any g \* r = (∑ n. g n \* r)*  
 ⟨*proof*⟩

**lemma** *Sum-any-right-distrib*:  
**fixes** *r :: 'a :: semiring-0*  
**assumes** *finite* {*a. g a ≠ 0*}  
**shows** *r \* Sum-any g = (∑ n. r \* g n)*  
 ⟨*proof*⟩

**lemma** *Sum-any-product*:  
**fixes** *f g :: 'b ⇒ 'a::semiring-0*  
**assumes** *finite* {*a. f a ≠ 0*} **and** *finite* {*b. g b ≠ 0*}  
**shows** *Sum-any f \* Sum-any g = (∑ a. ∑ b. f a \* g b)*  
 ⟨*proof*⟩

**lemma** *Sum-any-eq-zero-iff [simp]*:  
**fixes** *f :: 'a ⇒ nat*  
**assumes** *finite* {*a. f a ≠ 0*}  
**shows** *Sum-any f = 0 ⟷ f = (λ-. 0)*  
 ⟨*proof*⟩

### 48.3 Concrete product

**context** *comm-monoid-mult*  
**begin**

**sublocale** *Prod-any: comm-monoid-fun times 1*  
**rewrites** *comm-monoid-set.F times 1 = prod*  
**defines** *Prod-any = Prod-any.G*  
 ⟨*proof*⟩

**end**

**syntax** (*ASCII*)

*-Prod-any :: pptrn ⇒ 'a ⇒ 'a::comm-monoid-mult* (⟨⟨*indent=4 notation=⟨binder*  
*PROD⟩⟩PROD -. -⟩ [0, 10] 10)*

**syntax**

*-Prod-any :: pptrn ⇒ 'a ⇒ 'a::comm-monoid-mult* (⟨⟨*indent=2 notation=⟨binder*  
*∏⟩⟩∏[-. -] [0, 10] 10)*

**syntax-consts**

*-Prod-any == Prod-any*

**translations**

*∏ a. b == CONST Prod-any (λa. b)*

**lemma** *Prod-any-zero*:

**fixes** *f :: 'b ⇒ 'a :: comm-semiring-1*  
**assumes** *finite* {*a. f a ≠ 1*}  
**assumes** *f a = 0*

**shows**  $(\prod a. f a) = 0$   
 $\langle proof \rangle$

**lemma** *Prod-any-not-zero*:  
**fixes**  $f :: 'b \Rightarrow 'a :: comm-semiring-1$   
**assumes**  $finite \{a. f a \neq 1\}$   
**assumes**  $(\prod a. f a) \neq 0$   
**shows**  $f a \neq 0$   
 $\langle proof \rangle$

**lemma** *power-Sum-any*:  
**assumes**  $finite \{a. f a \neq 0\}$   
**shows**  $c \wedge (\sum a. f a) = (\prod a. c \wedge f a)$   
 $\langle proof \rangle$

**end**

## 49 Infinite Type Class

The type class of infinite sets (originally from the Incredible Proof Machine)

**theory** *Infinite-Typeclass*  
**imports** *Complex-Main*  
**begin**

**class** *infinite* =  
**assumes** *infinite-UNIV*:  $infinite (UNIV :: 'a set)$

**begin**

**lemma** *arb-element*:  $finite Y \implies \exists x :: 'a. x \notin Y$   
 $\langle proof \rangle$

**lemma** *arb-finite-subset*:  $finite Y \implies \exists X :: 'a set. Y \cap X = \{\} \wedge finite X \wedge n \leq card X$   
 $\langle proof \rangle$

**lemma** *arb-countable-map*:  $finite Y \implies \exists f :: (nat \Rightarrow 'a). inj f \wedge range f \subseteq UNIV - Y$   
 $\langle proof \rangle$

**end**

**instance**  $nat :: infinite$   
 $\langle proof \rangle$

**instance**  $int :: infinite$   
 $\langle proof \rangle$

```

instance rat :: infinite
  ⟨proof⟩

instance real :: infinite
  ⟨proof⟩

instance complex :: infinite
  ⟨proof⟩

instance option :: (infinite) infinite
  ⟨proof⟩

instance prod :: (infinite, type) infinite
  ⟨proof⟩

instance list :: (type) infinite
  ⟨proof⟩

end

```

## 50 Algebraic operations on sets

```

theory Set-Algebras
  imports Main
begin

```

This library lifts operations like addition and multiplication to sets. It was designed to support asymptotic calculations for the now-obsolete BigO theory, but has other uses.

```

instantiation set :: (plus) plus
begin

```

```

definition plus-set :: 'a::plus set ⇒ 'a set ⇒ 'a set
  where set-plus-def:  $A + B = \{c. \exists a \in A. \exists b \in B. c = a + b\}$ 

```

```

instance ⟨proof⟩

```

```

end

```

```

instantiation set :: (times) times
begin

```

```

definition times-set :: 'a::times set ⇒ 'a set ⇒ 'a set
  where set-times-def:  $A * B = \{c. \exists a \in A. \exists b \in B. c = a * b\}$ 

```

```

instance ⟨proof⟩

```

```

end

```



```

instantiation set :: (zero) zero
begin

definition set-zero[simp]: (0::'a::zero set) = {0}

instance ⟨proof⟩

end

instantiation set :: (one) one
begin

definition set-one[simp]: (1::'a::one set) = {1}

instance ⟨proof⟩

end

definition elt-set-plus :: 'a::plus ⇒ 'a set ⇒ 'a set (infixl ⟨+o⟩ 70)
  where a +o B = {c. ∃ b∈B. c = a + b}

definition elt-set-times :: 'a::times ⇒ 'a set ⇒ 'a set (infixl ⟨*o⟩ 80)
  where a *o B = {c. ∃ b∈B. c = a * b}

abbreviation (input) elt-set-eq :: 'a ⇒ 'a set ⇒ bool (infix ⟨=o⟩ 50)
  where x =o A ≡ x ∈ A

instance set :: (semigroup-add) semigroup-add
  ⟨proof⟩

instance set :: (ab-semigroup-add) ab-semigroup-add
  ⟨proof⟩

instance set :: (monoid-add) monoid-add
  ⟨proof⟩

instance set :: (comm-monoid-add) comm-monoid-add
  ⟨proof⟩

instance set :: (semigroup-mult) semigroup-mult
  ⟨proof⟩

instance set :: (ab-semigroup-mult) ab-semigroup-mult
  ⟨proof⟩

instance set :: (monoid-mult) monoid-mult
  ⟨proof⟩

instance set :: (comm-monoid-mult) comm-monoid-mult

```

*<proof>*

**lemma** *sumset-empty* [*simp*]:  $A + \{\} = \{\} \{\} + A = \{\}$   
*<proof>*

**lemma** *Un-set-plus*:  $(A \cup B) + C = (A+C) \cup (B+C)$  **and** *set-plus-Un*:  $C + (A \cup B) = (C+A) \cup (C+B)$   
*<proof>*

**lemma**  
**fixes**  $A :: 'a::comm-monoid-add\ set$   
**shows** *insert-set-plus*:  $(insert\ a\ A) + B = (A+B) \cup (((+)\ a)\ 'B)$  **and** *set-plus-insert*:  
 $B + (insert\ a\ A) = (B+A) \cup (((+)\ a)\ 'B)$   
*<proof>*

**lemma** *set-add-0* [*simp*]:  
**fixes**  $A :: 'a::comm-monoid-add\ set$   
**shows**  $\{0\} + A = A$   
*<proof>*

**lemma** *set-add-0-right* [*simp*]:  
**fixes**  $A :: 'a::comm-monoid-add\ set$   
**shows**  $A + \{0\} = A$   
*<proof>*

**lemma** *card-plus-sing*:  
**fixes**  $A :: 'a::ab-group-add\ set$   
**shows**  $card\ (A + \{a\}) = card\ A$   
*<proof>*

**lemma** *set-plus-intro* [*intro*]:  $a \in C \implies b \in D \implies a + b \in C + D$   
*<proof>*

**lemma** *set-plus-elim*:  
**assumes**  $x \in A + B$   
**obtains**  $a\ b$  **where**  $x = a + b$  **and**  $a \in A$  **and**  $b \in B$   
*<proof>*

**lemma** *set-plus-intro2* [*intro*]:  $b \in C \implies a + b \in a + o\ C$   
*<proof>*

**lemma** *set-plus-rearrange*:  $(a + o\ C) + (b + o\ D) = (a + b) + o\ (C + D)$   
**for**  $a\ b :: 'a::comm-monoid-add$   
*<proof>*

**lemma** *set-plus-rearrange2*:  $a + o\ (b + o\ C) = (a + b) + o\ C$   
**for**  $a\ b :: 'a::semigroup-add$   
*<proof>*

**lemma** *set-plus-rearrange3*:  $(a +_o B) + C = a +_o (B + C)$   
**for**  $a :: 'a::\text{semigroup-add}$   
 ⟨*proof*⟩

**theorem** *set-plus-rearrange4*:  $C + (a +_o D) = a +_o (C + D)$   
**for**  $a :: 'a::\text{comm-monoid-add}$   
 ⟨*proof*⟩

**lemmas** *set-plus-rearranges = set-plus-rearrange set-plus-rearrange2 set-plus-rearrange3 set-plus-rearrange4*

**lemma** *set-plus-mono* [*intro!*]:  $C \subseteq D \implies a +_o C \subseteq a +_o D$   
 ⟨*proof*⟩

**lemma** *set-plus-mono2* [*intro*]:  $C \subseteq D \implies E \subseteq F \implies C + E \subseteq D + F$   
**for**  $C D E F :: 'a::\text{plus set}$   
 ⟨*proof*⟩

**lemma** *set-plus-mono3* [*intro*]:  $a \in C \implies a +_o D \subseteq C + D$   
 ⟨*proof*⟩

**lemma** *set-plus-mono4* [*intro*]:  $a \in C \implies a +_o D \subseteq D + C$   
**for**  $a :: 'a::\text{comm-monoid-add}$   
 ⟨*proof*⟩

**lemma** *set-plus-mono5*:  $a \in C \implies B \subseteq D \implies a +_o B \subseteq C + D$   
 ⟨*proof*⟩

**lemma** *set-plus-mono-b*:  $C \subseteq D \implies x \in a +_o C \implies x \in a +_o D$   
 ⟨*proof*⟩

**lemma** *set-zero-plus* [*simp*]:  $0 +_o C = C$   
**for**  $C :: 'a::\text{comm-monoid-add set}$   
 ⟨*proof*⟩

**lemma** *set-zero-plus2*:  $0 \in A \implies B \subseteq A + B$   
**for**  $A B :: 'a::\text{comm-monoid-add set}$   
 ⟨*proof*⟩

**lemma** *set-plus-imp-minus*:  $a \in b +_o C \implies a - b \in C$   
**for**  $a b :: 'a::\text{ab-group-add}$   
 ⟨*proof*⟩

**lemma** *set-minus-imp-plus*:  $a - b \in C \implies a \in b +_o C$   
**for**  $a b :: 'a::\text{ab-group-add}$   
 ⟨*proof*⟩

**lemma** *set-minus-plus*:  $a - b \in C \iff a \in b +_o C$   
**for**  $a b :: 'a::\text{ab-group-add}$

*<proof>*

**lemma** *set-times-intro* [intro]:  $a \in C \implies b \in D \implies a * b \in C * D$   
*<proof>*

**lemma** *set-times-elim*:  
**assumes**  $x \in A * B$   
**obtains**  $a b$  **where**  $x = a * b$  **and**  $a \in A$  **and**  $b \in B$   
*<proof>*

**lemma** *set-times-intro2* [intro!]:  $b \in C \implies a * b \in a * o C$   
*<proof>*

**lemma** *set-times-rearrange*:  $(a * o C) * (b * o D) = (a * b) * o (C * D)$   
**for**  $a b :: 'a::comm-monoid-mult$   
*<proof>*

**lemma** *set-times-rearrange2*:  $a * o (b * o C) = (a * b) * o C$   
**for**  $a b :: 'a::semigroup-mult$   
*<proof>*

**lemma** *set-times-rearrange3*:  $(a * o B) * C = a * o (B * C)$   
**for**  $a :: 'a::semigroup-mult$   
*<proof>*

**theorem** *set-times-rearrange4*:  $C * (a * o D) = a * o (C * D)$   
**for**  $a :: 'a::comm-monoid-mult$   
*<proof>*

**lemmas** *set-times-rearranges = set-times-rearrange set-times-rearrange2 set-times-rearrange3 set-times-rearrange4*

**lemma** *set-times-mono* [intro]:  $C \subseteq D \implies a * o C \subseteq a * o D$   
*<proof>*

**lemma** *set-times-mono2* [intro]:  $C \subseteq D \implies E \subseteq F \implies C * E \subseteq D * F$   
**for**  $C D E F :: 'a::times set$   
*<proof>*

**lemma** *set-times-mono3* [intro]:  $a \in C \implies a * o D \subseteq C * D$   
*<proof>*

**lemma** *set-times-mono4* [intro]:  $a \in C \implies a * o D \subseteq D * C$   
**for**  $a :: 'a::comm-monoid-mult$   
*<proof>*

**lemma** *set-times-mono5*:  $a \in C \implies B \subseteq D \implies a * o B \subseteq C * D$   
*<proof>*

**lemma** *set-one-times* [*simp*]:  $1 *o C = C$   
**for**  $C :: 'a::comm-monoid-mult\ set$   
*<proof>*

**lemma** *set-times-plus-distrib*:  $a *o (b +o C) = (a * b) +o (a *o C)$   
**for**  $a\ b :: 'a::semiring$   
*<proof>*

**lemma** *set-times-plus-distrib2*:  $a *o (B + C) = (a *o B) + (a *o C)$   
**for**  $a :: 'a::semiring$   
*<proof>*

**lemma** *set-times-plus-distrib3*:  $(a +o C) * D \subseteq a *o D + C * D$   
**for**  $a :: 'a::semiring$   
*<proof>*

**lemmas** *set-times-plus-distrib3* =  
*set-times-plus-distrib*  
*set-times-plus-distrib2*

**lemma** *set-neg-intro*:  $a \in (-\ 1) *o C \implies -\ a \in C$   
**for**  $a :: 'a::ring-1$   
*<proof>*

**lemma** *set-neg-intro2*:  $a \in C \implies -\ a \in (-\ 1) *o C$   
**for**  $a :: 'a::ring-1$   
*<proof>*

**lemma** *set-plus-image*:  $S + T = (\lambda(x, y). x + y) ` (S \times T)$   
*<proof>*

**lemma** *set-times-image*:  $S * T = (\lambda(x, y). x * y) ` (S \times T)$   
*<proof>*

**lemma** *finite-set-plus*:  $finite\ s \implies finite\ t \implies finite\ (s + t)$   
*<proof>*

**lemma** *finite-set-times*:  $finite\ s \implies finite\ t \implies finite\ (s * t)$   
*<proof>*

**lemma** *set-sum-alt*:  
**assumes**  $fin: finite\ I$   
**shows**  $sum\ S\ I = \{sum\ s\ I \mid s. \forall i \in I. s\ i \in S\ i\}$   
*(is - = ?sum I)*  
*<proof>*

**lemma** *sum-set-cond-linear*:  
**fixes**  $f :: 'a::comm-monoid-add\ set \Rightarrow 'b::comm-monoid-add\ set$   
**assumes** [*intro!*]:  $\bigwedge A\ B. P\ A \implies P\ B \implies P\ (A + B)\ P\ \{0\}$

**and**  $f: \bigwedge A B. P A \implies P B \implies f (A + B) = f A + f B$   $f \{0\} = \{0\}$   
**assumes**  $all: \bigwedge i. i \in I \implies P (S i)$   
**shows**  $f (sum S I) = sum (f \circ S) I$   
 $\langle proof \rangle$

**lemma** *sum-set-linear*:

**fixes**  $f :: 'a::comm-monoid-add set \Rightarrow 'b::comm-monoid-add set$   
**assumes**  $\bigwedge A B. f(A) + f(B) = f(A + B)$   $f \{0\} = \{0\}$   
**shows**  $f (sum S I) = sum (f \circ S) I$   
 $\langle proof \rangle$

**lemma** *set-times-Un-distrib*:

$A * (B \cup C) = A * B \cup A * C$   
 $(A \cup B) * C = A * C \cup B * C$   
 $\langle proof \rangle$

**lemma** *set-times-UNION-distrib*:

$A * \bigcup (M ' I) = (\bigcup i \in I. A * M i)$   
 $\bigcup (M ' I) * A = (\bigcup i \in I. M i * A)$   
 $\langle proof \rangle$

**end**

## 51 Interval Type

**theory** *Interval*

**imports**

*Complex-Main*  
*Lattice-Algebras*  
*Set-Algebras*

**begin**

A type of non-empty, closed intervals.

**typedef** (overloaded)  $'a$  *interval* =  
 $\{(a::'a::preorder, b). a \leq b\}$   
**morphisms** *bounds-of-interval Interval*  
 $\langle proof \rangle$

**setup-lifting** *type-definition-interval*

**lift-definition**  $lower::('a::preorder)$  *interval*  $\Rightarrow 'a$  **is** *fst*  $\langle proof \rangle$

**lift-definition**  $upper::('a::preorder)$  *interval*  $\Rightarrow 'a$  **is** *snd*  $\langle proof \rangle$

**lemma** *interval-eq-iff*:  $a = b \iff lower a = lower b \wedge upper a = upper b$   
 $\langle proof \rangle$

**lemma** *interval-eqI*:  $lower a = lower b \implies upper a = upper b \implies a = b$   
 $\langle proof \rangle$

**lemma** *lower-le-upper*[simp]: *lower i ≤ upper i*  
 ⟨*proof*⟩

**lift-definition** *set-of* :: *'a::preorder interval ⇒ 'a set* **is**  $\lambda x. \{fst\ x \ ..\ snd\ x\}$  ⟨*proof*⟩

**lemma** *set-of-eq*: *set-of x = {lower x .. upper x}*  
 ⟨*proof*⟩

**context notes** [[*typedef-overloaded*]] **begin**

**lift-definition**(*code-dt*) *Interval'::'a::preorder ⇒ 'a::preorder ⇒ 'a interval option*  
**is**  $\lambda a\ b. \text{if } a \leq b \text{ then } \text{Some } (a, b) \text{ else } \text{None}$   
 ⟨*proof*⟩

**lemma** *Interval'-split*:  
 $P (Interval' a b) \longleftrightarrow$   
 $(\forall ivl. a \leq b \longrightarrow lower\ ivl = a \longrightarrow upper\ ivl = b \longrightarrow P (Some\ ivl)) \wedge (\neg a \leq b$   
 $\longrightarrow P\ None)$   
 ⟨*proof*⟩

**lemma** *Interval'-split-asm*:  
 $P (Interval' a b) \longleftrightarrow$   
 $\neg((\exists ivl. a \leq b \wedge lower\ ivl = a \wedge upper\ ivl = b \wedge \neg P (Some\ ivl)) \vee (\neg a \leq b \wedge$   
 $\neg P\ None))$   
 ⟨*proof*⟩

**lemmas** *Interval'-splits = Interval'-split Interval'-split-asm*

**lemma** *Interval'-eq-Some*: *Interval' a b = Some i ⇒ lower i = a ∧ upper i = b*  
 ⟨*proof*⟩

**end**

**instantiation** *interval* :: (*{preorder, equal}*) *equal*  
**begin**

**definition** *equal-class.equal*  $a\ b \equiv (lower\ a = lower\ b) \wedge (upper\ a = upper\ b)$

**instance** ⟨*proof*⟩  
**end**

**instantiation** *interval* :: (*preorder*) *ord* **begin**

**definition** *less-eq-interval* :: *'a interval ⇒ 'a interval ⇒ bool*  
**where** *less-eq-interval*  $a\ b \longleftrightarrow lower\ b \leq lower\ a \wedge upper\ a \leq upper\ b$

**definition** *less-interval* :: *'a interval ⇒ 'a interval ⇒ bool*  
**where** *less-interval*  $x\ y = (x \leq y \wedge \neg y \leq x)$

**instance**  $\langle proof \rangle$   
**end**

**instantiation** *interval* :: (*lattice*) *semilattice-sup*  
**begin**

**lift-definition** *sup-interval* :: 'a *interval*  $\Rightarrow$  'a *interval*  $\Rightarrow$  'a *interval*  
**is**  $\lambda(a, b) (c, d). (\text{inf } a \ c, \text{sup } b \ d)$   
 $\langle proof \rangle$

**lemma** *lower-sup[simp]*:  $\text{lower } (\text{sup } A \ B) = \text{inf } (\text{lower } A) (\text{lower } B)$   
 $\langle proof \rangle$

**lemma** *upper-sup[simp]*:  $\text{upper } (\text{sup } A \ B) = \text{sup } (\text{upper } A) (\text{upper } B)$   
 $\langle proof \rangle$

**instance**  $\langle proof \rangle$   
**end**

**lemma** *set-of-interval-union*:  $\text{set-of } A \cup \text{set-of } B \subseteq \text{set-of } (\text{sup } A \ B)$  **for**  $A :: 'a :: \text{lattice interval}$   
 $\langle proof \rangle$

**lemma** *interval-union-commute*:  $\text{sup } A \ B = \text{sup } B \ A$  **for**  $A :: 'a :: \text{lattice interval}$   
 $\langle proof \rangle$

**lemma** *interval-union-mono1*:  $\text{set-of } a \subseteq \text{set-of } (\text{sup } a \ A)$  **for**  $A :: 'a :: \text{lattice interval}$   
 $\langle proof \rangle$

**lemma** *interval-union-mono2*:  $\text{set-of } A \subseteq \text{set-of } (\text{sup } a \ A)$  **for**  $A :: 'a :: \text{lattice interval}$   
 $\langle proof \rangle$

**lift-definition** *interval-of* :: 'a::preorder  $\Rightarrow$  'a *interval* **is**  $\lambda x. (x, x)$   
 $\langle proof \rangle$

**lemma** *lower-interval-of[simp]*:  $\text{lower } (\text{interval-of } a) = a$   
 $\langle proof \rangle$

**lemma** *upper-interval-of[simp]*:  $\text{upper } (\text{interval-of } a) = a$   
 $\langle proof \rangle$

**definition** *width* :: 'a::{preorder,minus} *interval*  $\Rightarrow$  'a  
**where**  $\text{width } i = \text{upper } i - \text{lower } i$

**instantiation** *interval* :: (*ordered-ab-semigroup-add*) *ab-semigroup-add*



**begin**

**lift-definition** *plus-interval*::'a interval  $\Rightarrow$  'a interval  $\Rightarrow$  'a interval  
**is**  $\lambda(a, b). \lambda(c, d). (a + c, b + d)$   
 $\langle proof \rangle$

**lemma** *lower-plus[simp]*: *lower* (plus A B) = plus (lower A) (lower B)  
 $\langle proof \rangle$

**lemma** *upper-plus[simp]*: *upper* (plus A B) = plus (upper A) (upper B)  
 $\langle proof \rangle$

**instance**  $\langle proof \rangle$   
**end**

**instance** *interval* :: ( $\{ordered-ab-semigroup-add, lattice\}$ ) *ordered-ab-semigroup-add*  
 $\langle proof \rangle$

**instantiation** *interval* :: ( $\{preorder, zero\}$ ) *zero*  
**begin**

**lift-definition** *zero-interval*::'a interval **is** (0, 0)  $\langle proof \rangle$

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

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

**instance**  $\langle proof \rangle$   
**end**

**instance** *interval* :: ( $\{ordered-comm-monoid-add\}$ ) *comm-monoid-add*  
 $\langle proof \rangle$

**instance** *interval* :: ( $\{ordered-comm-monoid-add, lattice\}$ ) *ordered-comm-monoid-add*  
 $\langle proof \rangle$

**instantiation** *interval* :: ( $\{ordered-ab-group-add\}$ ) *uminus*  
**begin**

**lift-definition** *uminus-interval*::'a interval  $\Rightarrow$  'a interval **is**  $\lambda(a, b). (-b, -a)$   
 $\langle proof \rangle$

**lemma** *lower-uminus[simp]*: *lower* (- A) = - upper A  
 $\langle proof \rangle$

**lemma** *upper-uminus[simp]*: *upper* (- A) = - lower A  
 $\langle proof \rangle$

**instance**  $\langle proof \rangle$   
**end**

**instantiation** *interval* :: ( $\{ordered-ab-group-add\}$ ) *minus*  
**begin**

**definition** *minus-interval*::'a interval  $\Rightarrow$  'a interval  $\Rightarrow$  'a interval

**where** *minus-interval*  $a\ b = a + -\ b$   
**lemma** *lower-minus[simp]*:  $\text{lower}(\text{minus } A\ B) = \text{minus}(\text{lower } A)\ (\text{upper } B)$   
 ⟨*proof*⟩  
**lemma** *upper-minus[simp]*:  $\text{upper}(\text{minus } A\ B) = \text{minus}(\text{upper } A)\ (\text{lower } B)$   
 ⟨*proof*⟩

**instance** ⟨*proof*⟩  
**end**

**instantiation** *interval* :: ( $\{\text{times}, \text{linorder}\}$ ) *times*  
**begin**

**lift-definition** *times-interval* :: 'a *interval*  $\Rightarrow$  'a *interval*  $\Rightarrow$  'a *interval*  
**is**  $\lambda(a1, a2). \lambda(b1, b2).$   
 (let  $x1 = a1 * b1; x2 = a1 * b2; x3 = a2 * b1; x4 = a2 * b2$   
 in  $(\text{min } x1\ (\text{min } x2\ (\text{min } x3\ x4)), \text{max } x1\ (\text{max } x2\ (\text{max } x3\ x4)))$ )  
 ⟨*proof*⟩

**lemma** *lower-times*:  
 $\text{lower}(\text{times } A\ B) = \text{Min} \{\text{lower } A * \text{lower } B, \text{lower } A * \text{upper } B, \text{upper } A * \text{lower } B, \text{upper } A * \text{upper } B\}$   
 ⟨*proof*⟩

**lemma** *upper-times*:  
 $\text{upper}(\text{times } A\ B) = \text{Max} \{\text{lower } A * \text{lower } B, \text{lower } A * \text{upper } B, \text{upper } A * \text{lower } B, \text{upper } A * \text{upper } B\}$   
 ⟨*proof*⟩

**instance** ⟨*proof*⟩  
**end**

**lemma** *interval-eq-set-of-iff*:  $X = Y \iff \text{set-of } X = \text{set-of } Y$  **for**  $X\ Y :: 'a :: \text{order interval}$   
 ⟨*proof*⟩

## 51.1 Membership

**abbreviation** (in *preorder*) *in-interval* ( $\langle\langle\text{notation}=\langle\text{infix } \in_i\rangle\rangle\text{-/ } \in_i\ \rangle$ ) [51, 51]  
 50)

**where** *in-interval*  $x\ X \equiv x \in \text{set-of } X$

**lemma** *in-interval-to-interval[intro!]*:  $a \in_i \text{interval-of } a$   
 ⟨*proof*⟩

**lemma** *plus-in-intervalI*:  
**fixes**  $x\ y :: 'a :: \text{ordered-ab-semigroup-add}$   
**shows**  $x \in_i X \implies y \in_i Y \implies x + y \in_i X + Y$   
 ⟨*proof*⟩

**lemma** *connected-set-of*[*intro, simp*]:

*connected (set-of X) for X::'a::linear-continuum-topology interval*

*<proof>*

**lemma** *ex-sum-in-interval-lemma*:  $\exists xa \in \{la .. ua\}. \exists xb \in \{lb .. ub\}. x = xa + xb$

**if**  $la \leq ua$   $lb \leq ub$   $la + lb \leq x$   $x \leq ua + ub$

$ua - la \leq ub - lb$

**for**  $la$   $b$   $c$   $d$ ::'a::linordered-ab-group-add

*<proof>*

**lemma** *ex-sum-in-interval*:  $\exists xa \geq la. xa \leq ua \wedge (\exists xb \geq lb. xb \leq ub \wedge x = xa + xb)$

**if**  $a$ :  $la \leq ua$  **and**  $b$ :  $lb \leq ub$  **and**  $x$ :  $la + lb \leq x$   $x \leq ua + ub$

**for**  $la$   $b$   $c$   $d$ ::'a::linordered-ab-group-add

*<proof>*

**lemma** *Icc-plus-Icc*:

$\{a .. b\} + \{c .. d\} = \{a + c .. b + d\}$

**if**  $a \leq b$   $c \leq d$

**for**  $a$   $b$   $c$   $d$ ::'a::linordered-ab-group-add

*<proof>*

**lemma** *set-of-plus*:

**fixes**  $A$  :: 'a::linordered-ab-group-add interval

**shows** *set-of* (A + B) = *set-of* A + *set-of* B

*<proof>*

**lemma** *plus-in-intervalE*:

**fixes**  $xy$  :: 'a :: linordered-ab-group-add

**assumes**  $xy \in_i X + Y$

**obtains**  $x$   $y$  **where**  $xy = x + y$   $x \in_i X$   $y \in_i Y$

*<proof>*

**lemma** *set-of-uminus*: *set-of* (-X) =  $\{-x \mid x. x \in \text{set-of } X\}$

**for**  $X$  :: 'a :: ordered-ab-group-add interval

*<proof>*

**lemma** *uminus-in-intervalI*:

**fixes**  $x$  :: 'a :: ordered-ab-group-add

**shows**  $x \in_i X \implies -x \in_i -X$

*<proof>*

**lemma** *uminus-in-intervalD*:

**fixes**  $x$  :: 'a :: ordered-ab-group-add

**shows**  $x \in_i -X \implies -x \in_i X$

*<proof>*

**lemma** *minus-in-intervalI*:

**fixes**  $x$   $y$  :: 'a :: ordered-ab-group-add

**shows**  $x \in_i X \implies y \in_i Y \implies x - y \in_i X - Y$   
 ⟨proof⟩

**lemma** *set-of-minus*:  $set-of (X - Y) = \{x - y \mid x \in set-of X \wedge y \in set-of Y\}$   
**for**  $X Y :: 'a :: linordered-ab-group-add interval$   
 ⟨proof⟩

**lemma** *times-in-intervalI*:  
**fixes**  $x y :: 'a :: linordered-ring$   
**assumes**  $x \in_i X \ y \in_i Y$   
**shows**  $x * y \in_i X * Y$   
 ⟨proof⟩

**lemma** *times-in-intervalE*:  
**fixes**  $xy :: 'a :: \{linorder, real-normed-algebra, linear-continuum-topology\}$   
 — TODO: linear continuum topology is pretty strong  
**assumes**  $xy \in_i X * Y$   
**obtains**  $x y$  **where**  $xy = x * y \ x \in_i X \ y \in_i Y$   
 ⟨proof⟩

**thm** *times-in-intervalE[of 1::real]*  
**lemma** *set-of-times*:  $set-of (X * Y) = \{x * y \mid x \in set-of X \wedge y \in set-of Y\}$   
**for**  $X Y :: 'a :: \{linordered-ring, real-normed-algebra, linear-continuum-topology\}$   
*interval*  
 ⟨proof⟩

**instance** *interval* ::  $(linordered-idom) cancel-semigroup-add$   
 ⟨proof⟩

**lemma** *interval-mul-commute*:  $A * B = B * A$  **for**  $A B :: 'a :: linordered-idom interval$   
 ⟨proof⟩

**lemma** *interval-times-zero-right[simp]*:  $A * 0 = 0$  **for**  $A :: 'a :: linordered-ring interval$   
 ⟨proof⟩

**lemma** *interval-times-zero-left[simp]*:  
 $0 * A = 0$  **for**  $A :: 'a :: linordered-ring interval$   
 ⟨proof⟩

**instantiation** *interval* ::  $(\{preorder, one\}) one$   
**begin**

**lift-definition** *one-interval*:: $'a interval$  **is**  $(1, 1)$  ⟨proof⟩

**lemma** *lower-one[simp]*:  $lower\ 1 = 1$   
 ⟨proof⟩

**lemma** *upper-one[simp]*:  $upper\ 1 = 1$   
 ⟨proof⟩

**instance**  $\langle proof \rangle$   
**end**

**instance** *interval* :: (*one*, *preorder*, *linorder*, *times*) *power*  
 $\langle proof \rangle$

**lemma** *set-of-one[simp]*: *set-of* ( $1 :: 'a :: \{one, order\}$  *interval*) =  $\{1\}$   
 $\langle proof \rangle$

**instance** *interval* ::  
(*linordered-idom*, *real-normed-algebra*, *linear-continuum-topology*) *monoid-mult*  
 $\langle proof \rangle$

**lemma** *one-times-ivl-left[simp]*:  $1 * A = A$  **for**  $A :: 'a :: linordered-idom$  *interval*  
 $\langle proof \rangle$

**lemma** *one-times-ivl-right[simp]*:  $A * 1 = A$  **for**  $A :: 'a :: linordered-idom$  *interval*  
 $\langle proof \rangle$

**lemma** *set-of-power-mono*:  $a \hat{\ } n \in set-of (A \hat{\ } n)$  **if**  $a \in set-of A$   
**for**  $a :: 'a :: linordered-idom$   
 $\langle proof \rangle$

**lemma** *set-of-add-cong*:  
 $set-of (A + B) = set-of (A' + B')$   
**if**  $set-of A = set-of A'$   $set-of B = set-of B'$   
**for**  $A :: 'a :: linordered-ab-group-add$  *interval*  
 $\langle proof \rangle$

**lemma** *set-of-add-inc-left*:  
 $set-of (A + B) \subseteq set-of (A' + B)$   
**if**  $set-of A \subseteq set-of A'$   
**for**  $A :: 'a :: linordered-ab-group-add$  *interval*  
 $\langle proof \rangle$

**lemma** *set-of-add-inc-right*:  
 $set-of (A + B) \subseteq set-of (A + B')$   
**if**  $set-of B \subseteq set-of B'$   
**for**  $A :: 'a :: linordered-ab-group-add$  *interval*  
 $\langle proof \rangle$

**lemma** *set-of-add-inc*:  
 $set-of (A + B) \subseteq set-of (A' + B')$   
**if**  $set-of A \subseteq set-of A'$   $set-of B \subseteq set-of B'$   
**for**  $A :: 'a :: linordered-ab-group-add$  *interval*  
 $\langle proof \rangle$

**lemma** *set-of-neg-inc*:  
 $set-of (-A) \subseteq set-of (-A')$

**if**  $\text{set-of } A \subseteq \text{set-of } A'$   
**for**  $A :: 'a::\text{ordered-ab-group-add interval}$   
 $\langle \text{proof} \rangle$

**lemma** *set-of-sub-inc-left*:  
 $\text{set-of } (A - B) \subseteq \text{set-of } (A' - B)$   
**if**  $\text{set-of } A \subseteq \text{set-of } A'$   
**for**  $A :: 'a::\text{linordered-ab-group-add interval}$   
 $\langle \text{proof} \rangle$

**lemma** *set-of-sub-inc-right*:  
 $\text{set-of } (A - B) \subseteq \text{set-of } (A - B')$   
**if**  $\text{set-of } B \subseteq \text{set-of } B'$   
**for**  $A :: 'a::\text{linordered-ab-group-add interval}$   
 $\langle \text{proof} \rangle$

**lemma** *set-of-sub-inc*:  
 $\text{set-of } (A - B) \subseteq \text{set-of } (A' - B')$   
**if**  $\text{set-of } A \subseteq \text{set-of } A'$   $\text{set-of } B \subseteq \text{set-of } B'$   
**for**  $A :: 'a::\text{linordered-idom interval}$   
 $\langle \text{proof} \rangle$

**lemma** *set-of-mul-inc-right*:  
 $\text{set-of } (A * B) \subseteq \text{set-of } (A * B')$   
**if**  $\text{set-of } B \subseteq \text{set-of } B'$   
**for**  $A :: 'a::\text{linordered-ring interval}$   
 $\langle \text{proof} \rangle$

**lemma** *set-of-distrib-left*:  
 $\text{set-of } (B * (A1 + A2)) \subseteq \text{set-of } (B * A1 + B * A2)$   
**for**  $A1 :: 'a::\text{linordered-ring interval}$   
 $\langle \text{proof} \rangle$

**lemma** *set-of-distrib-right*:  
 $\text{set-of } ((A1 + A2) * B) \subseteq \text{set-of } (A1 * B + A2 * B)$   
**for**  $A1 A2 B :: 'a::\{\text{linordered-ring, real-normed-algebra, linear-continuum-topology}\}$   
 $\text{interval}$   
 $\langle \text{proof} \rangle$

**lemma** *set-of-mul-inc-left*:  
 $\text{set-of } (A * B) \subseteq \text{set-of } (A' * B)$   
**if**  $\text{set-of } A \subseteq \text{set-of } A'$   
**for**  $A :: 'a::\{\text{linordered-ring, real-normed-algebra, linear-continuum-topology}\}$   $\text{interval}$   
 $\langle \text{proof} \rangle$

**lemma** *set-of-mul-inc*:  
 $\text{set-of } (A * B) \subseteq \text{set-of } (A' * B')$   
**if**  $\text{set-of } A \subseteq \text{set-of } A'$   $\text{set-of } B \subseteq \text{set-of } B'$

**for**  $A :: 'a::\{\text{linordered-ring, real-normed-algebra, linear-continuum-topology}\}$  *interval*  
 ⟨*proof*⟩

**lemma** *set-of-pow-inc:*

$\text{set-of } (A^{\wedge n}) \subseteq \text{set-of } (A'^{\wedge n})$   
**if**  $\text{set-of } A \subseteq \text{set-of } A'$   
**for**  $A :: 'a::\{\text{linordered-idom, real-normed-algebra, linear-continuum-topology}\}$  *interval*  
 ⟨*proof*⟩

**lemma** *set-of-distrib-right-left:*

$\text{set-of } ((A1 + A2) * (B1 + B2)) \subseteq \text{set-of } (A1 * B1 + A1 * B2 + A2 * B1 + A2 * B2)$   
**for**  $A1 :: 'a::\{\text{linordered-idom, real-normed-algebra, linear-continuum-topology}\}$  *interval*  
 ⟨*proof*⟩

**lemma** *mult-bounds-enclose-zero1:*

$\min (la * lb) (\min (la * ub) (\min (lb * ua) (ua * ub))) \leq 0$   
 $0 \leq \max (la * lb) (\max (la * ub) (\max (lb * ua) (ua * ub)))$   
**if**  $la \leq 0 \ 0 \leq ua$   
**for**  $la \ lb \ ua \ ub:: 'a::\text{linordered-idom}$   
 ⟨*proof*⟩

**lemma** *mult-bounds-enclose-zero2:*

$\min (la * lb) (\min (la * ub) (\min (lb * ua) (ua * ub))) \leq 0$   
 $0 \leq \max (la * lb) (\max (la * ub) (\max (lb * ua) (ua * ub)))$   
**if**  $lb \leq 0 \ 0 \leq ub$   
**for**  $la \ lb \ ua \ ub:: 'a::\text{linordered-idom}$   
 ⟨*proof*⟩

**lemma** *set-of-mul-contains-zero:*

$0 \in \text{set-of } (A * B)$   
**if**  $0 \in \text{set-of } A \vee 0 \in \text{set-of } B$   
**for**  $A :: 'a::\text{linordered-idom interval}$   
 ⟨*proof*⟩

**instance** *interval* ::  $(\{\text{linordered-semiring, zero, times}\})$  *mult-zero*  
 ⟨*proof*⟩

**lift-definition** *min-interval::'a::linorder interval*  $\Rightarrow 'a$  *interval*  $\Rightarrow 'a$  *interval* **is**  
 $\lambda(l1, u1). \lambda(l2, u2). (\min l1 \ l2, \min u1 \ u2)$   
 ⟨*proof*⟩

**lemma** *lower-min-interval[simp]:*  $\text{lower } (\text{min-interval } x \ y) = \min (\text{lower } x) (\text{lower } y)$   
 ⟨*proof*⟩

**lemma** *upper-min-interval[simp]:*  $\text{upper } (\text{min-interval } x \ y) = \min (\text{upper } x) (\text{upper } y)$

*<proof>*

**lemma** *min-intervalI*:

$a \in_i A \implies b \in_i B \implies \min a b \in_i \min\text{-interval } A B$

*<proof>*

**lift-definition** *max-interval*:: $'a::\text{linorder interval} \Rightarrow 'a \text{ interval} \Rightarrow 'a \text{ interval}$  **is**

$\lambda(l1, u1). \lambda(l2, u2). (\max l1 l2, \max u1 u2)$

*<proof>*

**lemma** *lower-max-interval[simp]*:  $\text{lower } (\max\text{-interval } x y) = \max (\text{lower } x) (\text{lower } y)$

*<proof>*

**lemma** *upper-max-interval[simp]*:  $\text{upper } (\max\text{-interval } x y) = \max (\text{upper } x) (\text{upper } y)$

*<proof>*

**lemma** *max-intervalI*:

$a \in_i A \implies b \in_i B \implies \max a b \in_i \max\text{-interval } A B$

*<proof>*

**lift-definition** *abs-interval*:: $'a::\text{linordered-idom interval} \Rightarrow 'a \text{ interval}$  **is**

$(\lambda(l,u). (\text{if } l < 0 \wedge 0 < u \text{ then } 0 \text{ else } \min |l| |u|, \max |l| |u|))$

*<proof>*

**lemma** *lower-abs-interval[simp]*:

$\text{lower } (\text{abs-interval } x) = (\text{if } \text{lower } x < 0 \wedge 0 < \text{upper } x \text{ then } 0 \text{ else } \min |\text{lower } x| |\text{upper } x|)$

*<proof>*

**lemma** *upper-abs-interval[simp]*:  $\text{upper } (\text{abs-interval } x) = \max |\text{lower } x| |\text{upper } x|$

*<proof>*

**lemma** *in-abs-intervalI1*:

$lx < 0 \implies 0 < ux \implies 0 \leq xa \implies xa \leq \max (-lx) (ux) \implies xa \in \text{abs } \{lx..ux\}$

**for**  $xa::'a::\text{linordered-idom}$

*<proof>*

**lemma** *in-abs-intervalI2*:

$\min (|lx|) |ux| \leq xa \implies xa \leq \max |lx| |ux| \implies lx \leq ux \implies 0 \leq lx \vee ux \leq 0$

$\implies$

$xa \in \text{abs } \{lx..ux\}$

**for**  $xa::'a::\text{linordered-idom}$

*<proof>*

**lemma** *set-of-abs-interval*:  $\text{set-of } (\text{abs-interval } x) = \text{abs } \{ \text{set-of } x$

*<proof>*

**fun** *split-domain* ::  $('a::\text{preorder interval} \Rightarrow 'a \text{ interval list}) \Rightarrow 'a \text{ interval list} \Rightarrow 'a \text{ interval list list}$

**where** *split-domain split* [] = [[]]



```
| split-domain split (I#Is) = (
  let S = split I;
      D = split-domain split Is
  in concat (map (λd. map (λs. s # d) S) D)
)
```

**context notes** `[[typedef-overloaded]] begin`

**lift-definition** `(code-dt) split-interval::'a::linorder interval ⇒ 'a ⇒ ('a interval × 'a interval)`

```
is λ(l, u) x. ((min l x, max l x), (min u x, max u x))
⟨proof⟩
```

**end**

**lemma** `split-domain-nonempty:`

```
assumes ∧I. split I ≠ []
shows split-domain split I ≠ []
⟨proof⟩
```

**lemma** `lower-split-interval1: lower (fst (split-interval X m)) = min (lower X) m`  
**and** `lower-split-interval2: lower (snd (split-interval X m)) = min (upper X) m`  
**and** `upper-split-interval1: upper (fst (split-interval X m)) = max (lower X) m`  
**and** `upper-split-interval2: upper (snd (split-interval X m)) = max (upper X) m`  
`⟨proof⟩`

**lemma** `split-intervalD: split-interval X x = (A, B) ⇒ set-of X ⊆ set-of A ∪ set-of B`  
`⟨proof⟩`

**instantiation** `interval :: ({topological-space, preorder}) topological-space`  
**begin**

**definition** `open-interval-def[code del]: open (X::'a interval set) =`

```
(∀ x∈X.
  ∃ A B.
    open A ∧
    open B ∧
    lower x ∈ A ∧ upper x ∈ B ∧ Interval ' (A × B) ⊆ X)
```

**instance**

```
⟨proof⟩
```

**end**

## 51.2 Quickcheck

**lift-definition** `Ivl::'a ⇒ 'a::preorder ⇒ 'a interval is λa b. (min a b, b)`

```
⟨proof⟩
```

**instantiation** `interval :: ({exhaustive,preorder}) exhaustive`

**begin**

**definition** *exhaustive-interval*::('a interval  $\Rightarrow$  (bool  $\times$  term list) option)  
 $\Rightarrow$  natural  $\Rightarrow$  (bool  $\times$  term list) option

**where**

*exhaustive-interval* f d =

*Quickcheck-Exhaustive.exhaustive* ( $\lambda x$ . *Quickcheck-Exhaustive.exhaustive* ( $\lambda y$ . f (Ivl x y)) d) d

**instance**  $\langle$ proof $\rangle$

**end**

**context**

**includes** *term-syntax*

**begin**

**definition** [*code-unfold*]:

*valtermify-interval* x y = *Code-Evaluation.valtermify* (Ivl::'a::{preorder,typerep} $\Rightarrow$ -) { $\cdot$ } x { $\cdot$ } y

**end**

**instantiation** *interval* :: ({*full-exhaustive.preorder,typerep*}) *full-exhaustive*

**begin**

**definition** *full-exhaustive-interval*::

('a interval  $\times$  (unit  $\Rightarrow$  term)  $\Rightarrow$  (bool  $\times$  term list) option)

$\Rightarrow$  natural  $\Rightarrow$  (bool  $\times$  term list) option **where**

*full-exhaustive-interval* f d =

*Quickcheck-Exhaustive.full-exhaustive*

( $\lambda x$ . *Quickcheck-Exhaustive.full-exhaustive* ( $\lambda y$ . f (*valtermify-interval* x y)) d)

d

**instance**  $\langle$ proof $\rangle$

**end**

**instantiation** *interval* :: ({*random.preorder,typerep*}) *random*

**begin**

**definition** *random-interval* ::

natural

$\Rightarrow$  natural  $\times$  natural

$\Rightarrow$  ('a interval  $\times$  (unit  $\Rightarrow$  term))  $\times$  natural  $\times$  natural **where**

*random-interval* i =

*scomp* (*Quickcheck-Random.random* i)

( $\lambda man$ . *scomp* (*Quickcheck-Random.random* i) ( $\lambda exp$ . *Pair* (*valtermify-interval* man exp))))

```

instance ⟨proof⟩

end

lifting-update interval.lifting
lifting-forget interval.lifting

end

```

## 52 Approximate Operations on Intervals of Floating Point Numbers

```

theory Interval-Float
  imports
    Interval
    Float
begin

```

```

definition mid :: float interval ⇒ float
  where mid i = (lower i + upper i) * Float 1 (-1)

```

```

lemma mid-in-interval: mid i ∈i i
  ⟨proof⟩

```

```

lemma mid-le: lower i ≤ mid i mid i ≤ upper i
  ⟨proof⟩

```

```

definition centered :: float interval ⇒ float interval
  where centered i = i - interval-of (mid i)

```

```

definition split-float-interval x = split-interval x ((lower x + upper x) * Float 1 (-1))

```

```

lemma split-float-intervalD: split-float-interval X = (A, B) ⇒ set-of X ⊆ set-of
A ∪ set-of B
  ⟨proof⟩

```

```

lemma split-float-interval-bounds:
  shows

```

```

  lower-split-float-interval1: lower (fst (split-float-interval X)) = lower X
  and lower-split-float-interval2: lower (snd (split-float-interval X)) = mid X
  and upper-split-float-interval1: upper (fst (split-float-interval X)) = mid X
  and upper-split-float-interval2: upper (snd (split-float-interval X)) = upper X
  ⟨proof⟩

```

```

lemmas float-round-down-le[intro] = order-trans[OF float-round-down]
  and float-round-up-ge[intro] = order-trans[OF - float-round-up]

```

TODO: many of the lemmas should move to theories Float or Approximation (the latter should be based on type *interval*).

## 52.1 Intervals with Floating Point Bounds

**context includes** *interval.lifting* **begin**

**lift-definition** *round-interval* :: *nat*  $\Rightarrow$  *float interval*  $\Rightarrow$  *float interval*  
**is**  $\lambda p. \lambda(l, u). (\text{float-round-down } p \ l, \text{float-round-up } p \ u)$   
*<proof>*

**lemma** *lower-round-ivl[simp]*: *lower* (*round-interval* *p* *x*) = *float-round-down* *p* (*lower* *x*)  
*<proof>*

**lemma** *upper-round-ivl[simp]*: *upper* (*round-interval* *p* *x*) = *float-round-up* *p* (*upper* *x*)  
*<proof>*

**lemma** *round-ivl-correct*: *set-of* *A*  $\subseteq$  *set-of* (*round-interval* *prec* *A*)  
*<proof>*

**lift-definition** *truncate-ivl* :: *nat*  $\Rightarrow$  *real interval*  $\Rightarrow$  *real interval*  
**is**  $\lambda p. \lambda(l, u). (\text{truncate-down } p \ l, \text{truncate-up } p \ u)$   
*<proof>*

**lemma** *lower-truncate-ivl[simp]*: *lower* (*truncate-ivl* *p* *x*) = *truncate-down* *p* (*lower* *x*)  
*<proof>*

**lemma** *upper-truncate-ivl[simp]*: *upper* (*truncate-ivl* *p* *x*) = *truncate-up* *p* (*upper* *x*)  
*<proof>*

**lemma** *truncate-ivl-correct*: *set-of* *A*  $\subseteq$  *set-of* (*truncate-ivl* *prec* *A*)  
*<proof>*

**lift-definition** *real-interval*::*float interval*  $\Rightarrow$  *real interval*  
**is**  $\lambda(l, u). (\text{real-of-float } l, \text{real-of-float } u)$   
*<proof>*

**lemma** *lower-real-interval[simp]*: *lower* (*real-interval* *x*) = *lower* *x*  
*<proof>*

**lemma** *upper-real-interval[simp]*: *upper* (*real-interval* *x*) = *upper* *x*  
*<proof>*

**definition** *set-of'* *x* = (*case* *x* of *None*  $\Rightarrow$  *UNIV* | *Some* *i*  $\Rightarrow$  *set-of* (*real-interval* *i*))

**lemma** *real-interval-min-interval[simp]*:  
*real-interval* (*min-interval* *a* *b*) = *min-interval* (*real-interval* *a*) (*real-interval* *b*)

*<proof>*

**lemma** *real-interval-max-interval[simp]*:

*real-interval (max-interval a b) = max-interval (real-interval a) (real-interval b)*  
*<proof>*

**lemma** *in-intervalI*:

*x ∈<sub>i</sub> X if lower X ≤ x x ≤ upper X*  
*<proof>*

**abbreviation** *in-real-interval* (*<<notation=<infix ∈<sub>r</sub>>>-/ ∈<sub>r</sub> ->*) [51, 51] 50)

**where** *x ∈<sub>r</sub> X ≡ x ∈<sub>i</sub> real-interval X*

**lemma** *in-real-intervalI*:

*x ∈<sub>r</sub> X if lower X ≤ x x ≤ upper X for x::real and X::float interval*  
*<proof>*

## 52.2 intros for *real-interval*

**lemma** *in-round-intervalI*: *x ∈<sub>r</sub> A ⇒ x ∈<sub>r</sub> (round-interval prec A)*

*<proof>*

**lemma** *zero-in-float-intervalI*: *0 ∈<sub>r</sub> 0*

*<proof>*

**lemma** *plus-in-float-intervalI*: *a + b ∈<sub>r</sub> A + B if a ∈<sub>r</sub> A b ∈<sub>r</sub> B*

*<proof>*

**lemma** *minus-in-float-intervalI*: *a - b ∈<sub>r</sub> A - B if a ∈<sub>r</sub> A b ∈<sub>r</sub> B*

*<proof>*

**lemma** *uminus-in-float-intervalI*: *-a ∈<sub>r</sub> -A if a ∈<sub>r</sub> A*

*<proof>*

**lemma** *real-interval-times*: *real-interval (A \* B) = real-interval A \* real-interval B*

*<proof>*

**lemma** *times-in-float-intervalI*: *a \* b ∈<sub>r</sub> A \* B if a ∈<sub>r</sub> A b ∈<sub>r</sub> B*

*<proof>*

**lemma** *real-interval-abs*: *real-interval (abs-interval A) = abs-interval (real-interval A)*

*<proof>*

**lemma** *abs-in-float-intervalI*: *abs a ∈<sub>r</sub> abs-interval A if a ∈<sub>r</sub> A*

*<proof>*

**lemma** *interval-of[intro,simp]*: *x ∈<sub>r</sub> interval-of x*

*<proof>*

**lemma** *split-float-interval-realD*:  $\text{split-float-interval } X = (A, B) \implies x \in_r X \implies x \in_r A \vee x \in_r B$   
*<proof>*

### 52.3 bounds for lists

**lemma** *lower-Interval*:  $\text{lower } (\text{Interval } x) = \text{fst } x$   
**and** *upper-Interval*:  $\text{upper } (\text{Interval } x) = \text{snd } x$   
**if**  $\text{fst } x \leq \text{snd } x$   
*<proof>*

**definition** *all-in-i* :: 'a::preorder list  $\Rightarrow$  'a interval list  $\Rightarrow$  bool  
*(infix <(all'-in<sub>i</sub>)> 50)*  
**where**  $x \text{ all-in}_i I = (\text{length } x = \text{length } I \wedge (\forall i < \text{length } I. x ! i \in_i I ! i))$

**definition** *all-in* :: real list  $\Rightarrow$  float interval list  $\Rightarrow$  bool  
*(infix <(all'-in)> 50)*  
**where**  $x \text{ all-in } I = (\text{length } x = \text{length } I \wedge (\forall i < \text{length } I. x ! i \in_r I ! i))$

**definition** *all-subset* :: 'a::order interval list  $\Rightarrow$  'a interval list  $\Rightarrow$  bool  
*(infix <(all'-subset)> 50)*  
**where**  $I \text{ all-subset } J = (\text{length } I = \text{length } J \wedge (\forall i < \text{length } I. \text{set-of } (I!i) \subseteq \text{set-of } (J!i)))$

**lemmas** [*simp*] = *all-in-def all-subset-def*

**lemma** *all-subsetD*:  
**assumes**  $I \text{ all-subset } J$   
**assumes**  $x \text{ all-in } I$   
**shows**  $x \text{ all-in } J$   
*<proof>*

**lemma** *round-interval-mono*:  $\text{set-of } (\text{round-interval prec } X) \subseteq \text{set-of } (\text{round-interval prec } Y)$   
**if**  $\text{set-of } X \subseteq \text{set-of } Y$   
*<proof>*

**lemma** *Ivl-simps*[*simp*]:  $\text{lower } (\text{Ivl } a \ b) = \min \ a \ b$   $\text{upper } (\text{Ivl } a \ b) = b$   
*<proof>*

**lemma** *set-of-subset-iff*:  $\text{set-of } X \subseteq \text{set-of } Y \iff \text{lower } Y \leq \text{lower } X \wedge \text{upper } X \leq \text{upper } Y$   
**for**  $X \ Y :: 'a :: \text{linorder interval}$   
*<proof>*

**lemma** *set-of-subset-iff'*:  
 $\text{set-of } a \subseteq \text{set-of } (b :: 'a :: \text{linorder interval}) \iff a \leq b$

*<proof>*

**lemma** *bounds-of-interval-eq-lower-upper:*

*bounds-of-interval*  $ivl = (lower\ ivl, upper\ ivl)$  **if**  $lower\ ivl \leq upper\ ivl$

*<proof>*

**lemma** *real-interval-Ivl:*  $real\ interval\ (Ivl\ a\ b) = Ivl\ a\ b$

*<proof>*

**lemma** *set-of-mul-contains-real-zero:*

$0 \in_r (A * B)$  **if**  $0 \in_r A \vee 0 \in_r B$

*<proof>*

**fun** *subdivide-interval* ::  $nat \Rightarrow float\ interval \Rightarrow float\ interval\ list$

**where** *subdivide-interval*  $0\ I = [I]$

| *subdivide-interval*  $(Suc\ n)\ I = ($

$let\ m = mid\ I$

$in\ (subdivide-interval\ n\ (Ivl\ (lower\ I)\ m))\ @\ (subdivide-interval\ n\ (Ivl\ m$

$(upper\ I)))$

)

**lemma** *subdivide-interval-length:*

**shows**  $length\ (subdivide-interval\ n\ I) = 2^{\wedge}n$

*<proof>*

**lemma** *lower-le-mid:*  $lower\ x \leq mid\ x$  *real-of-float*  $(lower\ x) \leq mid\ x$

**and** *mid-le-upper:*  $mid\ x \leq upper\ x$  *real-of-float*  $(mid\ x) \leq upper\ x$

*<proof>*

**lemma** *subdivide-interval-correct:*

*list-ex*  $(\lambda i. x \in_r i)$   $(subdivide-interval\ n\ I)$  **if**  $x \in_r I$  **for**  $x::real$

*<proof>*

**fun** *interval-list-union* ::  $'a::lattice\ interval\ list \Rightarrow 'a\ interval$

**where** *interval-list-union*  $[] = undefined$

| *interval-list-union*  $[I] = I$

| *interval-list-union*  $(I\#Is) = sup\ I\ (interval-list-union\ Is)$

**lemma** *interval-list-union-correct:*

**assumes**  $S \neq []$

**assumes**  $i < length\ S$

**shows**  $set-of\ (S!i) \subseteq set-of\ (interval-list-union\ S)$

*<proof>*

**lemma** *split-domain-correct:*

**fixes**  $x :: real\ list$

**assumes**  $x\ all-in\ I$

**assumes** *split-correct:*  $\bigwedge x\ a\ I. x \in_r I \implies list-ex\ (\lambda i::float\ interval. x \in_r i)$   $(split\ I)$

**shows** *list-ex* ( $\lambda s. x$  *all-in*  $s$ ) (*split-domain* *split*  $I$ )  
 ⟨*proof*⟩

**lift-definition** (*code-dt*) *inverse-float-interval::nat*  $\Rightarrow$  *float interval*  $\Rightarrow$  *float interval*  
*option* **is**  
 $\lambda prec (l, u). \text{if } (0 < l \vee u < 0) \text{ then } \text{Some } (\text{float-divl } prec\ 1\ u, \text{float-divr } prec\ 1\ l) \text{ else } \text{None}$   
 ⟨*proof*⟩

**lemma** *inverse-float-interval-eq-Some-conv*:  
**defines**  $one \equiv (1::\text{float})$

**shows**  
 $\text{inverse-float-interval } p\ X = \text{Some } R \iff$   
 $(\text{lower } X > 0 \vee \text{upper } X < 0) \wedge$   
 $\text{lower } R = \text{float-divl } p\ one\ (\text{upper } X) \wedge$   
 $\text{upper } R = \text{float-divr } p\ one\ (\text{lower } X)$   
 ⟨*proof*⟩

**lemma** *inverse-float-interval*:  
 $\text{inverse } ' \text{set-of } (\text{real-interval } X) \subseteq \text{set-of } (\text{real-interval } Y)$   
**if**  $\text{inverse-float-interval } p\ X = \text{Some } Y$   
 ⟨*proof*⟩

**lemma** *inverse-float-intervalI*:  
 $x \in_r X \implies \text{inverse } x \in \text{set-of}' (\text{inverse-float-interval } p\ X)$   
 ⟨*proof*⟩

**lemma** *inverse-float-interval-eqI*:  $\text{inverse-float-interval } p\ X = \text{Some } IVL \implies x \in_r X \implies \text{inverse } x \in_r IVL$   
 ⟨*proof*⟩

**lemma** *real-interval-abs-interval[simp]*:  
 $\text{real-interval } (\text{abs-interval } x) = \text{abs-interval } (\text{real-interval } x)$   
 ⟨*proof*⟩

**lift-definition** *floor-float-interval::float interval*  $\Rightarrow$  *float interval* **is**  
 $\lambda(l, u). (\text{floor-fl } l, \text{floor-fl } u)$   
 ⟨*proof*⟩

**lemma** *lower-floor-float-interval[simp]*:  $\text{lower } (\text{floor-float-interval } x) = \text{floor-fl } (\text{lower } x)$   
 ⟨*proof*⟩

**lemma** *upper-floor-float-interval[simp]*:  $\text{upper } (\text{floor-float-interval } x) = \text{floor-fl } (\text{upper } x)$   
 ⟨*proof*⟩

**lemma** *floor-float-intervalI*:  $[x] \in_r \text{floor-float-interval } X$  **if**  $x \in_r X$   
 ⟨*proof*⟩



end

## 52.4 constants for code generation

**definition** *lowerF*::float interval  $\Rightarrow$  float **where** *lowerF* = lower  
**definition** *upperF*::float interval  $\Rightarrow$  float **where** *upperF* = upper

end

## 53 Immutable Arrays with Code Generation

**theory** *IArray*  
**imports** *Main*  
**begin**

### 53.1 Fundamental operations

Immutable arrays are lists wrapped up in an additional constructor. There are no update operations. Hence code generation can safely implement this type by efficient target language arrays. Currently only SML is provided. Could be extended to other target languages and more operations.

**context**  
**begin**

**datatype** 'a *iarray* = *IArray* 'a *list*

**qualified primrec** *list-of* :: 'a *iarray*  $\Rightarrow$  'a *list* **where**  
*list-of* (*IArray* *xs*) = *xs*

**qualified definition** *of-fun* :: (nat  $\Rightarrow$  'a)  $\Rightarrow$  nat  $\Rightarrow$  'a *iarray* **where**  
[*simp*]: *of-fun* *f* *n* = *IArray* (map *f* [0..*n*])

**qualified definition** *sub* :: 'a *iarray*  $\Rightarrow$  nat  $\Rightarrow$  'a (infixl <!!> 100) **where**  
[*simp*]: *as* !! *n* = *IArray.list-of as* ! *n*

**qualified definition** *length* :: 'a *iarray*  $\Rightarrow$  nat **where**  
[*simp*]: *length as* = *List.length* (*IArray.list-of as*)

**qualified definition** *all* :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a *iarray*  $\Rightarrow$  bool **where**  
[*simp*]: *all p as*  $\longleftrightarrow$  ( $\forall a \in \text{set } (\text{list-of } as). p a$ )

**qualified definition** *exists* :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a *iarray*  $\Rightarrow$  bool **where**  
[*simp*]: *exists p as*  $\longleftrightarrow$  ( $\exists a \in \text{set } (\text{list-of } as). p a$ )

**lemma** *of-fun-nth*:  
*IArray.of-fun f n* !! *i* = *f i* **if** *i* < *n*

*<proof>*

**end**

## 53.2 Generic code equations

**lemma** *[code]*:

*size (as :: 'a iarray) = Suc (IArray.length as)*

*<proof>*

**lemma** *[code]*:

*size-iarray f as = Suc (size-list f (IArray.list-of as))*

*<proof>*

**lemma** *[code]*:

*rec-iarray f as = f (IArray.list-of as)*

*<proof>*

**lemma** *[code]*:

*case-iarray f as = f (IArray.list-of as)*

*<proof>*

**lemma** *[code]*:

*set-iarray as = set (IArray.list-of as)*

*<proof>*

**lemma** *[code]*:

*map-iarray f as = IArray (map f (IArray.list-of as))*

*<proof>*

**lemma** *[code]*:

*rel-iarray r as bs = list-all2 r (IArray.list-of as) (IArray.list-of bs)*

*<proof>*

**lemma** *list-of-code* *[code]*:

*IArray.list-of as = map ( $\lambda n. as !! n$ ) [0 ..< IArray.length as]*

*<proof>*

**lemma** *[code]*:

*HOL.equal as bs  $\longleftrightarrow$  HOL.equal (IArray.list-of as) (IArray.list-of bs)*

*<proof>*

**lemma** *[code]*:

*IArray.all p = Not  $\circ$  IArray.exists (Not  $\circ$  p)*

*<proof>*

**context**

**includes** *term-syntax*

**begin**

**lemma** [code]:  
 $\text{Code-Evaluation.term-of } (as :: 'a::\text{typerrep iarray}) =$   
 $\text{Code-Evaluation.Const } (\text{STR } "IArray.iarray.IArray") (\text{TYPEREPEP}('a \text{ list} \Rightarrow 'a$   
 $\text{iarray})) \langle \cdot \rangle (\text{Code-Evaluation.term-of } (IArray.\text{list-of } as))$   
 $\langle \text{proof} \rangle$

**end**

### 53.3 Auxiliary operations for code generation

**context**

**begin**

**qualified primrec**  $\text{tabulate} :: \text{integer} \times (\text{integer} \Rightarrow 'a) \Rightarrow 'a \text{ iarray}$  **where**  
 $\text{tabulate } (n, f) = IArray (\text{map } (f \circ \text{integer-of-nat}) [0..<\text{nat-of-integer } n])$

**lemma** [code]:  
 $IArray.\text{of-fun } f \ n = IArray.\text{tabulate } (\text{integer-of-nat } n, f \circ \text{nat-of-integer})$   
 $\langle \text{proof} \rangle$  **primrec**  $\text{sub}' :: 'a \text{ iarray} \times \text{integer} \Rightarrow 'a$  **where**  
 $\text{sub}' (as, n) = as \ ! \ \text{nat-of-integer } n$

**lemma** [code]:  
 $IArray.\text{sub}' (IArray \ as, n) = as \ ! \ \text{nat-of-integer } n$   
 $\langle \text{proof} \rangle$

**lemma** [code]:  
 $as \ ! \ n = IArray.\text{sub}' (as, \text{integer-of-nat } n)$   
 $\langle \text{proof} \rangle$  **definition**  $\text{length}' :: 'a \text{ iarray} \Rightarrow \text{integer}$  **where**  
 $[\text{simp}]: \text{length}' \ as = \text{integer-of-nat } (\text{List.length } (IArray.\text{list-of } as))$

**lemma** [code]:  
 $IArray.\text{length}' (IArray \ as) = \text{integer-of-nat } (\text{List.length } as)$   
 $\langle \text{proof} \rangle$

**lemma** [code]:  
 $IArray.\text{length } as = \text{nat-of-integer } (IArray.\text{length}' \ as)$   
 $\langle \text{proof} \rangle$  **definition**  $\text{exists-upto} :: ('a \Rightarrow \text{bool}) \Rightarrow \text{integer} \Rightarrow 'a \text{ iarray} \Rightarrow \text{bool}$  **where**  
 $[\text{simp}]: \text{exists-upto } p \ k \ as \longleftrightarrow (\exists l. 0 \leq l \wedge l < k \wedge p (\text{sub}' (as, l)))$

**lemma**  $\text{exists-upto-of-nat}$ :  
 $\text{exists-upto } p \ (\text{of-nat } n) \ as \longleftrightarrow (\exists m < n. p (as \ ! \ m))$   
**including**  $\text{integer.lifting}$   $\langle \text{proof} \rangle$

**lemma** [code]:  
 $\text{exists-upto } p \ k \ as \longleftrightarrow (\text{if } k \leq 0 \text{ then False else}$   
 $\text{let } l = k - 1 \text{ in } p (\text{sub}' (as, l)) \vee \text{exists-upto } p \ l \ as)$   
 $\langle \text{proof} \rangle$   
**including**  $\text{integer.lifting}$   $\langle \text{proof} \rangle$

```

lemma [code]:
  IArray.exists p as  $\longleftrightarrow$  exists-upto p (length' as) as
  including integer.lifting ⟨proof⟩

end

```

### 53.4 Code Generation for SML

Note that arrays cannot be printed directly but only by turning them into lists first. Arrays could be converted back into lists for printing if they were wrapped up in an additional constructor.

**code-reserved** (SML) *Vector*

**code-printing**

```

type-constructor iarray  $\rightarrow$  (SML) - Vector.vector
| constant IArray  $\rightarrow$  (SML) Vector.fromList
| constant IArray.all  $\rightarrow$  (SML) Vector.all
| constant IArray.exists  $\rightarrow$  (SML) Vector.exists
| constant IArray.tabulate  $\rightarrow$  (SML) Vector.tabulate
| constant IArray.sub'  $\rightarrow$  (SML) Vector.sub
| constant IArray.length'  $\rightarrow$  (SML) Vector.length

```

### 53.5 Code Generation for Haskell

We map 'a iarrays in Isabelle/HOL to *Data.Array.IArray.array* in Haskell. Performance mapping to *Data.Array.Unboxed.Array* and *Data.Array.Array* is similar.

**code-printing**

```

code-module IArray  $\rightarrow$  (Haskell)  $\langle$ 
module IArray(IArray, tabulate, of-list, sub, length) where {

  import Prelude (Bool(True, False), not, Maybe(Nothing, Just),
    Integer, (+), (-), (<), fromInteger, toInteger, map, seq, ());
  import qualified Prelude;
  import qualified Data.Array.IArray;
  import qualified Data.Array.Base;
  import qualified Data.Ix;

  newtype IArray e = IArray (Data.Array.IArray.Array Integer e);

  tabulate :: (Integer, (Integer  $\rightarrow$  e))  $\rightarrow$  IArray e;
  tabulate (k, f) = IArray (Data.Array.IArray.array (0, k - 1) (map (\i  $\rightarrow$  let
    fi = f i in fi 'seq' (i, fi)) [0..k - 1]));

  of-list :: [e]  $\rightarrow$  IArray e;
  of-list l = IArray (Data.Array.IArray.listArray (0, (toInteger . Prelude.length) l
    - 1) l);

```

```

sub :: (IArray e, Integer) -> e;
sub (IArray v, i) = v 'Data.Array.Base.unsafeAt' fromInteger i;

length :: IArray e -> Integer;
length (IArray v) = toInteger (Data.Ix.rangeSize (Data.Array.IArray.bounds v));

} } for type-constructor iarray constant IArray IArray.tabulate IArray.sub' IAr-
ray.length'

code-reserved (Haskell) IArray-Impl

code-printing
  type-constructor iarray -> (Haskell) IArray.IArray -
| constant IArray -> (Haskell) IArray.of'-list
| constant IArray.tabulate -> (Haskell) IArray.tabulate
| constant IArray.sub' -> (Haskell) IArray.sub
| constant IArray.length' -> (Haskell) IArray.length

end

```

## 54 Definition of Landau symbols

```

theory Landau-Symbols
imports
  Complex-Main
begin

```

**lemma** *eventually-subst'*:

```

eventually (λx. f x = g x) F ⇒ eventually (λx. P x (f x)) F = eventually (λx.
P x (g x)) F
⟨proof⟩

```

### 54.1 Definition of Landau symbols

Our Landau symbols are sign-oblivious, i.e. any function always has the same growth as its absolute. This has the advantage of making some cancelling rules for sums nicer, but introduces some problems in other places. Nevertheless, we found this definition more convenient to work with.

**definition** *bigo* :: 'a filter ⇒ ('a ⇒ ('b :: real-normed-field)) ⇒ ('a ⇒ 'b) set  
 (⟨⟨⟨indent=1 notation=⟨mixfix bigo⟩ O[-]'(-)⟩⟩) )  
**where** *bigo* F g = {f. (∃ c>0. eventually (λx. norm (f x) ≤ c \* norm (g x)) F)}

**definition** *smallo* :: 'a filter ⇒ ('a ⇒ ('b :: real-normed-field)) ⇒ ('a ⇒ 'b) set  
 (⟨⟨⟨indent=1 notation=⟨mixfix smallo⟩ o[-]'(-)⟩⟩) )  
**where** *smallo* F g = {f. (∀ c>0. eventually (λx. norm (f x) ≤ c \* norm (g x)) F)}

**definition** *bigomega* :: 'a filter  $\Rightarrow$  ('a  $\Rightarrow$  ('b :: real-normed-field))  $\Rightarrow$  ('a  $\Rightarrow$  'b) set  
 ( $\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix bigomega} \rangle \Omega[-]'(-) \rangle \rangle$ )  
**where** *bigomega*  $F g = \{f. (\exists c > 0. \text{eventually } (\lambda x. \text{norm } (f x) \geq c * \text{norm } (g x)) F)\}$

**definition** *smallomega* :: 'a filter  $\Rightarrow$  ('a  $\Rightarrow$  ('b :: real-normed-field))  $\Rightarrow$  ('a  $\Rightarrow$  'b) set  
 set  
 ( $\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix smallomega} \rangle \omega[-]'(-) \rangle \rangle$ )  
**where** *smallomega*  $F g = \{f. (\forall c > 0. \text{eventually } (\lambda x. \text{norm } (f x) \geq c * \text{norm } (g x)) F)\}$

**definition** *bigheta* :: 'a filter  $\Rightarrow$  ('a  $\Rightarrow$  ('b :: real-normed-field))  $\Rightarrow$  ('a  $\Rightarrow$  'b) set  
 ( $\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix bigheta} \rangle \Theta[-]'(-) \rangle \rangle$ )  
**where** *bigheta*  $F g = \text{bigo } F g \cap \text{bigomega } F g$

**abbreviation** *bigo-at-top* ( $\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix bigo} \rangle O'(-) \rangle \rangle$ )  
**where**  $O(g) \equiv \text{bigo at-top } g$

**abbreviation** *smallo-at-top* ( $\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix smallo} \rangle o'(-) \rangle \rangle$ )  
**where**  $o(g) \equiv \text{smallo at-top } g$

**abbreviation** *bigomega-at-top* ( $\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix bigomega} \rangle \Omega'(-) \rangle \rangle$ )  
**where**  $\Omega(g) \equiv \text{bigomega at-top } g$

**abbreviation** *smallomega-at-top* ( $\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix smallomega} \rangle \omega'(-) \rangle \rangle$ )  
**where**  $\omega(g) \equiv \text{smallomega at-top } g$

**abbreviation** *bigheta-at-top* ( $\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix bigheta} \rangle \Theta'(-) \rangle \rangle$ )  
**where**  $\Theta(g) \equiv \text{bigheta at-top } g$

The following is a set of properties that all Landau symbols satisfy.

**named-theorems** *landau-divide-simps*

**locale** *landau-symbol* =

**fixes**  $L :: 'a \text{ filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b) \text{ set}$

**and**  $L' :: 'c \text{ filter} \Rightarrow ('c \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('c \Rightarrow 'b) \text{ set}$

**and**  $Lr :: 'a \text{ filter} \Rightarrow ('a \Rightarrow \text{real}) \Rightarrow ('a \Rightarrow \text{real}) \text{ set}$

**assumes** *bot'*:  $L \text{ bot } f = \text{UNIV}$

**assumes** *filter-mono'*:  $F1 \leq F2 \Longrightarrow L F2 f \subseteq L F1 f$

**assumes** *in-filtermap-iff*:

$f' \in L (\text{filtermap } h' F') g' \longleftrightarrow (\lambda x. f' (h' x)) \in L' F' (\lambda x. g' (h' x))$

**assumes** *filtercomap*:

$f' \in L F'' g' \Longrightarrow (\lambda x. f' (h' x)) \in L' (\text{filtercomap } h' F'') (\lambda x. g' (h' x))$

**assumes** *sup*:  $f \in L F1 g \Longrightarrow f \in L F2 g \Longrightarrow f \in L (\text{sup } F1 F2) g$

**assumes** *in-cong*:  $\text{eventually } (\lambda x. f x = g x) F \Longrightarrow f \in L F (h) \longleftrightarrow g \in L F$

(*h*)

**assumes** *cong*:  $\text{eventually } (\lambda x. f x = g x) F \Longrightarrow L F (f) = L F (g)$

**assumes** *cong-bigheta*:  $f \in \Theta[F](g) \Longrightarrow L F (f) = L F (g)$

**assumes** *in-cong-bigheta*:  $f \in \Theta[F](g) \Longrightarrow f \in L F (h) \longleftrightarrow g \in L F (h)$

**assumes** *cmult* [*simp*]:  $c \neq 0 \implies L F (\lambda x. c * f x) = L F (f)$   
**assumes** *cmult-in-iff* [*simp*]:  $c \neq 0 \implies (\lambda x. c * f x) \in L F (g) \iff f \in L F (g)$   
**assumes** *mult-left* [*simp*]:  $f \in L F (g) \implies (\lambda x. h x * f x) \in L F (\lambda x. h x * g x)$   
**assumes** *inverse*: *eventually*  $(\lambda x. f x \neq 0) F \implies \text{eventually } (\lambda x. g x \neq 0) F$   
 $\implies$

$f \in L F (g) \implies (\lambda x. \text{inverse } (g x)) \in L F (\lambda x. \text{inverse } (f x))$   
**assumes** *subsetI*:  $f \in L F (g) \implies L F (f) \subseteq L F (g)$   
**assumes** *plus-subset1*:  $f \in o[F](g) \implies L F (g) \subseteq L F (\lambda x. f x + g x)$   
**assumes** *trans*:  $f \in L F (g) \implies g \in L F (h) \implies f \in L F (h)$   
**assumes** *compose*:  $f \in L F (g) \implies \text{filterlim } h' F G \implies (\lambda x. f (h' x)) \in L' G$   
 $(\lambda x. g (h' x))$   
**assumes** *norm-iff* [*simp*]:  $(\lambda x. \text{norm } (f x)) \in Lr F (\lambda x. \text{norm } (g x)) \iff f \in L F (g)$   
**assumes** *abs* [*simp*]:  $Lr Fr (\lambda x. |fr x|) = Lr Fr fr$   
**assumes** *abs-in-iff* [*simp*]:  $(\lambda x. |fr x|) \in Lr Fr gr \iff fr \in Lr Fr gr$   
**begin**

**lemma** *bot* [*simp*]:  $f \in L \text{ bot } g \langle \text{proof} \rangle$

**lemma** *filter-mono*:  $F1 \leq F2 \implies f \in L F2 g \implies f \in L F1 g$   
 $\langle \text{proof} \rangle$

**lemma** *cong-ex*:  
*eventually*  $(\lambda x. f1 x = f2 x) F \implies \text{eventually } (\lambda x. g1 x = g2 x) F \implies$   
 $f1 \in L F (g1) \iff f2 \in L F (g2)$   
 $\langle \text{proof} \rangle$

**lemma** *cong-ex-bigtheta*:  
 $f1 \in \Theta[F](f2) \implies g1 \in \Theta[F](g2) \implies f1 \in L F (g1) \iff f2 \in L F (g2)$   
 $\langle \text{proof} \rangle$

**lemma** *bigtheta-trans1*:  
 $f \in L F (g) \implies g \in \Theta[F](h) \implies f \in L F (h)$   
 $\langle \text{proof} \rangle$

**lemma** *bigtheta-trans2*:  
 $f \in \Theta[F](g) \implies g \in L F (h) \implies f \in L F (h)$   
 $\langle \text{proof} \rangle$

**lemma** *cmult'* [*simp*]:  $c \neq 0 \implies L F (\lambda x. f x * c) = L F (f)$   
 $\langle \text{proof} \rangle$

**lemma** *cmult-in-iff'* [*simp*]:  $c \neq 0 \implies (\lambda x. f x * c) \in L F (g) \iff f \in L F (g)$   
 $\langle \text{proof} \rangle$

**lemma** *cdiv* [*simp*]:  $c \neq 0 \implies L F (\lambda x. f x / c) = L F (f)$   
 $\langle \text{proof} \rangle$

**lemma** *cdiv-in-iff'* [*simp*]:  $c \neq 0 \implies (\lambda x. f x / c) \in L F (g) \iff f \in L F (g)$

*<proof>*

**lemma** *uminus* [simp]:  $L F (\lambda x. -g x) = L F (g)$  *<proof>*

**lemma** *uminus-in-iff* [simp]:  $(\lambda x. -f x) \in L F (g) \longleftrightarrow f \in L F (g)$   
*<proof>*

**lemma** *const*:  $c \neq 0 \implies L F (\lambda-. c) = L F (\lambda-. 1)$   
*<proof>*

**lemma** *const'* [simp]: *NO-MATCH*  $1 c \implies c \neq 0 \implies L F (\lambda-. c) = L F (\lambda-. 1)$   
*<proof>*

**lemma** *const-in-iff*:  $c \neq 0 \implies (\lambda-. c) \in L F (f) \longleftrightarrow (\lambda-. 1) \in L F (f)$   
*<proof>*

**lemma** *const-in-iff'* [simp]: *NO-MATCH*  $1 c \implies c \neq 0 \implies (\lambda-. c) \in L F (f) \longleftrightarrow$   
 $(\lambda-. 1) \in L F (f)$   
*<proof>*

**lemma** *plus-subset2*:  $g \in o[F](f) \implies L F (f) \subseteq L F (\lambda x. f x + g x)$   
*<proof>*

**lemma** *mult-right* [simp]:  $f \in L F (g) \implies (\lambda x. f x * h x) \in L F (\lambda x. g x * h x)$   
*<proof>*

**lemma** *mult*:  $f1 \in L F (g1) \implies f2 \in L F (g2) \implies (\lambda x. f1 x * f2 x) \in L F (\lambda x.$   
 $g1 x * g2 x)$   
*<proof>*

**lemma** *inverse-cancel*:

**assumes** *eventually*  $(\lambda x. f x \neq 0) F$

**assumes** *eventually*  $(\lambda x. g x \neq 0) F$

**shows**  $(\lambda x. \text{inverse} (f x)) \in L F (\lambda x. \text{inverse} (g x)) \longleftrightarrow g \in L F (f)$

*<proof>*

**lemma** *divide-right*:

**assumes** *eventually*  $(\lambda x. h x \neq 0) F$

**assumes**  $f \in L F (g)$

**shows**  $(\lambda x. f x / h x) \in L F (\lambda x. g x / h x)$

*<proof>*

**lemma** *divide-right-iff*:

**assumes** *eventually*  $(\lambda x. h x \neq 0) F$

**shows**  $(\lambda x. f x / h x) \in L F (\lambda x. g x / h x) \longleftrightarrow f \in L F (g)$

*<proof>*

**lemma** *divide-left*:



**assumes** *eventually*  $(\lambda x. f x \neq 0) F$   
**assumes** *eventually*  $(\lambda x. g x \neq 0) F$   
**assumes**  $g \in L F(f)$   
**shows**  $(\lambda x. h x / f x) \in L F (\lambda x. h x / g x)$   
*<proof>*

**lemma** *divide-left-iff*:

**assumes** *eventually*  $(\lambda x. f x \neq 0) F$   
**assumes** *eventually*  $(\lambda x. g x \neq 0) F$   
**assumes** *eventually*  $(\lambda x. h x \neq 0) F$   
**shows**  $(\lambda x. h x / f x) \in L F (\lambda x. h x / g x) \longleftrightarrow g \in L F (f)$   
*<proof>*

**lemma** *divide*:

**assumes** *eventually*  $(\lambda x. g1 x \neq 0) F$   
**assumes** *eventually*  $(\lambda x. g2 x \neq 0) F$   
**assumes**  $f1 \in L F (f2) g2 \in L F (g1)$   
**shows**  $(\lambda x. f1 x / g1 x) \in L F (\lambda x. f2 x / g2 x)$   
*<proof>*

**lemma** *divide-eq1*:

**assumes** *eventually*  $(\lambda x. h x \neq 0) F$   
**shows**  $f \in L F (\lambda x. g x / h x) \longleftrightarrow (\lambda x. f x * h x) \in L F (g)$   
*<proof>*

**lemma** *divide-eq2*:

**assumes** *eventually*  $(\lambda x. h x \neq 0) F$   
**shows**  $(\lambda x. f x / h x) \in L F (\lambda x. g x) \longleftrightarrow f \in L F (\lambda x. g x * h x)$   
*<proof>*

**lemma** *inverse-eq1*:

**assumes** *eventually*  $(\lambda x. g x \neq 0) F$   
**shows**  $f \in L F (\lambda x. \text{inverse } (g x)) \longleftrightarrow (\lambda x. f x * g x) \in L F (\lambda-. 1)$   
*<proof>*

**lemma** *inverse-eq2*:

**assumes** *eventually*  $(\lambda x. f x \neq 0) F$   
**shows**  $(\lambda x. \text{inverse } (f x)) \in L F (g) \longleftrightarrow (\lambda x. 1) \in L F (\lambda x. f x * g x)$   
*<proof>*

**lemma** *inverse-flip*:

**assumes** *eventually*  $(\lambda x. g x \neq 0) F$   
**assumes** *eventually*  $(\lambda x. h x \neq 0) F$   
**assumes**  $(\lambda x. \text{inverse } (g x)) \in L F (h)$   
**shows**  $(\lambda x. \text{inverse } (h x)) \in L F (g)$   
*<proof>*

**lemma** *lift-trans*:

**assumes**  $f \in L F (g)$

**assumes**  $(\lambda x. t x (g x)) \in L F (h)$   
**assumes**  $\bigwedge f g. f \in L F (g) \implies (\lambda x. t x (f x)) \in L F (\lambda x. t x (g x))$   
**shows**  $(\lambda x. t x (f x)) \in L F (h)$   
*<proof>*

**lemma** *lift-trans'*:

**assumes**  $f \in L F (\lambda x. t x (g x))$   
**assumes**  $g \in L F (h)$   
**assumes**  $\bigwedge g h. g \in L F (h) \implies (\lambda x. t x (g x)) \in L F (\lambda x. t x (h x))$   
**shows**  $f \in L F (\lambda x. t x (h x))$   
*<proof>*

**lemma** *lift-trans-bigtheta*:

**assumes**  $f \in L F (g)$   
**assumes**  $(\lambda x. t x (g x)) \in \Theta[F](h)$   
**assumes**  $\bigwedge f g. f \in L F (g) \implies (\lambda x. t x (f x)) \in L F (\lambda x. t x (g x))$   
**shows**  $(\lambda x. t x (f x)) \in L F (h)$   
*<proof>*

**lemma** *lift-trans-bigtheta'*:

**assumes**  $f \in L F (\lambda x. t x (g x))$   
**assumes**  $g \in \Theta[F](h)$   
**assumes**  $\bigwedge g h. g \in \Theta[F](h) \implies (\lambda x. t x (g x)) \in \Theta[F](\lambda x. t x (h x))$   
**shows**  $f \in L F (\lambda x. t x (h x))$   
*<proof>*

**lemma** (*in landau-symbol*) *mult-in-1*:

**assumes**  $f \in L F (\lambda-. 1) g \in L F (\lambda-. 1)$   
**shows**  $(\lambda x. f x * g x) \in L F (\lambda-. 1)$   
*<proof>*

**lemma** (*in landau-symbol*) *of-real-cancel*:

$(\lambda x. \text{of-real } (f x)) \in L F (\lambda x. \text{of-real } (g x)) \implies f \in Lr F g$   
*<proof>*

**lemma** (*in landau-symbol*) *of-real-iff*:

$(\lambda x. \text{of-real } (f x)) \in L F (\lambda x. \text{of-real } (g x)) \iff f \in Lr F g$   
*<proof>*

**lemmas** [*landau-divide-simps*] =

*inverse-cancel divide-left-iff divide-eq1 divide-eq2 inverse-eq1 inverse-eq2*

**end**

The symbols  $O$  and  $o$  and  $\Omega$  and  $\omega$  are dual, so for many rules, replacing  $O$  with  $\Omega$ ,  $o$  with  $\omega$ , and  $\leq$  with  $\geq$  in a theorem yields another valid theorem. The following locale captures this fact.

**locale** *landau-pair* =

**fixes**  $L l :: 'a \text{ filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b) \text{ set}$

**fixes**  $L' l' :: 'c \text{ filter} \Rightarrow ('c \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('c \Rightarrow 'b) \text{ set}$   
**fixes**  $Lr lr :: 'a \text{ filter} \Rightarrow ('a \Rightarrow \text{real}) \Rightarrow ('a \Rightarrow \text{real}) \text{ set}$   
**and**  $R :: \text{real} \Rightarrow \text{real} \Rightarrow \text{bool}$   
**assumes**  $L\text{-def}: L F g = \{f. \exists c > 0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g x))) F\}$   
**and**  $l\text{-def}: l F g = \{f. \forall c > 0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g x))) F\}$   
**and**  $L'\text{-def}: L' F' g' = \{f. \exists c > 0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g' x))) F'\}$   
**and**  $l'\text{-def}: l' F' g' = \{f. \forall c > 0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g' x))) F'\}$   
**and**  $Lr\text{-def}: Lr F'' g'' = \{f. \exists c > 0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g'' x))) F''\}$   
**and**  $lr\text{-def}: lr F'' g'' = \{f. \forall c > 0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g'' x))) F''\}$   
**and**  $R: R = (\leq) \vee R = (\geq)$

**interpretation** *landau-o*:

*landau-pair bigo smallo bigo smallo bigo smallo* ( $\leq$ )  
*<proof>*

**interpretation** *landau-omega*:

*landau-pair bigomega smallomega bigomega smallomega bigomega smallomega*  
( $\geq$ )  
*<proof>*

**context** *landau-pair*

**begin**

**lemmas**  $R\text{-E} = \text{disjE } [OF R, \text{case-names } le ge]$

**lemma** *bigI*:

$c > 0 \implies \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g x))) F \implies f \in L F (g)$   
*<proof>*

**lemma** *bigE*:

**assumes**  $f \in L F (g)$   
**obtains**  $c$  **where**  $c > 0 \text{ eventually } (\lambda x. R (\text{norm } (f x)) (c * (\text{norm } (g x)))) F$   
*<proof>*

**lemma** *smallI*:

$(\bigwedge c. c > 0 \implies \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * (\text{norm } (g x)))) F \implies f \in l F (g)$   
*<proof>*

**lemma** *smallD*:

$f \in l F (g) \implies c > 0 \implies \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * (\text{norm } (g x)))) F$   
*<proof>*

**lemma** *bigE-nonneg-real*:

**assumes**  $f \in Lr F (g)$  eventually  $(\lambda x. f x \geq 0)$   $F$

**obtains**  $c$  **where**  $c > 0$  eventually  $(\lambda x. R (f x) (c * |g x|))$   $F$   
 $\langle proof \rangle$

**lemma** *smallD-nonneg-real*:

**assumes**  $f \in lr F (g)$  eventually  $(\lambda x. f x \geq 0)$   $F$   $c > 0$

**shows** eventually  $(\lambda x. R (f x) (c * |g x|))$   $F$

$\langle proof \rangle$

**lemma** *small-imp-big*:  $f \in l F (g) \implies f \in L F (g)$

$\langle proof \rangle$

**lemma** *small-subset-big*:  $l F (g) \subseteq L F (g)$

$\langle proof \rangle$

**lemma** *R-refl [simp]*:  $R x x$   $\langle proof \rangle$

**lemma** *R-linear*:  $\neg R x y \implies R y x$

$\langle proof \rangle$

**lemma** *R-trans [trans]*:  $R a b \implies R b c \implies R a c$

$\langle proof \rangle$

**lemma** *R-mult-left-mono*:  $R a b \implies c \geq 0 \implies R (c*a) (c*b)$

$\langle proof \rangle$

**lemma** *R-mult-right-mono*:  $R a b \implies c \geq 0 \implies R (a*c) (b*c)$

$\langle proof \rangle$

**lemma** *big-trans*:

**assumes**  $f \in L F (g)$   $g \in L F (h)$

**shows**  $f \in L F (h)$

$\langle proof \rangle$

**lemma** *big-small-trans*:

**assumes**  $f \in L F (g)$   $g \in l F (h)$

**shows**  $f \in l F (h)$

$\langle proof \rangle$

**lemma** *small-big-trans*:

**assumes**  $f \in l F (g)$   $g \in L F (h)$

**shows**  $f \in l F (h)$

$\langle proof \rangle$

**lemma** *small-trans*:

$f \in l F (g) \implies g \in l F (h) \implies f \in l F (h)$

$\langle proof \rangle$

**lemma** *small-big-trans'*:

$f \in l F (g) \implies g \in L F (h) \implies f \in L F (h)$   
 ⟨proof⟩

**lemma** *big-small-trans'*:

$f \in L F (g) \implies g \in l F (h) \implies f \in L F (h)$   
 ⟨proof⟩

**lemma** *big-subsetI* [intro]:  $f \in L F (g) \implies L F (f) \subseteq L F (g)$

⟨proof⟩

**lemma** *small-subsetI* [intro]:  $f \in L F (g) \implies l F (f) \subseteq l F (g)$

⟨proof⟩

**lemma** *big-refl* [simp]:  $f \in L F (f)$

⟨proof⟩

**lemma** *small-refl-iff*:  $f \in l F (f) \longleftrightarrow \text{eventually } (\lambda x. f x = 0) F$

⟨proof⟩

**lemma** *big-small-asymmetric*:  $f \in L F (g) \implies g \in l F (f) \implies \text{eventually } (\lambda x. f x = 0) F$

⟨proof⟩

**lemma** *small-big-asymmetric*:  $f \in l F (g) \implies g \in L F (f) \implies \text{eventually } (\lambda x. f x = 0) F$

⟨proof⟩

**lemma** *small-asymmetric*:  $f \in l F (g) \implies g \in l F (f) \implies \text{eventually } (\lambda x. f x = 0) F$

⟨proof⟩

**lemma** *plus-aux*:

**assumes**  $f \in o[F](g)$

**shows**  $g \in L F (\lambda x. f x + g x)$

⟨proof⟩

**end**

**lemma** *summable-comparison-test-bigo*:

**fixes**  $f :: \text{nat} \Rightarrow \text{real}$

**assumes**  $\text{summable } (\lambda n. \text{norm } (g n)) f \in O(g)$

**shows**  $\text{summable } f$

⟨proof⟩

**lemma** *bigomega-iff-bigo*:  $g \in \Omega[F](f) \longleftrightarrow f \in O[F](g)$

⟨proof⟩

**lemma** *smallomega-iff-smallo*:  $g \in \omega[F](f) \longleftrightarrow f \in o[F](g)$   
 ⟨proof⟩

**context** *landau-pair*  
**begin**

**lemma** *big-mono*:  
*eventually*  $(\lambda x. R (\text{norm } (f x)) (\text{norm } (g x))) F \implies f \in L F (g)$   
 ⟨proof⟩

**lemma** *big-mult*:  
**assumes**  $f1 \in L F (g1) f2 \in L F (g2)$   
**shows**  $(\lambda x. f1 x * f2 x) \in L F (\lambda x. g1 x * g2 x)$   
 ⟨proof⟩

**lemma** *small-big-mult*:  
**assumes**  $f1 \in l F (g1) f2 \in L F (g2)$   
**shows**  $(\lambda x. f1 x * f2 x) \in l F (\lambda x. g1 x * g2 x)$   
 ⟨proof⟩

**lemma** *big-small-mult*:  
 $f1 \in L F (g1) \implies f2 \in l F (g2) \implies (\lambda x. f1 x * f2 x) \in l F (\lambda x. g1 x * g2 x)$   
 ⟨proof⟩

**lemma** *small-mult*:  $f1 \in l F (g1) \implies f2 \in l F (g2) \implies (\lambda x. f1 x * f2 x) \in l F (\lambda x. g1 x * g2 x)$   
 ⟨proof⟩

**lemmas** *mult = big-mult small-big-mult big-small-mult small-mult*

**lemma** *big-power*:  
**assumes**  $f \in L F (g)$   
**shows**  $(\lambda x. f x ^ m) \in L F (\lambda x. g x ^ m)$   
 ⟨proof⟩

**lemma** (in *landau-pair*) *small-power*:  
**assumes**  $f \in l F (g) m > 0$   
**shows**  $(\lambda x. f x ^ m) \in l F (\lambda x. g x ^ m)$   
 ⟨proof⟩

**lemma** *big-power-increasing*:  
**assumes**  $(\lambda-. 1) \in L F f m \leq n$   
**shows**  $(\lambda x. f x ^ m) \in L F (\lambda x. f x ^ n)$   
 ⟨proof⟩

**lemma** *small-power-increasing*:  
**assumes**  $(\lambda-. 1) \in l F f m < n$

**shows**  $(\lambda x. f x \hat{=} m) \in l F (\lambda x. f x \hat{=} n)$   
 ⟨proof⟩

**sublocale** *big: landau-symbol L L' Lr*  
 ⟨proof⟩

**sublocale** *small: landau-symbol l l' lr*  
 ⟨proof⟩

These rules allow chaining of Landau symbol propositions in Isar with "also".

**lemma** *big-mult-1*:  $f \in L F (g) \Longrightarrow (\lambda-. 1) \in L F (h) \Longrightarrow f \in L F (\lambda x. g x * h x)$

**and** *big-mult-1'*:  $(\lambda-. 1) \in L F (g) \Longrightarrow f \in L F (h) \Longrightarrow f \in L F (\lambda x. g x * h x)$

**and** *small-mult-1*:  $f \in l F (g) \Longrightarrow (\lambda-. 1) \in L F (h) \Longrightarrow f \in l F (\lambda x. g x * h x)$

**and** *small-mult-1'*:  $(\lambda-. 1) \in L F (g) \Longrightarrow f \in l F (h) \Longrightarrow f \in l F (\lambda x. g x * h x)$

**and** *small-mult-1''*:  $f \in L F (g) \Longrightarrow (\lambda-. 1) \in l F (h) \Longrightarrow f \in l F (\lambda x. g x * h x)$

**and** *small-mult-1'''*:  $(\lambda-. 1) \in l F (g) \Longrightarrow f \in L F (h) \Longrightarrow f \in l F (\lambda x. g x * h x)$

⟨proof⟩

**lemma** *big-1-mult*:  $f \in L F (g) \Longrightarrow h \in L F (\lambda-. 1) \Longrightarrow (\lambda x. f x * h x) \in L F (g)$

**and** *big-1-mult'*:  $h \in L F (\lambda-. 1) \Longrightarrow f \in L F (g) \Longrightarrow (\lambda x. f x * h x) \in L F (g)$

**and** *small-1-mult*:  $f \in l F (g) \Longrightarrow h \in L F (\lambda-. 1) \Longrightarrow (\lambda x. f x * h x) \in l F (g)$

**and** *small-1-mult'*:  $h \in L F (\lambda-. 1) \Longrightarrow f \in l F (g) \Longrightarrow (\lambda x. f x * h x) \in l F (g)$

**and** *small-1-mult''*:  $f \in L F (g) \Longrightarrow h \in l F (\lambda-. 1) \Longrightarrow (\lambda x. f x * h x) \in l F (g)$

**and** *small-1-mult'''*:  $h \in l F (\lambda-. 1) \Longrightarrow f \in L F (g) \Longrightarrow (\lambda x. f x * h x) \in l F (g)$

⟨proof⟩

**lemmas** *mult-1-trans =*

*big-mult-1 big-mult-1' small-mult-1 small-mult-1' small-mult-1'' small-mult-1'''*

*big-1-mult big-1-mult' small-1-mult small-1-mult' small-1-mult'' small-1-mult'''*

**lemma** *big-equal-iff-bigtheta*:  $L F (f) = L F (g) \longleftrightarrow f \in \Theta[F](g)$   
 ⟨proof⟩

**lemma** *big-prod*:

**assumes**  $\bigwedge x. x \in A \Longrightarrow f x \in L F (g x)$

**shows**  $(\lambda y. \prod_{x \in A}. f x y) \in L F (\lambda y. \prod_{x \in A}. g x y)$

⟨proof⟩

**lemma** *big-prod-in-1*:

**assumes**  $\bigwedge x. x \in A \implies f x \in L F (\lambda-. 1)$   
**shows**  $(\lambda y. \prod x \in A. f x y) \in L F (\lambda-. 1)$   
 $\langle proof \rangle$

**end**

**context** *landau-symbol*  
**begin**

**lemma** *plus-absorb1*:  
**assumes**  $f \in o[F](g)$   
**shows**  $L F (\lambda x. f x + g x) = L F (g)$   
 $\langle proof \rangle$

**lemma** *plus-absorb2*:  $g \in o[F](f) \implies L F (\lambda x. f x + g x) = L F (f)$   
 $\langle proof \rangle$

**lemma** *diff-absorb1*:  $f \in o[F](g) \implies L F (\lambda x. f x - g x) = L F (g)$   
 $\langle proof \rangle$

**lemma** *diff-absorb2*:  $g \in o[F](f) \implies L F (\lambda x. f x - g x) = L F (f)$   
 $\langle proof \rangle$

**lemmas** *absorb = plus-absorb1 plus-absorb2 diff-absorb1 diff-absorb2*

**end**

**lemma** *bigthetaI [intro]*:  $f \in O[F](g) \implies f \in \Omega[F](g) \implies f \in \Theta[F](g)$   
 $\langle proof \rangle$

**lemma** *bigthetaD1 [dest]*:  $f \in \Theta[F](g) \implies f \in O[F](g)$   
**and** *bigthetaD2 [dest]*:  $f \in \Theta[F](g) \implies f \in \Omega[F](g)$   
 $\langle proof \rangle$

**lemma** *bigtheta-refl [simp]*:  $f \in \Theta[F](f)$   
 $\langle proof \rangle$

**lemma** *bigtheta-sym*:  $f \in \Theta[F](g) \longleftrightarrow g \in \Theta[F](f)$   
 $\langle proof \rangle$

**lemmas** *landau-flip =*  
*bigomega-iff-bigo[symmetric] smallomega-iff-smallo[symmetric]*  
*bigomega-iff-bigo smallomega-iff-smallo bigtheta-sym*

**interpretation** *landau-theta*: *landau-symbol bigtheta bigtheta bigtheta*  
 $\langle proof \rangle$



**lemmas** *landau-symbols* =  
*landau-o.big.landau-symbol-axioms landau-o.small.landau-symbol-axioms*  
*landau-omega.big.landau-symbol-axioms landau-omega.small.landau-symbol-axioms*  
*landau-theta.landau-symbol-axioms*

**lemma** *bigO* [*intro*]:  
**assumes** *eventually*  $(\lambda x. (\text{norm } (f x)) \leq c * (\text{norm } (g x))) F$   
**shows**  $f \in O[F](g)$   
*<proof>*

**lemma** *smallomegaD* [*dest*]:  
**assumes**  $f \in \omega[F](g)$   
**shows** *eventually*  $(\lambda x. (\text{norm } (f x)) \geq c * (\text{norm } (g x))) F$   
*<proof>*

**lemma** *bigthetaI'*:  
**assumes**  $c1 > 0 c2 > 0$   
**assumes** *eventually*  $(\lambda x. c1 * (\text{norm } (g x)) \leq (\text{norm } (f x)) \wedge (\text{norm } (f x)) \leq c2 * (\text{norm } (g x))) F$   
**shows**  $f \in \Theta[F](g)$   
*<proof>*

**lemma** *bigthetaI-cong*: *eventually*  $(\lambda x. f x = g x) F \implies f \in \Theta[F](g)$   
*<proof>*

**lemma** (**in** *landau-symbol*) *ev-eq-trans1*:  
 $f \in L F (\lambda x. g x (h x)) \implies \text{eventually } (\lambda x. h x = h' x) F \implies f \in L F (\lambda x. g x (h' x))$   
*<proof>*

**lemma** (**in** *landau-symbol*) *ev-eq-trans2*:  
*eventually*  $(\lambda x. f x = f' x) F \implies (\lambda x. g x (f' x)) \in L F (h) \implies (\lambda x. g x (f x)) \in L F (h)$   
*<proof>*

**declare** *landau-o.smallI landau-omega.bigI landau-omega.smallI* [*intro*]  
**declare** *landau-o.bigE landau-omega.bigE* [*elim*]  
**declare** *landau-o.smallD*

**lemma** (**in** *landau-symbol*) *bigtheta-trans1'*:  
 $f \in L F (g) \implies h \in \Theta[F](g) \implies f \in L F (h)$   
*<proof>*

**lemma** (**in** *landau-symbol*) *bigtheta-trans2'*:  
 $g \in \Theta[F](f) \implies g \in L F (h) \implies f \in L F (h)$   
*<proof>*

**lemma** *bigo-bigomega-trans*:  $f \in O[F](g) \implies h \in \Omega[F](g) \implies f \in O[F](h)$   
**and** *bigo-smallomega-trans*:  $f \in O[F](g) \implies h \in \omega[F](g) \implies f \in o[F](h)$   
**and** *smallo-bigomega-trans*:  $f \in o[F](g) \implies h \in \Omega[F](g) \implies f \in o[F](h)$   
**and** *smallo-smallomega-trans*:  $f \in o[F](g) \implies h \in \omega[F](g) \implies f \in o[F](h)$   
**and** *bigomega-bigo-trans*:  $f \in \Omega[F](g) \implies h \in O[F](g) \implies f \in \Omega[F](h)$   
**and** *bigomega-smallo-trans*:  $f \in \Omega[F](g) \implies h \in o[F](g) \implies f \in \omega[F](h)$   
**and** *smallomega-bigo-trans*:  $f \in \omega[F](g) \implies h \in O[F](g) \implies f \in \omega[F](h)$   
**and** *smallomega-smallo-trans*:  $f \in \omega[F](g) \implies h \in o[F](g) \implies f \in \omega[F](h)$   
 ⟨proof⟩

**lemmas** *landau-trans-lift* [trans] =  
*landau-symbols*[THEN *landau-symbol.lift-trans*]  
*landau-symbols*[THEN *landau-symbol.lift-trans*']  
*landau-symbols*[THEN *landau-symbol.lift-trans-bigtheta*]  
*landau-symbols*[THEN *landau-symbol.lift-trans-bigtheta*']

**lemmas** *landau-mult-1-trans* [trans] =  
*landau-o.mult-1-trans* *landau-omega.mult-1-trans*

**lemmas** *landau-trans* [trans] =  
*landau-symbols*[THEN *landau-symbol.bigtheta-trans1*]  
*landau-symbols*[THEN *landau-symbol.bigtheta-trans2*]  
*landau-symbols*[THEN *landau-symbol.bigtheta-trans1*']  
*landau-symbols*[THEN *landau-symbol.bigtheta-trans2*']  
*landau-symbols*[THEN *landau-symbol.ev-eq-trans1*]  
*landau-symbols*[THEN *landau-symbol.ev-eq-trans2*]

*landau-o.big-trans* *landau-o.small-trans* *landau-o.small-big-trans* *landau-o.big-small-trans*  
*landau-omega.big-trans* *landau-omega.small-trans*  
*landau-omega.small-big-trans* *landau-omega.big-small-trans*

*bigo-bigomega-trans* *bigo-smallomega-trans* *smallo-bigomega-trans* *smallo-smallomega-trans*  
*bigomega-bigo-trans* *bigomega-smallo-trans* *smallomega-bigo-trans* *smallomega-smallo-trans*

**lemma** *bigtheta-inverse* [simp]:  
**shows**  $(\lambda x. \text{inverse } (f x)) \in \Theta[F](\lambda x. \text{inverse } (g x)) \iff f \in \Theta[F](g)$   
 ⟨proof⟩

**lemma** *bigtheta-divide*:  
**assumes**  $f1 \in \Theta(f2)$   $g1 \in \Theta(g2)$   
**shows**  $(\lambda x. f1 x / g1 x) \in \Theta(\lambda x. f2 x / g2 x)$   
 ⟨proof⟩

**lemma** *eventually-nonzero-bigtheta*:  
**assumes**  $f \in \Theta[F](g)$   
**shows**  $\text{eventually } (\lambda x. f x \neq 0) F \iff \text{eventually } (\lambda x. g x \neq 0) F$   
 ⟨proof⟩

## 54.2 Landau symbols and limits

**lemma** *bigOI-tendsto-norm*:

**fixes**  $f\ g$

**assumes**  $((\lambda x. \text{norm } (f\ x / g\ x)) \longrightarrow c)\ F$

**assumes** *eventually*  $(\lambda x. g\ x \neq 0)\ F$

**shows**  $f \in O[F](g)$

*<proof>*

**lemma** *bigOI-tendsto*:

**assumes**  $((\lambda x. f\ x / g\ x) \longrightarrow c)\ F$

**assumes** *eventually*  $(\lambda x. g\ x \neq 0)\ F$

**shows**  $f \in O[F](g)$

*<proof>*

**lemma** *bigomegaI-tendsto-norm*:

**assumes** *c-not-0*:  $(c::\text{real}) \neq 0$

**assumes** *lim*:  $((\lambda x. \text{norm } (f\ x / g\ x)) \longrightarrow c)\ F$

**shows**  $f \in \Omega[F](g)$

*<proof>*

**lemma** *bigomegaI-tendsto*:

**assumes** *c-not-0*:  $(c::\text{real}) \neq 0$

**assumes** *lim*:  $((\lambda x. f\ x / g\ x) \longrightarrow c)\ F$

**shows**  $f \in \Omega[F](g)$

*<proof>*

**lemma** *smallomegaI-filterlim-at-top-norm*:

**assumes** *lim*: *filterlim*  $(\lambda x. \text{norm } (f\ x / g\ x))\ \text{at-top}\ F$

**shows**  $f \in \omega[F](g)$

*<proof>*

**lemma** *smallomegaI-filterlim-at-infinity*:

**assumes** *lim*: *filterlim*  $(\lambda x. f\ x / g\ x)\ \text{at-infinity}\ F$

**shows**  $f \in \omega[F](g)$

*<proof>*

**lemma** *smallomegaD-filterlim-at-top-norm*:

**assumes**  $f \in \omega[F](g)$

**assumes** *eventually*  $(\lambda x. g\ x \neq 0)\ F$

**shows**  $LIM\ x\ F. \text{norm } (f\ x / g\ x) \text{:> at-top}$

*<proof>*

**lemma** *smallomegaD-filterlim-at-infinity*:

**assumes**  $f \in \omega[F](g)$

**assumes** *eventually*  $(\lambda x. g\ x \neq 0)\ F$

**shows**  $LIM\ x\ F. f\ x / g\ x \text{:> at-infinity}$

*<proof>*

**lemma** *smallomega-1-conv-filterlim*:  $f \in \omega[F](\lambda-. 1) \iff \text{filterlim } f\ \text{at-infinity}\ F$

*<proof>*

**lemma** *smalloI-tendsto*:

**assumes** *lim*:  $((\lambda x. f\ x / g\ x) \longrightarrow 0)\ F$

**assumes** *eventually*  $(\lambda x. g\ x \neq 0)\ F$

**shows**  $f \in o[F](g)$

*<proof>*

**lemma** *smalloD-tendsto*:

**assumes**  $f \in o[F](g)$

**shows**  $((\lambda x. f\ x / g\ x) \longrightarrow 0)\ F$

*<proof>*

**lemma** *bigthetaI-tendsto-norm*:

**assumes** *c-not-0*:  $(c::real) \neq 0$

**assumes** *lim*:  $((\lambda x. norm\ (f\ x / g\ x)) \longrightarrow c)\ F$

**shows**  $f \in \Theta[F](g)$

*<proof>*

**lemma** *bigthetaI-tendsto*:

**assumes** *c-not-0*:  $(c::real) \neq 0$

**assumes** *lim*:  $((\lambda x. f\ x / g\ x) \longrightarrow c)\ F$

**shows**  $f \in \Theta[F](g)$

*<proof>*

**lemma** *tendsto-add-smallo*:

**assumes**  $(f1 \longrightarrow a)\ F$

**assumes**  $f2 \in o[F](f1)$

**shows**  $((\lambda x. f1\ x + f2\ x) \longrightarrow a)\ F$

*<proof>*

**lemma** *tendsto-diff-smallo*:

**shows**  $(f1 \longrightarrow a)\ F \implies f2 \in o[F](f1) \implies ((\lambda x. f1\ x - f2\ x) \longrightarrow a)\ F$

*<proof>*

**lemma** *tendsto-add-smallo-iff*:

**assumes**  $f2 \in o[F](f1)$

**shows**  $(f1 \longrightarrow a)\ F \iff ((\lambda x. f1\ x + f2\ x) \longrightarrow a)\ F$

*<proof>*

**lemma** *tendsto-diff-smallo-iff*:

**shows**  $f2 \in o[F](f1) \implies (f1 \longrightarrow a)\ F \iff ((\lambda x. f1\ x - f2\ x) \longrightarrow a)\ F$

*<proof>*

**lemma** *tendsto-divide-smallo*:

**assumes**  $((\lambda x. f1\ x / g1\ x) \longrightarrow a)\ F$

**assumes**  $f2 \in o[F](f1)\ g2 \in o[F](g1)$

**assumes** *eventually*  $(\lambda x. g1\ x \neq 0)\ F$

**shows**  $((\lambda x. (f1\ x + f2\ x) / (g1\ x + g2\ x)) \longrightarrow a)\ F$  (**is**  $(?f \longrightarrow -)$  -)

$\langle proof \rangle$

**lemma** *bigo-powr*:

**fixes**  $f :: 'a \Rightarrow real$

**assumes**  $f \in O[F](g) \ p \geq 0$

**shows**  $(\lambda x. |f x| \ powr p) \in O[F](\lambda x. |g x| \ powr p)$

$\langle proof \rangle$

**lemma** *smallo-powr*:

**fixes**  $f :: 'a \Rightarrow real$

**assumes**  $f \in o[F](g) \ p > 0$

**shows**  $(\lambda x. |f x| \ powr p) \in o[F](\lambda x. |g x| \ powr p)$

$\langle proof \rangle$

**lemma** *smallo-powr-nonneg*:

**fixes**  $f :: 'a \Rightarrow real$

**assumes**  $f \in o[F](g) \ p > 0 \ eventually (\lambda x. f x \geq 0) \ F \ eventually (\lambda x. g x \geq 0)$

$F$

**shows**  $(\lambda x. f x \ powr p) \in o[F](\lambda x. g x \ powr p)$

$\langle proof \rangle$

**lemma** *bigtheta-powr*:

**fixes**  $f :: 'a \Rightarrow real$

**shows**  $f \in \Theta[F](g) \implies (\lambda x. |f x| \ powr p) \in \Theta[F](\lambda x. |g x| \ powr p)$

$\langle proof \rangle$

**lemma** *bigo-powr-nonneg*:

**fixes**  $f :: 'a \Rightarrow real$

**assumes**  $f \in O[F](g) \ p \geq 0 \ eventually (\lambda x. f x \geq 0) \ F \ eventually (\lambda x. g x \geq 0)$

$F$

**shows**  $(\lambda x. f x \ powr p) \in O[F](\lambda x. g x \ powr p)$

$\langle proof \rangle$

**lemma** *zero-in-smallo* [simp]:  $(\lambda-. 0) \in o[F](f)$

$\langle proof \rangle$

**lemma** *zero-in-bigo* [simp]:  $(\lambda-. 0) \in O[F](f)$

$\langle proof \rangle$

**lemma** *in-bigomega-zero* [simp]:  $f \in \Omega[F](\lambda x. 0)$

$\langle proof \rangle$

**lemma** *in-smallomega-zero* [simp]:  $f \in \omega[F](\lambda x. 0)$

$\langle proof \rangle$

**lemma** *in-smallo-zero-iff* [simp]:  $f \in o[F](\lambda-. 0) \longleftrightarrow eventually (\lambda x. f x = 0) \ F$

$\langle proof \rangle$

**lemma** *in-bigo-zero-iff* [simp]:  $f \in O[F](\lambda-. 0) \longleftrightarrow \text{eventually } (\lambda x. f x = 0) F$   
 ⟨proof⟩

**lemma** *zero-in-smallomega-iff* [simp]:  $(\lambda-. 0) \in \omega[F](f) \longleftrightarrow \text{eventually } (\lambda x. f x = 0) F$   
 ⟨proof⟩

**lemma** *zero-in-bigomega-iff* [simp]:  $(\lambda-. 0) \in \Omega[F](f) \longleftrightarrow \text{eventually } (\lambda x. f x = 0) F$   
 ⟨proof⟩

**lemma** *zero-in-bigtheta-iff* [simp]:  $(\lambda-. 0) \in \Theta[F](f) \longleftrightarrow \text{eventually } (\lambda x. f x = 0) F$   
 ⟨proof⟩

**lemma** *in-bigtheta-zero-iff* [simp]:  $f \in \Theta[F](\lambda x. 0) \longleftrightarrow \text{eventually } (\lambda x. f x = 0) F$   
 ⟨proof⟩

**lemma** *cmult-in-bigo-iff* [simp]:  $(\lambda x. c * f x) \in O[F](g) \longleftrightarrow c = 0 \vee f \in O[F](g)$

**and** *cmult-in-bigo-iff'* [simp]:  $(\lambda x. f x * c) \in O[F](g) \longleftrightarrow c = 0 \vee f \in O[F](g)$

**and** *cmult-in-smallo-iff* [simp]:  $(\lambda x. c * f x) \in o[F](g) \longleftrightarrow c = 0 \vee f \in o[F](g)$

**and** *cmult-in-smallo-iff'* [simp]:  $(\lambda x. f x * c) \in o[F](g) \longleftrightarrow c = 0 \vee f \in o[F](g)$

⟨proof⟩

**lemma** *bigo-const* [simp]:  $(\lambda-. c) \in O[F](\lambda-. 1)$  ⟨proof⟩

**lemma** *bigo-const-iff* [simp]:  $(\lambda-. c1) \in O[F](\lambda-. c2) \longleftrightarrow F = \text{bot} \vee c1 = 0 \vee c2 \neq 0$   
 ⟨proof⟩

**lemma** *bigomega-const-iff* [simp]:  $(\lambda-. c1) \in \Omega[F](\lambda-. c2) \longleftrightarrow F = \text{bot} \vee c1 \neq 0 \vee c2 = 0$   
 ⟨proof⟩

**lemma** *smallo-real-nat-transfer*:

$(f :: \text{real} \Rightarrow \text{real}) \in o(g) \Longrightarrow (\lambda x :: \text{nat}. f (\text{real } x)) \in o(\lambda x. g (\text{real } x))$

⟨proof⟩

**lemma** *bigo-real-nat-transfer*:

$(f :: \text{real} \Rightarrow \text{real}) \in O(g) \Longrightarrow (\lambda x :: \text{nat}. f (\text{real } x)) \in O(\lambda x. g (\text{real } x))$

⟨proof⟩

**lemma** *smallomega-real-nat-transfer*:

$(f :: \text{real} \Rightarrow \text{real}) \in \omega(g) \Longrightarrow (\lambda x :: \text{nat}. f (\text{real } x)) \in \omega(\lambda x. g (\text{real } x))$

⟨proof⟩

**lemma** *bigomega-real-nat-transfer*:

$(f :: \text{real} \Rightarrow \text{real}) \in \Omega(g) \Longrightarrow (\lambda x :: \text{nat}. f (\text{real } x)) \in \Omega(\lambda x. g (\text{real } x))$   
 ⟨proof⟩

**lemma** *bigheta-real-nat-transfer*:

$(f :: \text{real} \Rightarrow \text{real}) \in \Theta(g) \Longrightarrow (\lambda x :: \text{nat}. f (\text{real } x)) \in \Theta(\lambda x. g (\text{real } x))$   
 ⟨proof⟩

**lemmas** *landau-real-nat-transfer [intro]* =

*bigomega-real-nat-transfer smallo-real-nat-transfer bigomega-real-nat-transfer*  
*smallomega-real-nat-transfer bigheta-real-nat-transfer*

**lemma** *landau-symbol-if-at-top-eq [simp]*:

**assumes** *landau-symbol*  $L L' Lr$

**shows**  $L \text{ at-top } (\lambda x :: 'a :: \text{linordered-semidom}. \text{if } x = a \text{ then } f x \text{ else } g x) = L$   
*at-top*  $(g)$   
 ⟨proof⟩

**lemmas** *landau-symbols-if-at-top-eq [simp]* = *landau-symbols[THEN landau-symbol-if-at-top-eq]*

**lemma** *sum-in-smallo*:

**assumes**  $f \in o[F](h) \ g \in o[F](h)$

**shows**  $(\lambda x. f x + g x) \in o[F](h) \ (\lambda x. f x - g x) \in o[F](h)$   
 ⟨proof⟩

**lemma** *big-sum-in-smallo*:

**assumes**  $\bigwedge x. x \in A \Longrightarrow f x \in o[F](g)$

**shows**  $(\lambda x. \text{sum } (\lambda y. f y x) A) \in o[F](g)$

⟨proof⟩

**lemma** *sum-in-bigo*:

**assumes**  $f \in O[F](h) \ g \in O[F](h)$

**shows**  $(\lambda x. f x + g x) \in O[F](h) \ (\lambda x. f x - g x) \in O[F](h)$   
 ⟨proof⟩

**lemma** *big-sum-in-bigo*:

**assumes**  $\bigwedge x. x \in A \Longrightarrow f x \in O[F](g)$

**shows**  $(\lambda x. \text{sum } (\lambda y. f y x) A) \in O[F](g)$

⟨proof⟩

**lemma** *smallo-multiples*:

**assumes**  $f: f \in o(\text{real})$  **and**  $k > 0$

**shows**  $(\lambda n. f (k * n)) \in o(\text{real})$

⟨proof⟩

**lemma** *maxmin-in-smallo*:

**assumes**  $f \in o[F](h)$   $g \in o[F](h)$

**shows**  $(\lambda k. \max (f k) (g k)) \in o[F](h)$   $(\lambda k. \min (f k) (g k)) \in o[F](h)$

*<proof>*

**lemma** *le-imp-bigo-real*:

**assumes**  $c \geq 0$  *eventually*  $(\lambda x. f x \leq c * (g x :: real))$   $F$  *eventually*  $(\lambda x. 0 \leq f x)$   $F$

**shows**  $f \in O[F](g)$

*<proof>*

**context** *landau-symbol*

**begin**

**lemma** *mult-cancel-left*:

**assumes**  $f1 \in \Theta[F](g1)$  **and** *eventually*  $(\lambda x. g1 x \neq 0)$   $F$

**notes**  $[trans] = \text{bitheta-trans1 bitheta-trans2}$

**shows**  $(\lambda x. f1 x * f2 x) \in L F (\lambda x. g1 x * g2 x) \longleftrightarrow f2 \in L F (g2)$

*<proof>*

**lemma** *mult-cancel-right*:

**assumes**  $f2 \in \Theta[F](g2)$  **and** *eventually*  $(\lambda x. g2 x \neq 0)$   $F$

**shows**  $(\lambda x. f1 x * f2 x) \in L F (\lambda x. g1 x * g2 x) \longleftrightarrow f1 \in L F (g1)$

*<proof>*

**lemma** *divide-cancel-right*:

**assumes**  $f2 \in \Theta[F](g2)$  **and** *eventually*  $(\lambda x. g2 x \neq 0)$   $F$

**shows**  $(\lambda x. f1 x / f2 x) \in L F (\lambda x. g1 x / g2 x) \longleftrightarrow f1 \in L F (g1)$

*<proof>*

**lemma** *divide-cancel-left*:

**assumes**  $f1 \in \Theta[F](g1)$  **and** *eventually*  $(\lambda x. g1 x \neq 0)$   $F$

**shows**  $(\lambda x. f1 x / f2 x) \in L F (\lambda x. g1 x / g2 x) \longleftrightarrow$

$(\lambda x. \text{inverse } (f2 x)) \in L F (\lambda x. \text{inverse } (g2 x))$

*<proof>*

**end**

**lemma** *powr-smallo-iff*:

**assumes** *filterlim*  $g$  *at-top*  $F$   $F \neq \text{bot}$

**shows**  $(\lambda x. g x \text{ powr } p :: real) \in o[F](\lambda x. g x \text{ powr } q) \longleftrightarrow p < q$

*<proof>*

**lemma** *powr-bigo-iff*:

**assumes** *filterlim*  $g$  *at-top*  $F$   $F \neq \text{bot}$

**shows**  $(\lambda x. g x \text{ powr } p :: real) \in O[F](\lambda x. g x \text{ powr } q) \longleftrightarrow p \leq q$

*<proof>*



**lemma** *powr-bigtheta-iff*:

**assumes** *filterlim g at-top F F*  $\neq$  *bot*

**shows**  $(\lambda x. g\ x\ \text{powr}\ p :: \text{real}) \in \Theta[F](\lambda x. g\ x\ \text{powr}\ q) \longleftrightarrow p = q$   
*<proof>*

### 54.3 Flatness of real functions

Given two real-valued functions  $f$  and  $g$ , we say that  $f$  is flatter than  $g$  if any power of  $f(x)$  is asymptotically dominated by any positive power of  $g(x)$ . This is a useful notion since, given two products of powers of functions sorted by flatness, we can compare them asymptotically by simply comparing the exponent lists lexicographically.

A simple sufficient criterion for flatness is that  $\ln f(x) \in o(\ln g(x))$ , which we show now.

**lemma** *ln-smallo-imp-flat*:

**fixes**  $f\ g :: \text{real} \Rightarrow \text{real}$

**assumes** *lim-f: filterlim f at-top at-top*

**assumes** *lim-g: filterlim g at-top at-top*

**assumes** *ln-o-ln:  $(\lambda x. \ln (f\ x)) \in o(\lambda x. \ln (g\ x))$*

**assumes**  $q: q > 0$

**shows**  $(\lambda x. f\ x\ \text{powr}\ p) \in o(\lambda x. g\ x\ \text{powr}\ q)$

*<proof>*

**lemma** *ln-smallo-imp-flat'*:

**fixes**  $f\ g :: \text{real} \Rightarrow \text{real}$

**assumes** *lim-f: filterlim f at-top at-top*

**assumes** *lim-g: filterlim g at-top at-top*

**assumes** *ln-o-ln:  $(\lambda x. \ln (f\ x)) \in o(\lambda x. \ln (g\ x))$*

**assumes**  $q: q < 0$

**shows**  $(\lambda x. g\ x\ \text{powr}\ q) \in o(\lambda x. f\ x\ \text{powr}\ p)$

*<proof>*

### 54.4 Asymptotic Equivalence

**named-theorems** *asympt-equiv-intros*

**named-theorems** *asympt-equiv-simps*

**definition** *asympt-equiv* ::  $('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow 'a\ \text{filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$

$(\langle \langle \text{open-block notation} = \langle \text{mixfix asympt-equiv} \rangle \rangle \sim [-] \rangle [51, 10, 51] 50)$

**where**  $f \sim[F] g \longleftrightarrow ((\lambda x. \text{if } f\ x = 0 \wedge g\ x = 0 \text{ then } 1 \text{ else } f\ x / g\ x) \longrightarrow 1)\ F$

**abbreviation** *(input) asympt-equiv-at-top where*

*asympt-equiv-at-top*  $f\ g \equiv f \sim[at-top] g$

**bundle** *asympt-equiv-syntax*

**begin**

**notation** *asympt-equiv-at-top (infix  $\langle \sim \rangle$  50)*

**end**

**lemma** *asympt-equivI*:  $((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x) \longrightarrow 1)$   
 $F \Longrightarrow f \sim[F] g$   
 ⟨proof⟩

**lemma** *asympt-equivD*:  $f \sim[F] g \Longrightarrow ((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x) \longrightarrow 1) F$   
 ⟨proof⟩

**lemma** *asympt-equiv-filtermap-iff*:  
 $f \sim[\text{filtermap } h F] g \longleftrightarrow (\lambda x. f (h x)) \sim[F] (\lambda x. g (h x))$   
 ⟨proof⟩

**lemma** *asympt-equiv-refl* [*simp*, *asympt-equiv-intros*]:  $f \sim[F] f$   
 ⟨proof⟩

**lemma** *asympt-equiv-symI*:  
 assumes  $f \sim[F] g$   
 shows  $g \sim[F] f$   
 ⟨proof⟩

**lemma** *asympt-equiv-sym*:  $f \sim[F] g \longleftrightarrow g \sim[F] f$   
 ⟨proof⟩

**lemma** *asympt-equivI'*:  
 assumes  $((\lambda x. f x / g x) \longrightarrow 1) F$   
 shows  $f \sim[F] g$   
 ⟨proof⟩

**lemma** *tendsto-imp-asympt-equiv-const*:  
 assumes  $(f \longrightarrow c) F$   $c \neq 0$   
 shows  $f \sim[F] (\lambda-. c)$   
 ⟨proof⟩

**lemma** *asympt-equiv-cong*:  
 assumes *eventually*  $(\lambda x. f1 x = f2 x) F$  *eventually*  $(\lambda x. g1 x = g2 x) F$   
 shows  $f1 \sim[F] g1 \longleftrightarrow f2 \sim[F] g2$   
 ⟨proof⟩

**lemma** *asympt-equiv-eventually-zeros*:  
 fixes  $f g :: 'a \Rightarrow 'b :: \text{real-normed-field}$   
 assumes  $f \sim[F] g$   
 shows *eventually*  $(\lambda x. f x = 0 \longleftrightarrow g x = 0) F$   
 ⟨proof⟩

**lemma** *asympt-equiv-transfer*:  
 assumes  $f1 \sim[F] g1$  *eventually*  $(\lambda x. f1 x = f2 x) F$  *eventually*  $(\lambda x. g1 x = g2 x) F$

**shows**  $f2 \sim[F] g2$   
 ⟨proof⟩

**lemma** *asympt-equiv-transfer-trans* [trans]:  
**assumes**  $(\lambda x. f x (h1 x)) \sim[F] (\lambda x. g x (h1 x))$   
**assumes** *eventually*  $(\lambda x. h1 x = h2 x) F$   
**shows**  $(\lambda x. f x (h2 x)) \sim[F] (\lambda x. g x (h2 x))$   
 ⟨proof⟩

**lemma** *asympt-equiv-trans* [trans]:  
**fixes**  $f g h$   
**assumes**  $f \sim[F] g$   $g \sim[F] h$   
**shows**  $f \sim[F] h$   
 ⟨proof⟩

**lemma** *asympt-equiv-trans-lift1* [trans]:  
**assumes**  $a \sim[F] f$   $b \sim[F] c$   $\wedge c d. c \sim[F] d \implies f c \sim[F] f d$   
**shows**  $a \sim[F] f c$   
 ⟨proof⟩

**lemma** *asympt-equiv-trans-lift2* [trans]:  
**assumes**  $f a \sim[F] b$   $a \sim[F] c$   $\wedge c d. c \sim[F] d \implies f c \sim[F] f d$   
**shows**  $f c \sim[F] b$   
 ⟨proof⟩

**lemma** *asympt-equivD-const*:  
**assumes**  $f \sim[F] (\lambda-. c)$   
**shows**  $(f \longrightarrow c) F$   
 ⟨proof⟩

**lemma** *asympt-equiv-refl-ev*:  
**assumes** *eventually*  $(\lambda x. f x = g x) F$   
**shows**  $f \sim[F] g$   
 ⟨proof⟩

**lemma** *asympt-equiv-nhds-iff*:  $f \sim[nhds (z :: 'a :: t1-space)] g \iff f \sim[at z] g \wedge f z = g z$   
 ⟨proof⟩

**lemma** *asympt-equiv-sandwich*:  
**fixes**  $f g h :: 'a \Rightarrow 'b :: \{\text{real-normed-field, order-topology, linordered-field}\}$   
**assumes** *eventually*  $(\lambda x. f x \geq 0) F$   
**assumes** *eventually*  $(\lambda x. f x \leq g x) F$   
**assumes** *eventually*  $(\lambda x. g x \leq h x) F$   
**assumes**  $f \sim[F] h$   
**shows**  $g \sim[F] f$   $g \sim[F] h$   
 ⟨proof⟩

**lemma** *asympt-equiv-imp-eventually-same-sign*:

**fixes**  $f g :: \text{real} \Rightarrow \text{real}$   
**assumes**  $f \sim[F] g$   
**shows** *eventually*  $(\lambda x. \text{sgn} (f x) = \text{sgn} (g x)) F$   
 $\langle \text{proof} \rangle$

**lemma**

**fixes**  $f g :: - \Rightarrow \text{real}$   
**assumes**  $f \sim[F] g$   
**shows** *asympt-equiv-eventually-same-sign*: *eventually*  $(\lambda x. \text{sgn} (f x) = \text{sgn} (g x)) F$  (**is** *?th1*)  
**and** *asympt-equiv-eventually-neg-iff*: *eventually*  $(\lambda x. f x < 0 \longleftrightarrow g x < 0)$   
 $F$  (**is** *?th2*)  
**and** *asympt-equiv-eventually-pos-iff*: *eventually*  $(\lambda x. f x > 0 \longleftrightarrow g x > 0)$   
 $F$  (**is** *?th3*)  
 $\langle \text{proof} \rangle$

**lemma** *asympt-equiv-tendsto-transfer*:

**assumes**  $f \sim[F] g$  **and**  $(f \longrightarrow c) F$   
**shows**  $(g \longrightarrow c) F$   
 $\langle \text{proof} \rangle$

**lemma** *tendsto-asympt-equiv-cong*:

**assumes**  $f \sim[F] g$   
**shows**  $(f \longrightarrow c) F \longleftrightarrow (g \longrightarrow c) F$   
 $\langle \text{proof} \rangle$

**lemma** *smallo-imp-eventually-sgn*:

**fixes**  $f g :: \text{real} \Rightarrow \text{real}$   
**assumes**  $g \in o(f)$   
**shows** *eventually*  $(\lambda x. \text{sgn} (f x + g x) = \text{sgn} (f x)) \text{at-top}$   
 $\langle \text{proof} \rangle$

**context**

**begin**

**private lemma** *asympt-equiv-add-rightI*:

**assumes**  $f \sim[F] g$   $h \in o[F](g)$   
**shows**  $(\lambda x. f x + h x) \sim[F] g$   
 $\langle \text{proof} \rangle$

**lemma** *asympt-equiv-add-right* [*asympt-equiv-simps*]:

**assumes**  $h \in o[F](g)$   
**shows**  $(\lambda x. f x + h x) \sim[F] g \longleftrightarrow f \sim[F] g$   
 $\langle \text{proof} \rangle$

**end**

**lemma** *asympt-equiv-add-left* [*asympt-equiv-simps*]:

**assumes**  $h \in o[F](g)$   
**shows**  $(\lambda x. h x + f x) \sim[F] g \longleftrightarrow f \sim[F] g$   
 ⟨proof⟩

**lemma** *asympt-equiv-add-right'* [*asympt-equiv-simps*]:

**assumes**  $h \in o[F](g)$   
**shows**  $g \sim[F] (\lambda x. f x + h x) \longleftrightarrow g \sim[F] f$   
 ⟨proof⟩

**lemma** *asympt-equiv-add-left'* [*asympt-equiv-simps*]:

**assumes**  $h \in o[F](g)$   
**shows**  $g \sim[F] (\lambda x. h x + f x) \longleftrightarrow g \sim[F] f$   
 ⟨proof⟩

**lemma** *smallo-imp-asympt-equiv*:

**assumes**  $(\lambda x. f x - g x) \in o[F](g)$   
**shows**  $f \sim[F] g$   
 ⟨proof⟩

**lemma** *asympt-equiv-uminus* [*asympt-equiv-intros*]:

$f \sim[F] g \implies (\lambda x. -f x) \sim[F] (\lambda x. -g x)$   
 ⟨proof⟩

**lemma** *asympt-equiv-uminus-iff* [*asympt-equiv-simps*]:

$(\lambda x. -f x) \sim[F] g \longleftrightarrow f \sim[F] (\lambda x. -g x)$   
 ⟨proof⟩

**lemma** *asympt-equiv-mult* [*asympt-equiv-intros*]:

**fixes**  $f1 f2 g1 g2 :: 'a \Rightarrow 'b :: \text{real-normed-field}$   
**assumes**  $f1 \sim[F] g1 f2 \sim[F] g2$   
**shows**  $(\lambda x. f1 x * f2 x) \sim[F] (\lambda x. g1 x * g2 x)$   
 ⟨proof⟩

**lemma** *asympt-equiv-power* [*asympt-equiv-intros*]:

$f \sim[F] g \implies (\lambda x. f x ^ n) \sim[F] (\lambda x. g x ^ n)$   
 ⟨proof⟩

**lemma** *asympt-equiv-inverse* [*asympt-equiv-intros*]:

**assumes**  $f \sim[F] g$   
**shows**  $(\lambda x. \text{inverse } (f x)) \sim[F] (\lambda x. \text{inverse } (g x))$   
 ⟨proof⟩

**lemma** *asympt-equiv-inverse-iff* [*asympt-equiv-simps*]:

$(\lambda x. \text{inverse } (f x)) \sim[F] (\lambda x. \text{inverse } (g x)) \longleftrightarrow f \sim[F] g$   
 ⟨proof⟩

**lemma** *asympt-equiv-divide* [*asympt-equiv-intros*]:

**assumes**  $f1 \sim[F] g1 f2 \sim[F] g2$   
**shows**  $(\lambda x. f1 x / f2 x) \sim[F] (\lambda x. g1 x / g2 x)$

*<proof>*

**lemma** *asympt-equivD-strong*:

**assumes**  $f \sim[F] g$  eventually  $(\lambda x. f x \neq 0 \vee g x \neq 0)$   $F$

**shows**  $(\lambda x. f x / g x \longrightarrow 1)$   $F$

*<proof>*

**lemma** *asympt-equiv-compose* [*asympt-equiv-intros*]:

**assumes**  $f \sim[G] g$  filterlim  $h$   $G$   $F$

**shows**  $f \circ h \sim[F] g \circ h$

*<proof>*

**lemma** *asympt-equiv-compose'*:

**assumes**  $f \sim[G] g$  filterlim  $h$   $G$   $F$

**shows**  $(\lambda x. f (h x)) \sim[F] (\lambda x. g (h x))$

*<proof>*

**lemma** *asympt-equiv-powr-real* [*asympt-equiv-intros*]:

**fixes**  $f g :: 'a \Rightarrow \text{real}$

**assumes**  $f \sim[F] g$  eventually  $(\lambda x. f x \geq 0)$   $F$  eventually  $(\lambda x. g x \geq 0)$   $F$

**shows**  $(\lambda x. f x \text{ powr } y) \sim[F] (\lambda x. g x \text{ powr } y)$

*<proof>*

**lemma** *asympt-equiv-norm* [*asympt-equiv-intros*]:

**fixes**  $f g :: 'a \Rightarrow 'b :: \text{real-normed-field}$

**assumes**  $f \sim[F] g$

**shows**  $(\lambda x. \text{norm } (f x)) \sim[F] (\lambda x. \text{norm } (g x))$

*<proof>*

**lemma** *asympt-equiv-abs-real* [*asympt-equiv-intros*]:

**fixes**  $f g :: 'a \Rightarrow \text{real}$

**assumes**  $f \sim[F] g$

**shows**  $(\lambda x. |f x|) \sim[F] (\lambda x. |g x|)$

*<proof>*

**lemma** *asympt-equiv-imp-eventually-le*:

**assumes**  $f \sim[F] g$   $c > 1$

**shows** eventually  $(\lambda x. \text{norm } (f x) \leq c * \text{norm } (g x))$   $F$

*<proof>*

**lemma** *asympt-equiv-imp-eventually-ge*:

**assumes**  $f \sim[F] g$   $c < 1$

**shows** eventually  $(\lambda x. \text{norm } (f x) \geq c * \text{norm } (g x))$   $F$

*<proof>*

**lemma** *asympt-equiv-imp-bigo*:

**assumes**  $f \sim[F] g$

**shows**  $f \in O[F](g)$

*<proof>*

**lemma** *asympt-equiv-imp-bigomega*:

$f \sim[F] g \implies f \in \Omega[F](g)$   
 ⟨proof⟩

**lemma** *asympt-equiv-imp-bigtheta*:

$f \sim[F] g \implies f \in \Theta[F](g)$   
 ⟨proof⟩

**lemma** *asympt-equiv-at-infinity-transfer*:

**assumes**  $f \sim[F] g$  *filterlim f at-infinity F*  
**shows** *filterlim g at-infinity F*  
 ⟨proof⟩

**lemma** *asympt-equiv-at-top-transfer*:

**fixes**  $f g :: - \Rightarrow \text{real}$   
**assumes**  $f \sim[F] g$  *filterlim f at-top F*  
**shows** *filterlim g at-top F*  
 ⟨proof⟩

**lemma** *asympt-equiv-at-bot-transfer*:

**fixes**  $f g :: - \Rightarrow \text{real}$   
**assumes**  $f \sim[F] g$  *filterlim f at-bot F*  
**shows** *filterlim g at-bot F*  
 ⟨proof⟩

**lemma** *asympt-equivI'-const*:

**assumes**  $((\lambda x. f x / g x) \longrightarrow c) F c \neq 0$   
**shows**  $f \sim[F] (\lambda x. c * g x)$   
 ⟨proof⟩

**lemma** *asympt-equivI'-inverse-const*:

**assumes**  $((\lambda x. f x / g x) \longrightarrow \text{inverse } c) F c \neq 0$   
**shows**  $(\lambda x. c * f x) \sim[F] g$   
 ⟨proof⟩

**lemma** *filterlim-at-bot-imp-at-infinity*: *filterlim f at-bot F  $\implies$  filterlim f at-infinity F*

**for**  $f :: - \Rightarrow \text{real}$  ⟨proof⟩

**lemma** *asympt-equiv-imp-diff-smallo*:

**assumes**  $f \sim[F] g$   
**shows**  $(\lambda x. f x - g x) \in o[F](g)$   
 ⟨proof⟩

**lemma** *asympt-equiv-altdef*:

$f \sim[F] g \iff (\lambda x. f x - g x) \in o[F](g)$   
 ⟨proof⟩

**lemma** *asympt-equiv-0-left-iff* [simp]:  $(\lambda-. 0) \sim[F] f \longleftrightarrow \text{eventually } (\lambda x. f x = 0)$   
 $F$

**and** *asympt-equiv-0-right-iff* [simp]:  $f \sim[F] (\lambda-. 0) \longleftrightarrow \text{eventually } (\lambda x. f x = 0)$   
 $F$   
 ⟨proof⟩

**lemma** *asympt-equiv-sandwich-real*:

**fixes**  $f g l u :: 'a \Rightarrow \text{real}$

**assumes**  $l \sim[F] g$   $u \sim[F] g$  *eventually*  $(\lambda x. f x \in \{l x..u x\}) F$

**shows**  $f \sim[F] g$

⟨proof⟩

**lemma** *asympt-equiv-sandwich-real'*:

**fixes**  $f g l u :: 'a \Rightarrow \text{real}$

**assumes**  $f \sim[F] l$   $f \sim[F] u$  *eventually*  $(\lambda x. g x \in \{l x..u x\}) F$

**shows**  $f \sim[F] g$

⟨proof⟩

**lemma** *asympt-equiv-sandwich-real''*:

**fixes**  $f g l u :: 'a \Rightarrow \text{real}$

**assumes**  $l1 \sim[F] u1$   $u1 \sim[F] l2$   $l2 \sim[F] u2$

*eventually*  $(\lambda x. f x \in \{l1 x..u1 x\}) F$  *eventually*  $(\lambda x. g x \in \{l2 x..u2 x\}) F$

**shows**  $f \sim[F] g$

⟨proof⟩

end

## 55 Values extended by a bottom element

**theory** *Lattice-Constructions*

**imports** *Main*

**begin**

**datatype**  $'a \text{ bot} = \text{Value } 'a \mid \text{Bot}$

**instantiation**  $\text{bot} :: (\text{preorder}) \text{preorder}$

**begin**

**definition** *less-eq-bot* **where**

$x \leq y \longleftrightarrow (\text{case } x \text{ of Bot} \Rightarrow \text{True} \mid \text{Value } x \Rightarrow (\text{case } y \text{ of Bot} \Rightarrow \text{False} \mid \text{Value } y \Rightarrow x \leq y))$

**definition** *less-bot* **where**

$x < y \longleftrightarrow (\text{case } y \text{ of Bot} \Rightarrow \text{False} \mid \text{Value } y \Rightarrow (\text{case } x \text{ of Bot} \Rightarrow \text{True} \mid \text{Value } x \Rightarrow x < y))$

**lemma** *less-eq-bot-Bot* [simp]:  $\text{Bot} \leq x$

⟨proof⟩



```

lemma less-eq-bot-Bot-code [code]:  $Bot \leq x \longleftrightarrow True$ 
  ⟨proof⟩

lemma less-eq-bot-Bot-is-Bot:  $x \leq Bot \implies x = Bot$ 
  ⟨proof⟩

lemma less-eq-bot-Value-Bot [simp, code]:  $Value\ x \leq Bot \longleftrightarrow False$ 
  ⟨proof⟩

lemma less-eq-bot-Value [simp, code]:  $Value\ x \leq Value\ y \longleftrightarrow x \leq y$ 
  ⟨proof⟩

lemma less-bot-Bot [simp, code]:  $x < Bot \longleftrightarrow False$ 
  ⟨proof⟩

lemma less-bot-Bot-is-Value:  $Bot < x \implies \exists z. x = Value\ z$ 
  ⟨proof⟩

lemma less-bot-Bot-Value [simp]:  $Bot < Value\ x$ 
  ⟨proof⟩

lemma less-bot-Bot-Value-code [code]:  $Bot < Value\ x \longleftrightarrow True$ 
  ⟨proof⟩

lemma less-bot-Value [simp, code]:  $Value\ x < Value\ y \longleftrightarrow x < y$ 
  ⟨proof⟩

instance
  ⟨proof⟩

end

instance bot :: (order) order
  ⟨proof⟩

instance bot :: (linorder) linorder
  ⟨proof⟩

instantiation bot :: (order) bot
begin
  definition bot = Bot
  instance ⟨proof⟩
end

instantiation bot :: (top) top
begin
  definition top = Value top
  instance ⟨proof⟩
end

```

**instantiation** *bot* :: (*semilattice-inf*) *semilattice-inf*  
**begin**

**definition** *inf-bot*

**where**

$$\begin{aligned} \textit{inf } x \ y = & \\ & (\textit{case } x \ \textit{of} \\ & \quad \textit{Bot} \Rightarrow \textit{Bot} \\ & | \ \textit{Value } v \Rightarrow \\ & \quad (\textit{case } y \ \textit{of} \\ & \quad \quad \textit{Bot} \Rightarrow \textit{Bot} \\ & \quad | \ \textit{Value } v' \Rightarrow \textit{Value } (\textit{inf } v \ v')) \end{aligned}$$

**instance**

*<proof>*

**end**

**instantiation** *bot* :: (*semilattice-sup*) *semilattice-sup*  
**begin**

**definition** *sup-bot*

**where**

$$\begin{aligned} \textit{sup } x \ y = & \\ & (\textit{case } x \ \textit{of} \\ & \quad \textit{Bot} \Rightarrow y \\ & | \ \textit{Value } v \Rightarrow \\ & \quad (\textit{case } y \ \textit{of} \\ & \quad \quad \textit{Bot} \Rightarrow x \\ & \quad | \ \textit{Value } v' \Rightarrow \textit{Value } (\textit{sup } v \ v')) \end{aligned}$$

**instance**

*<proof>*

**end**

**instance** *bot* :: (*lattice*) *bounded-lattice-bot*  
*<proof>*

## 55.1 Values extended by a top element

**datatype** *'a top* = *Value 'a* | *Top*

**instantiation** *top* :: (*preorder*) *preorder*

**begin**

**definition** *less-eq-top* **where**

$$x \leq y \iff (\textit{case } y \ \textit{of } \textit{Top} \Rightarrow \textit{True} \mid \textit{Value } y \Rightarrow (\textit{case } x \ \textit{of } \textit{Top} \Rightarrow \textit{False} \mid \textit{Value } x$$

$\Rightarrow x \leq y))$

**definition** *less-top* **where**

$x < y \longleftrightarrow (\text{case } x \text{ of } Top \Rightarrow False \mid \text{Value } x \Rightarrow (\text{case } y \text{ of } Top \Rightarrow True \mid \text{Value } y \Rightarrow x < y))$

**lemma** *less-eq-top-Top* [*simp*]:  $x \leq Top$   
 $\langle \text{proof} \rangle$

**lemma** *less-eq-top-Top-code* [*code*]:  $x \leq Top \longleftrightarrow True$   
 $\langle \text{proof} \rangle$

**lemma** *less-eq-top-is-Top*:  $Top \leq x \Longrightarrow x = Top$   
 $\langle \text{proof} \rangle$

**lemma** *less-eq-top-Top-Value* [*simp*, *code*]:  $Top \leq \text{Value } x \longleftrightarrow False$   
 $\langle \text{proof} \rangle$

**lemma** *less-eq-top-Value-Value* [*simp*, *code*]:  $\text{Value } x \leq \text{Value } y \longleftrightarrow x \leq y$   
 $\langle \text{proof} \rangle$

**lemma** *less-top-Top* [*simp*, *code*]:  $Top < x \longleftrightarrow False$   
 $\langle \text{proof} \rangle$

**lemma** *less-top-Top-is-Value*:  $x < Top \Longrightarrow \exists z. x = \text{Value } z$   
 $\langle \text{proof} \rangle$

**lemma** *less-top-Value-Top* [*simp*]:  $\text{Value } x < Top$   
 $\langle \text{proof} \rangle$

**lemma** *less-top-Value-Top-code* [*code*]:  $\text{Value } x < Top \longleftrightarrow True$   
 $\langle \text{proof} \rangle$

**lemma** *less-top-Value* [*simp*, *code*]:  $\text{Value } x < \text{Value } y \longleftrightarrow x < y$   
 $\langle \text{proof} \rangle$

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

**end**

**instance** *top* :: (*order*) *order*  
 $\langle \text{proof} \rangle$

**instance** *top* :: (*linorder*) *linorder*  
 $\langle \text{proof} \rangle$

**instantiation** *top* :: (*order*) *top*  
**begin**

```

definition top = Top
instance ⟨proof⟩
end

```

```

instantiation top :: (bot) bot
begin
  definition bot = Value bot
  instance ⟨proof⟩
end

```

```

instantiation top :: (semilattice-inf) semilattice-inf
begin

```

```

definition inf-top
where
  inf x y =
    (case x of
      Top ⇒ y
    | Value v ⇒
      (case y of
        Top ⇒ x
      | Value v' ⇒ Value (inf v v')))

```

```

instance
  ⟨proof⟩

```

```

end

```

```

instantiation top :: (semilattice-sup) semilattice-sup
begin

```

```

definition sup-top
where
  sup x y =
    (case x of
      Top ⇒ Top
    | Value v ⇒
      (case y of
        Top ⇒ Top
      | Value v' ⇒ Value (sup v v')))

```

```

instance
  ⟨proof⟩

```

```

end

```

```

instance top :: (lattice) bounded-lattice-top
  ⟨proof⟩

```

## 55.2 Values extended by a top and a bottom element

**datatype** *'a flat-complete-lattice* = Value *'a* | *Bot* | *Top*

**instantiation** *flat-complete-lattice* :: (type) order  
**begin**

**definition** *less-eq-flat-complete-lattice*

**where**

$$x \leq y \equiv$$

(case *x* of  
*Bot*  $\Rightarrow$  *True*  
| *Value v1*  $\Rightarrow$   
(case *y* of  
*Bot*  $\Rightarrow$  *False*  
| *Value v2*  $\Rightarrow$  *v1 = v2*  
| *Top*  $\Rightarrow$  *True*)  
| *Top*  $\Rightarrow$  *y = Top*)

**definition** *less-flat-complete-lattice*

**where**

$$x < y =$$

(case *x* of  
*Bot*  $\Rightarrow$  *y  $\neq$  Bot*  
| *Value v1*  $\Rightarrow$  *y = Top*  
| *Top*  $\Rightarrow$  *False*)

**lemma** [*simp*]: *Bot*  $\leq$  *y*  
 $\langle$ *proof* $\rangle$

**lemma** [*simp*]: *y*  $\leq$  *Top*  
 $\langle$ *proof* $\rangle$

**lemma** *greater-than-two-values*:

**assumes** *a*  $\neq$  *b* *Value a*  $\leq$  *z* *Value b*  $\leq$  *z*  
**shows** *z = Top*  
 $\langle$ *proof* $\rangle$

**lemma** *lesser-than-two-values*:

**assumes** *a*  $\neq$  *b* *z*  $\leq$  *Value a* *z*  $\leq$  *Value b*  
**shows** *z = Bot*  
 $\langle$ *proof* $\rangle$

**instance**

$\langle$ *proof* $\rangle$

**end**

**instantiation** *flat-complete-lattice* :: (type) bot  
**begin**

**definition**  $bot = Bot$   
**instance**  $\langle proof \rangle$   
**end**

**instantiation**  $flat-complete-lattice :: (type) top$   
**begin**  
**definition**  $top = Top$   
**instance**  $\langle proof \rangle$   
**end**

**instantiation**  $flat-complete-lattice :: (type) lattice$   
**begin**

**definition**  $inf-flat-complete-lattice$   
**where**  
 $inf\ x\ y =$   
 (case  $x$  of  
    $Bot \Rightarrow Bot$   
 |  $Value\ v1 \Rightarrow$   
   (case  $y$  of  
      $Bot \Rightarrow Bot$   
     |  $Value\ v2 \Rightarrow if\ v1 = v2\ then\ x\ else\ Bot$   
     |  $Top \Rightarrow x$ )  
 |  $Top \Rightarrow y$ )

**definition**  $sup-flat-complete-lattice$   
**where**  
 $sup\ x\ y =$   
 (case  $x$  of  
    $Bot \Rightarrow y$   
 |  $Value\ v1 \Rightarrow$   
   (case  $y$  of  
      $Bot \Rightarrow x$   
     |  $Value\ v2 \Rightarrow if\ v1 = v2\ then\ x\ else\ Top$   
     |  $Top \Rightarrow Top$ )  
 |  $Top \Rightarrow Top$ )

**instance**  
 $\langle proof \rangle$

**end**

**instantiation**  $flat-complete-lattice :: (type) complete-lattice$   
**begin**

**definition**  $Sup-flat-complete-lattice$   
**where**  
 $Sup\ A =$   
 (if  $A = \{\}$   $\vee$   $A = \{Bot\}$  then  $Bot$ )

else if  $\exists v. A - \{Bot\} = \{Value\ v\}$  then Value (THE v. A - {Bot} = {Value v})  
 else Top)

**definition** *Inf-flat-complete-lattice*

**where**

Inf A =  
 (if A = {}  $\vee$  A = {Top} then Top  
 else if  $\exists v. A - \{Top\} = \{Value\ v\}$  then Value (THE v. A - {Top} = {Value v})  
 else Bot)

**instance**

*<proof>*

**end**

**end**

## 56 Infinite Streams

**theory** *Stream*

**imports** *Nat-Bijection*

**begin**

**codatatype** (sset: 'a) stream =  
 SCons (shd: 'a) (stl: 'a stream) (**infixr** <##> 65)  
**for**  
 map: smap  
 rel: stream-all2

**context**

**begin**

— for code generation only

**qualified definition** *smember* :: 'a  $\Rightarrow$  'a stream  $\Rightarrow$  bool **where**

[code-abbrev]: *smember* x s  $\longleftrightarrow$  x  $\in$  sset s

**lemma** *smember-code*[code, simp]: *smember* x (y ## s) = (if x = y then True else *smember* x s)  
*<proof>*

**end**

**lemmas** *smap-simps*[simp] = stream.map-sel

**lemmas** *shd-sset* = stream.set-sel(1)

**lemmas** *stl-sset* = stream.set-sel(2)

**theorem** *sset-induct*[consumes 1, case-names shd stl, induct set: sset]:

**assumes**  $y \in \text{sset } s$  **and**  $\bigwedge s. P (\text{shd } s) s$  **and**  $\bigwedge s y. \llbracket y \in \text{sset } (\text{stl } s); P y (\text{stl } s) \rrbracket$   
 $\implies P y s$   
**shows**  $P y s$   
 $\langle \text{proof} \rangle$

**lemma** *smap-ctr*:  $\text{smap } f s = x \#\# s' \longleftrightarrow f (\text{shd } s) = x \wedge \text{smap } f (\text{stl } s) = s'$   
 $\langle \text{proof} \rangle$

### 56.1 prepend list to stream

**primrec** *shift* :: 'a list  $\Rightarrow$  'a stream  $\Rightarrow$  'a stream (**infixr**  $\langle @- \rangle$  65) **where**  
*shift* []  $s = s$   
| *shift* (x # xs)  $s = x \#\# \text{shift } xs s$

**lemma** *smap-shift[simp]*:  $\text{smap } f (xs @- s) = \text{map } f xs @- \text{smap } f s$   
 $\langle \text{proof} \rangle$

**lemma** *shift-append[simp]*:  $(xs @ ys) @- s = xs @- ys @- s$   
 $\langle \text{proof} \rangle$

**lemma** *shift-simps[simp]*:  
 $\text{shd } (xs @- s) = (\text{if } xs = [] \text{ then } \text{shd } s \text{ else } \text{hd } xs)$   
 $\text{stl } (xs @- s) = (\text{if } xs = [] \text{ then } \text{stl } s \text{ else } \text{tl } xs @- s)$   
 $\langle \text{proof} \rangle$

**lemma** *sset-shift[simp]*:  $\text{sset } (xs @- s) = \text{set } xs \cup \text{sset } s$   
 $\langle \text{proof} \rangle$

**lemma** *shift-left-inj[simp]*:  $xs @- s1 = xs @- s2 \longleftrightarrow s1 = s2$   
 $\langle \text{proof} \rangle$

### 56.2 set of streams with elements in some fixed set

**context**

**notes**  $\llbracket \text{inductive-internals} \rrbracket$

**begin**

**coinductive-set**

*streams* :: 'a set  $\Rightarrow$  'a stream set

**for**  $A :: 'a \text{ set}$

**where**

*Stream*[*intro!*, *simp*, *no-atp*]:  $\llbracket a \in A; s \in \text{streams } A \rrbracket \implies a \#\# s \in \text{streams } A$

**end**

**lemma** *in-streams*:  $\text{stl } s \in \text{streams } S \implies \text{shd } s \in S \implies s \in \text{streams } S$   
 $\langle \text{proof} \rangle$

**lemma** *streamsE*:  $s \in \text{streams } A \implies (\text{shd } s \in A \implies \text{stl } s \in \text{streams } A \implies P)$   
 $\implies P$



*<proof>*

**lemma** *Stream-image*:  $x \#\# y \in ((\#\#) x') \cdot Y \longleftrightarrow x = x' \wedge y \in Y$   
*<proof>*

**lemma** *shift-streams*:  $\llbracket w \in \text{lists } A; s \in \text{streams } A \rrbracket \Longrightarrow w @- s \in \text{streams } A$   
*<proof>*

**lemma** *streams-Stream*:  $x \#\# s \in \text{streams } A \longleftrightarrow x \in A \wedge s \in \text{streams } A$   
*<proof>*

**lemma** *streams-stl*:  $s \in \text{streams } A \Longrightarrow \text{stl } s \in \text{streams } A$   
*<proof>*

**lemma** *streams-shd*:  $s \in \text{streams } A \Longrightarrow \text{shd } s \in A$   
*<proof>*

**lemma** *sset-streams*:  
**assumes**  $\text{sset } s \subseteq A$   
**shows**  $s \in \text{streams } A$   
*<proof>*

**lemma** *streams-sset*:  
**assumes**  $s \in \text{streams } A$   
**shows**  $\text{sset } s \subseteq A$   
*<proof>*

**lemma** *streams-iff-sset*:  $s \in \text{streams } A \longleftrightarrow \text{sset } s \subseteq A$   
*<proof>*

**lemma** *streams-mono*:  $s \in \text{streams } A \Longrightarrow A \subseteq B \Longrightarrow s \in \text{streams } B$   
*<proof>*

**lemma** *streams-mono2*:  $S \subseteq T \Longrightarrow \text{streams } S \subseteq \text{streams } T$   
*<proof>*

**lemma** *smap-streams*:  $s \in \text{streams } A \Longrightarrow (\bigwedge x. x \in A \Longrightarrow f x \in B) \Longrightarrow \text{smap } f s \in \text{streams } B$   
*<proof>*

**lemma** *streams-empty*:  $\text{streams } \{\} = \{\}$   
*<proof>*

**lemma** *streams-UNIV[simp]*:  $\text{streams } UNIV = UNIV$   
*<proof>*

### 56.3 nth, take, drop for streams

**primrec**  $\text{snth} :: 'a \text{ stream} \Rightarrow \text{nat} \Rightarrow 'a$  (infixl  $\langle !! \rangle$  100) **where**

$s !! 0 = shd s$   
 $| s !! Suc n = stl s !! n$

**lemma** *snth-Stream*:  $(x \#\# s) !! Suc i = s !! i$   
 $\langle proof \rangle$

**lemma** *snth-smap[simp]*:  $smap f s !! n = f (s !! n)$   
 $\langle proof \rangle$

**lemma** *shift-snth-less[simp]*:  $p < length xs \implies (xs @- s) !! p = xs ! p$   
 $\langle proof \rangle$

**lemma** *shift-snth-ge[simp]*:  $p \geq length xs \implies (xs @- s) !! p = s !! (p - length xs)$   
 $\langle proof \rangle$

**lemma** *shift-snth*:  $(xs @- s) !! n = (if n < length xs then xs ! n else s !! (n - length xs))$   
 $\langle proof \rangle$

**lemma** *snth-sset[simp]*:  $s !! n \in sset s$   
 $\langle proof \rangle$

**lemma** *sset-range*:  $sset s = range (snth s)$   
 $\langle proof \rangle$

**lemma** *streams-iff-snth*:  $s \in streams X \iff (\forall n. s !! n \in X)$   
 $\langle proof \rangle$

**lemma** *snth-in*:  $s \in streams X \implies s !! n \in X$   
 $\langle proof \rangle$

**primrec** *stake* ::  $nat \Rightarrow 'a stream \Rightarrow 'a list$  **where**  
 $stake 0 s = []$   
 $| stake (Suc n) s = shd s \# stake n (stl s)$

**lemma** *length-stake[simp]*:  $length (stake n s) = n$   
 $\langle proof \rangle$

**lemma** *stake-smap[simp]*:  $stake n (smap f s) = map f (stake n s)$   
 $\langle proof \rangle$

**lemma** *take-stake*:  $take n (stake m s) = stake (min n m) s$   
 $\langle proof \rangle$

**primrec** *sdrop* ::  $nat \Rightarrow 'a stream \Rightarrow 'a stream$  **where**  
 $sdrop 0 s = s$   
 $| sdrop (Suc n) s = sdrop n (stl s)$

**lemma** *sdrop-simps[simp]*:

$shd (sdrop\ n\ s) = s\ !!\ n\ stl (sdrop\ n\ s) = sdrop\ (Suc\ n)\ s$   
 ⟨proof⟩

**lemma** *sdrop-smap[simp]*:  $sdrop\ n\ (smap\ f\ s) = smap\ f\ (sdrop\ n\ s)$   
 ⟨proof⟩

**lemma** *sdrop-stl*:  $sdrop\ n\ (stl\ s) = stl\ (sdrop\ n\ s)$   
 ⟨proof⟩

**lemma** *drop-stake*:  $drop\ n\ (stake\ m\ s) = stake\ (m - n)\ (sdrop\ n\ s)$   
 ⟨proof⟩

**lemma** *stake-sdrop*:  $stake\ n\ s\ @- sdrop\ n\ s = s$   
 ⟨proof⟩

**lemma** *id-stake-snth-sdrop*:  
 $s = stake\ i\ s\ @- s\ !!\ i\ ##\ sdrop\ (Suc\ i)\ s$   
 ⟨proof⟩

**lemma** *smap-alt*:  $smap\ f\ s = s' \longleftrightarrow (\forall n. f\ (s\ !!\ n) = s'\ !!\ n) \text{ (is ?L = ?R)}$   
 ⟨proof⟩

**lemma** *stake-invert-Nil[iff]*:  $stake\ n\ s = [] \longleftrightarrow n = 0$   
 ⟨proof⟩

**lemma** *sdrop-shift*:  $sdrop\ i\ (w\ @- s) = drop\ i\ w\ @- sdrop\ (i - length\ w)\ s$   
 ⟨proof⟩

**lemma** *stake-shift*:  $stake\ i\ (w\ @- s) = take\ i\ w\ @ stake\ (i - length\ w)\ s$   
 ⟨proof⟩

**lemma** *stake-add[simp]*:  $stake\ m\ s\ @ stake\ n\ (sdrop\ m\ s) = stake\ (m + n)\ s$   
 ⟨proof⟩

**lemma** *sdrop-add[simp]*:  $sdrop\ n\ (sdrop\ m\ s) = sdrop\ (m + n)\ s$   
 ⟨proof⟩

**lemma** *sdrop-snth*:  $sdrop\ n\ s\ !!\ m = s\ !!\ (n + m)$   
 ⟨proof⟩

**partial-function** (*tailrec*) *sdrop-while* ::  $('a \Rightarrow bool) \Rightarrow 'a\ stream \Rightarrow 'a\ stream$   
**where**

$sdrop-while\ P\ s = (if\ P\ (shd\ s) then\ sdrop-while\ P\ (stl\ s) else\ s)$

**lemma** *sdrop-while-SCons[code]*:  
 $sdrop-while\ P\ (a\ ##\ s) = (if\ P\ a then\ sdrop-while\ P\ s else\ a\ ##\ s)$   
 ⟨proof⟩

**lemma** *sdrop-while-sdrop-LEAST*:

**assumes**  $\exists n. P (s !! n)$   
**shows**  $sdrop\text{-}while (Not \circ P) s = sdrop (LEAST n. P (s !! n)) s$   
 $\langle proof \rangle$

**primcorec** *sfilter* **where**

$shd (sfilter P s) = shd (sdrop\text{-}while (Not \circ P) s)$   
 $| stl (sfilter P s) = sfilter P (stl (sdrop\text{-}while (Not \circ P) s))$

**lemma** *sfilter-Stream*:  $sfilter P (x \#\# s) = (if P x then x \#\# sfilter P s else sfilter P s)$   
 $\langle proof \rangle$

## 56.4 unary predicates lifted to streams

**definition** *stream-all*  $P s = (\forall p. P (s !! p))$

**lemma** *stream-all-iff*[*iff*]:  $stream\text{-}all P s \longleftrightarrow Ball (sset s) P$   
 $\langle proof \rangle$

**lemma** *stream-all-shift*[*simp*]:  $stream\text{-}all P (xs @- s) = (list\text{-}all P xs \wedge stream\text{-}all P s)$   
 $\langle proof \rangle$

**lemma** *stream-all-Stream*:  $stream\text{-}all P (x \#\# X) \longleftrightarrow P x \wedge stream\text{-}all P X$   
 $\langle proof \rangle$

## 56.5 recurring stream out of a list

**primcorec** *cycle* :: 'a list  $\Rightarrow$  'a stream **where**

$shd (cycle xs) = hd xs$   
 $| stl (cycle xs) = cycle (tl xs @ [hd xs])$

**lemma** *cycle-decomp*:  $u \neq [] \implies cycle u = u @- cycle u$   
 $\langle proof \rangle$

**lemma** *cycle-Cons*[*code*]:  $cycle (x \# xs) = x \#\# cycle (xs @ [x])$   
 $\langle proof \rangle$

**lemma** *cycle-rotated*:  $\llbracket v \neq []; cycle u = v @- s \rrbracket \implies cycle (tl u @ [hd u]) = tl v @- s$   
 $\langle proof \rangle$

**lemma** *stake-append*:  $stake n (u @- s) = take (min (length u) n) u @ stake (n - length u) s$   
 $\langle proof \rangle$

**lemma** *stake-cycle-le*[*simp*]:

**assumes**  $u \neq []$   $n < length u$   
**shows**  $stake n (cycle u) = take n u$   
 $\langle proof \rangle$

**lemma** *stake-cycle-eq[simp]*:  $u \neq [] \implies \text{stake } (\text{length } u) (\text{cycle } u) = u$   
 ⟨proof⟩

**lemma** *sdrop-cycle-eq[simp]*:  $u \neq [] \implies \text{sdrop } (\text{length } u) (\text{cycle } u) = \text{cycle } u$   
 ⟨proof⟩

**lemma** *stake-cycle-eq-mod-0[simp]*:  $\llbracket u \neq []; n \bmod \text{length } u = 0 \rrbracket \implies$   
 $\text{stake } n (\text{cycle } u) = \text{concat } (\text{replicate } (n \text{ div } \text{length } u) u)$   
 ⟨proof⟩

**lemma** *sdrop-cycle-eq-mod-0[simp]*:  $\llbracket u \neq []; n \bmod \text{length } u = 0 \rrbracket \implies$   
 $\text{sdrop } n (\text{cycle } u) = \text{cycle } u$   
 ⟨proof⟩

**lemma** *stake-cycle*:  $u \neq [] \implies$   
 $\text{stake } n (\text{cycle } u) = \text{concat } (\text{replicate } (n \text{ div } \text{length } u) u) @ \text{take } (n \bmod \text{length } u) u$   
 ⟨proof⟩

**lemma** *sdrop-cycle*:  $u \neq [] \implies \text{sdrop } n (\text{cycle } u) = \text{cycle } (\text{rotate } (n \bmod \text{length } u) u)$   
 ⟨proof⟩

**lemma** *sset-cycle[simp]*:  
**assumes**  $xs \neq []$   
**shows**  $\text{sset } (\text{cycle } xs) = \text{set } xs$   
 ⟨proof⟩

## 56.6 iterated application of a function

**primcorec** *siterate* **where**  
 $\text{shd } (\text{siterate } f x) = x$   
 $| \text{stl } (\text{siterate } f x) = \text{siterate } f (f x)$

**lemma** *stake-Suc*:  $\text{stake } (\text{Suc } n) s = \text{stake } n s @ [s !! n]$   
 ⟨proof⟩

**lemma** *snth-siterate[simp]*:  $\text{siterate } f x !! n = (f \overset{\sim}{\sim} n) x$   
 ⟨proof⟩

**lemma** *sdrop-siterate[simp]*:  $\text{sdrop } n (\text{siterate } f x) = \text{siterate } f ((f \overset{\sim}{\sim} n) x)$   
 ⟨proof⟩

**lemma** *stake-siterate[simp]*:  $\text{stake } n (\text{siterate } f x) = \text{map } (\lambda n. (f \overset{\sim}{\sim} n) x) [0 ..< n]$   
 ⟨proof⟩

**lemma** *sset-siterate*:  $\text{sset } (\text{siterate } f x) = \{(f \overset{\sim}{\sim} n) x \mid n. \text{True}\}$   
 ⟨proof⟩

**lemma** *smap-siterate*:  $\text{smap } f (\text{siterate } f x) = \text{siterate } f (f x)$   
 ⟨proof⟩

## 56.7 stream repeating a single element

**abbreviation** *sconst*  $\equiv$  *siterate id*

**lemma** *shift-replicate-sconst[simp]*: *replicate*  $n x @- \text{sconst } x = \text{sconst } x$   
 ⟨proof⟩

**lemma** *sset-sconst[simp]*:  $\text{sset } (\text{sconst } x) = \{x\}$   
 ⟨proof⟩

**lemma** *sconst-alt*:  $s = \text{sconst } x \longleftrightarrow \text{sset } s = \{x\}$   
 ⟨proof⟩

**lemma** *sconst-cycle*:  $\text{sconst } x = \text{cycle } [x]$   
 ⟨proof⟩

**lemma** *smap-sconst*:  $\text{smap } f (\text{sconst } x) = \text{sconst } (f x)$   
 ⟨proof⟩

**lemma** *sconst-streams*:  $x \in A \implies \text{sconst } x \in \text{streams } A$   
 ⟨proof⟩

**lemma** *streams-empty-iff*:  $\text{streams } S = \{\} \longleftrightarrow S = \{\}$   
 ⟨proof⟩

## 56.8 stream of natural numbers

**abbreviation** *fromN*  $\equiv$  *siterate Suc*

**abbreviation** *nats*  $\equiv$  *fromN 0*

**lemma** *sset-fromN[simp]*:  $\text{sset } (\text{fromN } n) = \{n ..\}$   
 ⟨proof⟩

**lemma** *stream-smap-fromN*:  $s = \text{smap } (\lambda j. \text{let } i = j - n \text{ in } s !! i) (\text{fromN } n)$   
 ⟨proof⟩

**lemma** *stream-smap-nats*:  $s = \text{smap } (\text{snth } s) \text{ nats}$   
 ⟨proof⟩

## 56.9 flatten a stream of lists

**primcorec** *flat* where

$\text{shd } (\text{flat } ws) = \text{hd } (\text{shd } ws)$   
 $|\ \text{stl } (\text{flat } ws) = \text{flat } (\text{if } \text{tl } (\text{shd } ws) = [] \text{ then } \text{stl } ws \text{ else } \text{tl } (\text{shd } ws) \#\# \text{stl } ws)$

**lemma** *flat-Cons*[simp, code]:  $\text{flat } ((x \# xs) \#\# ws) = x \#\# \text{flat } (if\ xs = []\ then\ ws\ else\ xs \#\# ws)$

*<proof>*

**lemma** *flat-Stream*[simp]:  $xs \neq [] \implies \text{flat } (xs \#\# ws) = xs @- \text{flat } ws$

*<proof>*

**lemma** *flat-unfold*:  $\text{shd } ws \neq [] \implies \text{flat } ws = \text{shd } ws @- \text{flat } (\text{stl } ws)$

*<proof>*

**lemma** *flat-snth*:  $\forall xs \in \text{sset } s. xs \neq [] \implies \text{flat } s !! n = (if\ n < \text{length } (\text{shd } s)\ then\ \text{shd } s ! n\ else\ \text{flat } (\text{stl } s) !! (n - \text{length } (\text{shd } s)))$

*<proof>*

**lemma** *sset-flat*[simp]:  $\forall xs \in \text{sset } s. xs \neq [] \implies \text{sset } (\text{flat } s) = (\bigcup xs \in \text{sset } s. \text{set } xs) \text{ (is } ?P \implies ?L = ?R)$

*<proof>*

### 56.10 merge a stream of streams

**definition** *smerge* :: 'a stream  $\Rightarrow$  'a stream **where**

$\text{smerge } ss = \text{flat } (\text{smap } (\lambda n. \text{map } (\lambda s. s !! n) (\text{stake } (\text{Suc } n) ss) @ \text{stake } n (ss !! n)) \text{ nats})$

**lemma** *stake-nth*[simp]:  $m < n \implies \text{stake } n\ s ! m = s !! m$

*<proof>*

**lemma** *snth-sset-smerge*:  $ss !! n !! m \in \text{sset } (\text{smerge } ss)$

*<proof>*

**lemma** *sset-smerge*:  $\text{sset } (\text{smerge } ss) = \bigcup (\text{sset } ` (\text{sset } ss))$

*<proof>*

### 56.11 product of two streams

**definition** *sproduct* :: 'a stream  $\Rightarrow$  'b stream  $\Rightarrow$  ('a  $\times$  'b) stream **where**

$\text{sproduct } s1\ s2 = \text{smerge } (\text{smap } (\lambda x. \text{smap } (\text{Pair } x) s2) s1)$

**lemma** *sset-sproduct*:  $\text{sset } (\text{sproduct } s1\ s2) = \text{sset } s1 \times \text{sset } s2$

*<proof>*

### 56.12 interleave two streams

**primcorec** *sinterleave* **where**

$\text{shd } (\text{sinterleave } s1\ s2) = \text{shd } s1$

$| \text{stl } (\text{sinterleave } s1\ s2) = \text{sinterleave } s2\ (\text{stl } s1)$

**lemma** *sinterleave-code*[code]:

$\text{sinterleave } (x \#\# s1)\ s2 = x \#\# \text{sinterleave } s2\ s1$

*<proof>*

**lemma** *sinterleave-snth*[simp]:

*even n*  $\implies$  *sinterleave s1 s2 !! n = s1 !! (n div 2)*  
*odd n*  $\implies$  *sinterleave s1 s2 !! n = s2 !! (n div 2)*  
 ⟨proof⟩

**lemma** *sset-sinterleave*: *sset (sinterleave s1 s2) = sset s1  $\cup$  sset s2*  
 ⟨proof⟩

### 56.13 zip

**primcorec** *szip* **where**

*shd (szip s1 s2) = (shd s1, shd s2)*  
 | *stl (szip s1 s2) = szip (stl s1) (stl s2)*

**lemma** *szip-unfold*[code]: *szip (a ## s1) (b ## s2) = (a, b) ## (szip s1 s2)*  
 ⟨proof⟩

**lemma** *snth-szip*[simp]: *szip s1 s2 !! n = (s1 !! n, s2 !! n)*  
 ⟨proof⟩

**lemma** *stake-szip*[simp]:

*stake n (szip s1 s2) = zip (stake n s1) (stake n s2)*  
 ⟨proof⟩

**lemma** *sdrop-szip*[simp]: *sdrop n (szip s1 s2) = szip (sdrop n s1) (sdrop n s2)*  
 ⟨proof⟩

**lemma** *smap-szip-fst*:

*smap ( $\lambda x. f (fst x)$ ) (szip s1 s2) = smap f s1*  
 ⟨proof⟩

**lemma** *smap-szip-snd*:

*smap ( $\lambda x. g (snd x)$ ) (szip s1 s2) = smap g s2*  
 ⟨proof⟩

### 56.14 zip via function

**primcorec** *smap2* **where**

*shd (smap2 f s1 s2) = f (shd s1) (shd s2)*  
 | *stl (smap2 f s1 s2) = smap2 f (stl s1) (stl s2)*

**lemma** *smap2-unfold*[code]:

*smap2 f (a ## s1) (b ## s2) = f a b ## (smap2 f s1 s2)*  
 ⟨proof⟩

**lemma** *smap2-szip*:

*smap2 f s1 s2 = smap (case-prod f) (szip s1 s2)*  
 ⟨proof⟩



**lemma** *smap-smap2[simp]*:

$smap\ f\ (smap2\ g\ s1\ s2) = smap2\ (\lambda x\ y.\ f\ (g\ x\ y))\ s1\ s2$   
 ⟨proof⟩

**lemma** *smap2-alt*:

$(smap2\ f\ s1\ s2 = s) = (\forall n.\ f\ (s1\ !!\ n)\ (s2\ !!\ n) = s\ !!\ n)$   
 ⟨proof⟩

**lemma** *snth-smap2[simp]*:

$smap2\ f\ s1\ s2\ !!\ n = f\ (s1\ !!\ n)\ (s2\ !!\ n)$   
 ⟨proof⟩

**lemma** *stake-smap2[simp]*:

$stake\ n\ (smap2\ f\ s1\ s2) = map\ (case-prod\ f)\ (zip\ (stake\ n\ s1)\ (stake\ n\ s2))$   
 ⟨proof⟩

**lemma** *sdrop-smap2[simp]*:

$sdrop\ n\ (smap2\ f\ s1\ s2) = smap2\ f\ (sdrop\ n\ s1)\ (sdrop\ n\ s2)$   
 ⟨proof⟩

end

## 57 List prefixes, suffixes, and homeomorphic embedding

**theory** *Sublist*

**imports** *Main*

**begin**

### 57.1 Prefix order on lists

**definition** *prefix* :: 'a list ⇒ 'a list ⇒ bool

where  $prefix\ xs\ ys \longleftrightarrow (\exists\ zs.\ ys = xs\ @\ zs)$

**definition** *strict-prefix* :: 'a list ⇒ 'a list ⇒ bool

where  $strict-prefix\ xs\ ys \longleftrightarrow prefix\ xs\ ys \wedge xs \neq ys$

**global-interpretation** *prefix-order*: ordering *prefix* *strict-prefix*

⟨proof⟩

**interpretation** *prefix-order*: order *prefix* *strict-prefix*

⟨proof⟩

**global-interpretation** *prefix-bot*: ordering-top ⟨λxs ys. prefix ys xs⟩ ⟨λxs ys. strict-prefix ys xs⟩ ⟨[]⟩

⟨proof⟩

**interpretation** *prefix-bot*: order-bot Nil *prefix* *strict-prefix*

*<proof>*

**lemma** *prefixI* [*intro?*]:  $ys = xs @ zs \implies \text{prefix } xs \ ys$   
*<proof>*

**lemma** *prefixE* [*elim?*]:  
**assumes** *prefix xs ys*  
**obtains** *zs where ys = xs @ zs*  
*<proof>*

**lemma** *strict-prefixI'* [*intro?*]:  $ys = xs @ z \# zs \implies \text{strict-prefix } xs \ ys$   
*<proof>*

**lemma** *strict-prefixE'* [*elim?*]:  
**assumes** *strict-prefix xs ys*  
**obtains** *z zs where ys = xs @ z # zs*  
*<proof>*

**lemma** *strict-prefixI* [*intro?*]:  $\text{prefix } xs \ ys \implies xs \neq ys \implies \text{strict-prefix } xs \ ys$   
*<proof>*

**lemma** *strict-prefixE* [*elim?*]:  
**fixes** *xs ys :: 'a list*  
**assumes** *strict-prefix xs ys*  
**obtains** *prefix xs ys and xs \neq ys*  
*<proof>*

## 57.2 Basic properties of prefixes

**theorem** *Nil-prefix* [*simp*]:  $\text{prefix } [] \ xs$   
*<proof>*

**theorem** *prefix-Nil* [*simp*]:  $(\text{prefix } xs \ []) = (xs = [])$   
*<proof>*

**lemma** *prefix-snoc* [*simp*]:  $\text{prefix } xs \ (ys @ [y]) \longleftrightarrow xs = ys @ [y] \vee \text{prefix } xs \ ys$   
*<proof>*

**lemma** *Cons-prefix-Cons* [*simp*]:  $\text{prefix } (x \# xs) \ (y \# ys) = (x = y \wedge \text{prefix } xs \ ys)$   
*<proof>*

**lemma** *prefix-code* [*code*]:  
 $\text{prefix } [] \ xs \longleftrightarrow \text{True}$   
 $\text{prefix } (x \# xs) \ [] \longleftrightarrow \text{False}$   
 $\text{prefix } (x \# xs) \ (y \# ys) \longleftrightarrow x = y \wedge \text{prefix } xs \ ys$   
*<proof>*

**lemma** *same-prefix-prefix* [simp]:  $\text{prefix } (xs @ ys) (xs @ zs) = \text{prefix } ys \ zs$   
 ⟨proof⟩

**lemma** *same-prefix-nil* [simp]:  $\text{prefix } (xs @ ys) \ [] = (ys = [])$   
 ⟨proof⟩

**lemma** *prefix-prefix* [simp]:  $\text{prefix } xs \ ys \implies \text{prefix } xs \ (ys @ zs)$   
 ⟨proof⟩

**lemma** *append-prefixD*:  $\text{prefix } (xs @ ys) \ zs \implies \text{prefix } xs \ zs$   
 ⟨proof⟩

**theorem** *prefix-Cons*:  $\text{prefix } xs \ (y \# ys) = (xs = [] \vee (\exists zs. xs = y \# zs \wedge \text{prefix } zs \ ys))$   
 ⟨proof⟩

**theorem** *prefix-append*:  
 $\text{prefix } xs \ (ys @ zs) = (\text{prefix } xs \ ys \vee (\exists us. xs = ys @ us \wedge \text{prefix } us \ zs))$   
 ⟨proof⟩

**lemma** *append-one-prefix*:  
 $\text{prefix } xs \ ys \implies \text{length } xs < \text{length } ys \implies \text{prefix } (xs @ [ys ! \text{length } xs]) \ ys$   
 ⟨proof⟩

**theorem** *prefix-length-le*:  $\text{prefix } xs \ ys \implies \text{length } xs \leq \text{length } ys$   
 ⟨proof⟩

**lemma** *prefix-same-cases*:  
 $\text{prefix } (xs_1 :: 'a \ \text{list}) \ ys \implies \text{prefix } xs_2 \ ys \implies \text{prefix } xs_1 \ xs_2 \vee \text{prefix } xs_2 \ xs_1$   
 ⟨proof⟩

**lemma** *prefix-length-prefix*:  
 $\text{prefix } ps \ xs \implies \text{prefix } qs \ xs \implies \text{length } ps \leq \text{length } qs \implies \text{prefix } ps \ qs$   
 ⟨proof⟩

**lemma** *set-mono-prefix*:  $\text{prefix } xs \ ys \implies \text{set } xs \subseteq \text{set } ys$   
 ⟨proof⟩

**lemma** *take-is-prefix*:  $\text{prefix } (\text{take } n \ xs) \ xs$   
 ⟨proof⟩

**lemma** *takeWhile-is-prefix*:  $\text{prefix } (\text{takeWhile } P \ xs) \ xs$   
 ⟨proof⟩

**lemma** *prefixeq-butlast*:  $\text{prefix } (\text{butlast } xs) \ xs$   
 ⟨proof⟩

**lemma** *prefix-map-rightE*:  
 assumes  $\text{prefix } xs \ (\text{map } f \ ys)$

**shows**  $\exists xs'. \text{prefix } xs' \text{ } ys \wedge xs = \text{map } f \text{ } xs'$   
 ⟨proof⟩

**lemma** *map-mono-prefix*:  $\text{prefix } xs \text{ } ys \implies \text{prefix } (\text{map } f \text{ } xs) (\text{map } f \text{ } ys)$   
 ⟨proof⟩

**lemma** *filter-mono-prefix*:  $\text{prefix } xs \text{ } ys \implies \text{prefix } (\text{filter } P \text{ } xs) (\text{filter } P \text{ } ys)$   
 ⟨proof⟩

**lemma** *sorted-antimono-prefix*:  $\text{prefix } xs \text{ } ys \implies \text{sorted } ys \implies \text{sorted } xs$   
 ⟨proof⟩

**lemma** *prefix-length-less*:  $\text{strict-prefix } xs \text{ } ys \implies \text{length } xs < \text{length } ys$   
 ⟨proof⟩

**lemma** *prefix-snocD*:  $\text{prefix } (xs@[x]) \text{ } ys \implies \text{strict-prefix } xs \text{ } ys$   
 ⟨proof⟩

**lemma** *strict-prefix-simps* [*simp*, *code*]:  
 $\text{strict-prefix } xs \text{ } [] \longleftrightarrow \text{False}$   
 $\text{strict-prefix } [] \text{ } (x \# xs) \longleftrightarrow \text{True}$   
 $\text{strict-prefix } (x \# xs) \text{ } (y \# ys) \longleftrightarrow x = y \wedge \text{strict-prefix } xs \text{ } ys$   
 ⟨proof⟩

**lemma** *take-strict-prefix*:  $\text{strict-prefix } xs \text{ } ys \implies \text{strict-prefix } (\text{take } n \text{ } xs) \text{ } ys$   
 ⟨proof⟩

**lemma** *prefix-takeWhile*:  
**assumes**  $\text{prefix } xs \text{ } ys$   
**shows**  $\text{prefix } (\text{takeWhile } P \text{ } xs) (\text{takeWhile } P \text{ } ys)$   
 ⟨proof⟩

**lemma** *prefix-dropWhile*:  
**assumes**  $\text{prefix } xs \text{ } ys$   
**shows**  $\text{prefix } (\text{dropWhile } P \text{ } xs) (\text{dropWhile } P \text{ } ys)$   
 ⟨proof⟩

**lemma** *prefix-remdups-adj*:  
**assumes**  $\text{prefix } xs \text{ } ys$   
**shows**  $\text{prefix } (\text{remdups-adj } xs) (\text{remdups-adj } ys)$   
 ⟨proof⟩

**lemma** *not-prefix-cases*:  
**assumes**  $\text{pfx}: \neg \text{prefix } ps \text{ } ls$   
**obtains**  
 (c1)  $ps \neq []$  **and**  $ls = []$   
 | (c2)  $a \text{ } as \text{ } x \text{ } xs$  **where**  $ps = a \# as$  **and**  $ls = x \# xs$  **and**  $x = a$  **and**  $\neg \text{prefix } as \text{ } xs$   
 | (c3)  $a \text{ } as \text{ } x \text{ } xs$  **where**  $ps = a \# as$  **and**  $ls = x \# xs$  **and**  $x \neq a$   
 ⟨proof⟩

**lemma** *not-prefix-induct* [consumes 1, case-names Nil Neq Eq]:

**assumes** *np*:  $\neg \text{prefix } ps \ ls$   
**and** *base*:  $\bigwedge x \ xs. P \ (x\#xs) \ []$   
**and** *r1*:  $\bigwedge x \ xs \ y \ ys. x \neq y \implies P \ (x\#xs) \ (y\#ys)$   
**and** *r2*:  $\bigwedge x \ xs \ y \ ys. [x = y; \neg \text{prefix } xs \ ys; P \ xs \ ys] \implies P \ (x\#xs) \ (y\#ys)$   
**shows**  $P \ ps \ ls$  *<proof>*

### 57.3 Prefixes

**primrec** *prefixes* **where**

*prefixes* [] = [[]] |  
*prefixes* (x#xs) = [] # map ((#) x) (*prefixes* xs)

**lemma** *in-set-prefixes[simp]*:  $xs \in \text{set} \ (\text{prefixes } ys) \longleftrightarrow \text{prefix } xs \ ys$   
*<proof>*

**lemma** *length-prefixes[simp]*:  $\text{length} \ (\text{prefixes } xs) = \text{length } xs + 1$   
*<proof>*

**lemma** *distinct-prefixes [intro]*:  $\text{distinct} \ (\text{prefixes } xs)$   
*<proof>*

**lemma** *prefixes-snoc [simp]*:  $\text{prefixes} \ (xs@[x]) = \text{prefixes } xs \ @ \ [xs@[x]]$   
*<proof>*

**lemma** *prefixes-not-Nil [simp]*:  $\text{prefixes } xs \neq []$   
*<proof>*

**lemma** *hd-prefixes [simp]*:  $\text{hd} \ (\text{prefixes } xs) = []$   
*<proof>*

**lemma** *last-prefixes [simp]*:  $\text{last} \ (\text{prefixes } xs) = xs$   
*<proof>*

**lemma** *prefixes-append*:

$\text{prefixes} \ (xs \ @ \ ys) = \text{prefixes } xs \ @ \ \text{map} \ (\lambda ys'. xs \ @ \ ys') \ (\text{tl} \ (\text{prefixes } ys))$   
*<proof>*

**lemma** *prefixes-eq-snoc*:

$\text{prefixes } ys = xs \ @ \ [x] \longleftrightarrow$   
 $(ys = [] \wedge xs = [] \vee (\exists z \ zs. ys = zs@[z] \wedge xs = \text{prefixes } zs)) \wedge x = ys$   
*<proof>*

**lemma** *prefixes-tailrec [code]*:

$\text{prefixes } xs = \text{rev} \ (\text{snd} \ (\text{foldl} \ (\lambda(\text{acc1}, \text{acc2}) \ x. (x\#\text{acc1}, \text{rev} \ (x\#\text{acc1})\#\text{acc2}))$   
 $([], []) \ xs))$   
*<proof>*

**lemma** *set-prefixes-eq*:  $set (prefixes\ xs) = \{ys.\ prefix\ ys\ xs\}$   
 ⟨proof⟩

**lemma** *card-set-prefixes [simp]*:  $card (set (prefixes\ xs)) = Suc (length\ xs)$   
 ⟨proof⟩

**lemma** *set-prefixes-append*:  
 $set (prefixes (xs @ ys)) = set (prefixes\ xs) \cup \{xs @ ys' \mid ys' \in set (prefixes\ ys)\}$   
 ⟨proof⟩

## 57.4 Longest Common Prefix

**definition** *Longest-common-prefix* :: 'a list set  $\Rightarrow$  'a list **where**  
*Longest-common-prefix*  $L = (ARG-MAX\ length\ ps.\ \forall xs \in L.\ prefix\ ps\ xs)$

**lemma** *Longest-common-prefix-ex*:  $L \neq \{\} \Longrightarrow$   
 $\exists ps.\ (\forall xs \in L.\ prefix\ ps\ xs) \wedge (\forall qs.\ (\forall xs \in L.\ prefix\ qs\ xs) \longrightarrow size\ qs \leq size\ ps)$   
 (is -  $\Longrightarrow \exists ps.\ ?P\ L\ ps$ )  
 ⟨proof⟩

**lemma** *Longest-common-prefix-unique*:  
 $\langle \exists! ps.\ (\forall xs \in L.\ prefix\ ps\ xs) \wedge (\forall qs.\ (\forall xs \in L.\ prefix\ qs\ xs) \longrightarrow length\ qs \leq length\ ps) \rangle$   
 if  $\langle L \neq \{\} \rangle$   
 ⟨proof⟩

**lemma** *Longest-common-prefix-eq*:  
 $\llbracket L \neq \{\};\ \forall xs \in L.\ prefix\ ps\ xs;\ \forall qs.\ (\forall xs \in L.\ prefix\ qs\ xs) \longrightarrow size\ qs \leq size\ ps \rrbracket$   
 $\Longrightarrow Longest-common-prefix\ L = ps$   
 ⟨proof⟩

**lemma** *Longest-common-prefix-prefix*:  
 $xs \in L \Longrightarrow prefix (Longest-common-prefix\ L)\ xs$   
 ⟨proof⟩

**lemma** *Longest-common-prefix-longest*:  
 $L \neq \{\} \Longrightarrow \forall xs \in L.\ prefix\ ps\ xs \Longrightarrow length\ ps \leq length (Longest-common-prefix\ L)$   
 ⟨proof⟩

**lemma** *Longest-common-prefix-max-prefix*:  
 $L \neq \{\} \Longrightarrow \forall xs \in L.\ prefix\ ps\ xs \Longrightarrow prefix\ ps (Longest-common-prefix\ L)$   
 ⟨proof⟩

**lemma** *Longest-common-prefix-Nil*:  $\llbracket \in L \rrbracket \Longrightarrow Longest-common-prefix\ L = \llbracket \rrbracket$   
 ⟨proof⟩

**lemma** *Longest-common-prefix-image-Cons*:  $L \neq \{\} \implies$   
*Longest-common-prefix*  $((\#) x \text{ ' } L) = x \#$  *Longest-common-prefix*  $L$   
 ⟨proof⟩

**lemma** *Longest-common-prefix-eq-Cons*: **assumes**  $L \neq \{\} \ \ [] \notin L \ \ \forall xs \in L. \text{hd } xs = x$   
**shows** *Longest-common-prefix*  $L = x \#$  *Longest-common-prefix*  $\{ys. x \# ys \in L\}$   
 ⟨proof⟩

**lemma** *Longest-common-prefix-eq-Nil*:  
 $[[x \# ys \in L; y \# zs \in L; x \neq y]] \implies$  *Longest-common-prefix*  $L = []$   
 ⟨proof⟩

**fun** *longest-common-prefix* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list **where**  
*longest-common-prefix*  $(x \# xs) (y \# ys) =$   
 (if  $x=y$  then  $x \#$  *longest-common-prefix*  $xs \ ys$  else  $[]$ ) |  
*longest-common-prefix* - - =  $[]$

**lemma** *longest-common-prefix-prefix1*:  
*prefix* (*longest-common-prefix*  $xs \ ys$ )  $xs$   
 ⟨proof⟩

**lemma** *longest-common-prefix-prefix2*:  
*prefix* (*longest-common-prefix*  $xs \ ys$ )  $ys$   
 ⟨proof⟩

**lemma** *longest-common-prefix-max-prefix*:  
 $[[\text{prefix } ps \ xs; \text{prefix } ps \ ys]]$   
 $\implies$  *prefix*  $ps$  (*longest-common-prefix*  $xs \ ys$ )  
 ⟨proof⟩

## 57.5 Parallel lists

**definition** *parallel* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool (**infixl**  $\langle || \rangle$  50)  
**where**  $(xs \ || \ ys) = (\neg \text{prefix } xs \ ys \wedge \neg \text{prefix } ys \ xs)$

**lemma** *parallelI* [intro]:  $\neg \text{prefix } xs \ ys \implies \neg \text{prefix } ys \ xs \implies xs \ || \ ys$   
 ⟨proof⟩

**lemma** *parallelE* [elim]:  
**assumes**  $xs \ || \ ys$   
**obtains**  $\neg \text{prefix } xs \ ys \wedge \neg \text{prefix } ys \ xs$   
 ⟨proof⟩

**theorem** *prefix-cases*:  
**obtains**  $\text{prefix } xs \ ys \ | \ \text{strict-prefix } ys \ xs \ | \ xs \ || \ ys$   
 ⟨proof⟩

**lemma** *parallel-cancel*:  $a\#xs \parallel a\#ys \implies xs \parallel ys$   
 ⟨proof⟩

**theorem** *parallel-decomp*:  
 $xs \parallel ys \implies \exists as\ b\ bs\ c\ cs. b \neq c \wedge xs = as @ b \# bs \wedge ys = as @ c \# cs$   
 ⟨proof⟩

**lemma** *parallel-append*:  $a \parallel b \implies a @ c \parallel b @ d$   
 ⟨proof⟩

**lemma** *parallel-appendI*:  $xs \parallel ys \implies x = xs @ xs' \implies y = ys @ ys' \implies x \parallel y$   
 ⟨proof⟩

**lemma** *parallel-commute*:  $a \parallel b \longleftrightarrow b \parallel a$   
 ⟨proof⟩

## 57.6 Suffix order on lists

**definition** *suffix* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool  
 where *suffix*  $xs\ ys = (\exists zs. ys = zs @ xs)$

**definition** *strict-suffix* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool  
 where *strict-suffix*  $xs\ ys \longleftrightarrow \text{suffix } xs\ ys \wedge xs \neq ys$

**global-interpretation** *suffix-order*: ordering *suffix* *strict-suffix*  
 ⟨proof⟩

**interpretation** *suffix-order*: order *suffix* *strict-suffix*  
 ⟨proof⟩

**global-interpretation** *suffix-bot*: ordering-top  $\langle \lambda xs\ ys. \text{suffix } ys\ xs \rangle \langle \lambda xs\ ys. \text{strict-suffix } ys\ xs \rangle \langle [] \rangle$   
 ⟨proof⟩

**interpretation** *suffix-bot*: order-bot Nil *suffix* *strict-suffix*  
 ⟨proof⟩

**lemma** *suffixI* [*intro?*]:  $ys = zs @ xs \implies \text{suffix } xs\ ys$   
 ⟨proof⟩

**lemma** *suffixE* [*elim?*]:  
 assumes *suffix*  $xs\ ys$   
 obtains  $zs$  where  $ys = zs @ xs$   
 ⟨proof⟩

**lemma** *suffix-tl* [*simp*]: *suffix* (tl  $xs$ )  $xs$   
 ⟨proof⟩

**lemma** *strict-suffix-tl* [*simp*]:  $xs \neq [] \implies \text{strict-suffix } (tl\ xs)\ xs$



*<proof>*

**lemma** *Nil-suffix* [simp]:  $\text{suffix } [] \ xs$   
*<proof>*

**lemma** *suffix-Nil* [simp]:  $(\text{suffix } xs \ []) = (xs = [])$   
*<proof>*

**lemma** *suffix-ConsI*:  $\text{suffix } xs \ ys \implies \text{suffix } xs \ (y \# \ ys)$   
*<proof>*

**lemma** *suffix-ConsD*:  $\text{suffix } (x \# \ xs) \ ys \implies \text{suffix } xs \ ys$   
*<proof>*

**lemma** *suffix-appendI*:  $\text{suffix } xs \ ys \implies \text{suffix } xs \ (zs \ @ \ ys)$   
*<proof>*

**lemma** *suffix-appendD*:  $\text{suffix } (zs \ @ \ xs) \ ys \implies \text{suffix } xs \ ys$   
*<proof>*

**lemma** *strict-suffix-set-subset*:  $\text{strict-suffix } xs \ ys \implies \text{set } xs \subseteq \text{set } ys$   
*<proof>*

**lemma** *set-mono-suffix*:  $\text{suffix } xs \ ys \implies \text{set } xs \subseteq \text{set } ys$   
*<proof>*

**lemma** *sorted-antimono-suffix*:  $\text{suffix } xs \ ys \implies \text{sorted } ys \implies \text{sorted } xs$   
*<proof>*

**lemma** *suffix-ConsD2*:  $\text{suffix } (x \# \ xs) \ (y \# \ ys) \implies \text{suffix } xs \ ys$   
*<proof>*

**lemma** *suffix-to-prefix* [code]:  $\text{suffix } xs \ ys \longleftrightarrow \text{prefix } (\text{rev } xs) \ (\text{rev } ys)$   
*<proof>*

**lemma** *strict-suffix-to-prefix* [code]:  $\text{strict-suffix } xs \ ys \longleftrightarrow \text{strict-prefix } (\text{rev } xs) \ (\text{rev } ys)$   
*<proof>*

**lemma** *distinct-suffix*:  $\text{distinct } ys \implies \text{suffix } xs \ ys \implies \text{distinct } xs$   
*<proof>*

**lemma** *map-mono-suffix*:  $\text{suffix } xs \ ys \implies \text{suffix } (\text{map } f \ xs) \ (\text{map } f \ ys)$   
*<proof>*

**lemma** *map-mono-strict-suffix*:  $\text{strict-suffix } xs \ ys \implies \text{strict-suffix } (\text{map } f \ xs) \ (\text{map } f \ ys)$   
*<proof>*

**lemma** *filter-mono-suffix*:  $\text{suffix } xs \ ys \implies \text{suffix } (\text{filter } P \ xs) \ (\text{filter } P \ ys)$   
 ⟨proof⟩

**lemma** *suffix-drop*:  $\text{suffix } (\text{drop } n \ as) \ as$   
 ⟨proof⟩

**lemma** *suffix-dropWhile*:  $\text{suffix } (\text{dropWhile } P \ xs) \ xs$   
 ⟨proof⟩

**lemma** *suffix-take*:  $\text{suffix } xs \ ys \implies ys = \text{take } (\text{length } ys - \text{length } xs) \ ys \ @ \ xs$   
 ⟨proof⟩

**lemma** *strict-suffix-reflclp-conv*:  $\text{strict-suffix}^{==} = \text{suffix}$   
 ⟨proof⟩

**lemma** *suffix-lists*:  $\text{suffix } xs \ ys \implies ys \in \text{lists } A \implies xs \in \text{lists } A$   
 ⟨proof⟩

**lemma** *suffix-snoc [simp]*:  $\text{suffix } xs \ (ys \ @ \ [y]) \longleftrightarrow xs = [] \vee (\exists zs. xs = zs \ @ \ [y] \wedge \text{suffix } zs \ ys)$   
 ⟨proof⟩

**lemma** *snoc-suffix-snoc [simp]*:  $\text{suffix } (xs \ @ \ [x]) \ (ys \ @ \ [y]) = (x = y \wedge \text{suffix } xs \ ys)$   
 ⟨proof⟩

**lemma** *same-suffix-suffix [simp]*:  $\text{suffix } (ys \ @ \ xs) \ (zs \ @ \ xs) = \text{suffix } ys \ zs$   
 ⟨proof⟩

**lemma** *same-suffix-nil [simp]*:  $\text{suffix } (ys \ @ \ xs) \ xs = (ys = [])$   
 ⟨proof⟩

**theorem** *suffix-Cons*:  $\text{suffix } xs \ (y \ # \ ys) \longleftrightarrow xs = y \ # \ ys \vee \text{suffix } xs \ ys$   
 ⟨proof⟩

**theorem** *suffix-append*:  
 $\text{suffix } xs \ (ys \ @ \ zs) \longleftrightarrow \text{suffix } xs \ zs \vee (\exists xs'. xs = xs' \ @ \ zs \wedge \text{suffix } xs' \ ys)$   
 ⟨proof⟩

**theorem** *suffix-length-le*:  $\text{suffix } xs \ ys \implies \text{length } xs \leq \text{length } ys$   
 ⟨proof⟩

**lemma** *suffix-same-cases*:  
 $\text{suffix } (xs_1 :: 'a \ \text{list}) \ ys \implies \text{suffix } xs_2 \ ys \implies \text{suffix } xs_1 \ xs_2 \vee \text{suffix } xs_2 \ xs_1$   
 ⟨proof⟩

**lemma** *suffix-length-suffix*:  
 $\text{suffix } ps \ xs \implies \text{suffix } qs \ xs \implies \text{length } ps \leq \text{length } qs \implies \text{suffix } ps \ qs$   
 ⟨proof⟩

**lemma** *suffix-length-less*: *strict-suffix xs ys*  $\implies$  *length xs* < *length ys*  
 ⟨*proof*⟩

**lemma** *suffix-ConsD'*: *suffix (x#xs) ys*  $\implies$  *strict-suffix xs ys*  
 ⟨*proof*⟩

**lemma** *drop-strict-suffix*: *strict-suffix xs ys*  $\implies$  *strict-suffix (drop n xs) ys*  
 ⟨*proof*⟩

**lemma** *suffix-map-rightE*:  
**assumes** *suffix xs (map f ys)*  
**shows**  $\exists xs'. \text{suffix } xs' \text{ ys} \wedge xs = \text{map } f \text{ } xs'$   
 ⟨*proof*⟩

**lemma** *suffix-remdups-adj*: *suffix xs ys*  $\implies$  *suffix (remdups-adj xs) (remdups-adj ys)*  
 ⟨*proof*⟩

**lemma** *not-suffix-cases*:  
**assumes** *pf*:  $\neg \text{suffix } ps \text{ } ls$   
**obtains**  
 (c1) *ps*  $\neq []$  **and** *ls* = []  
 | (c2) *a as x xs* **where** *ps* = *as@[a]* **and** *ls* = *xs@[x]* **and** *x* = *a* **and**  $\neg \text{suffix } as \text{ } xs$   
 | (c3) *a as x xs* **where** *ps* = *as@[a]* **and** *ls* = *xs@[x]* **and** *x*  $\neq a$   
 ⟨*proof*⟩

**lemma** *not-suffix-induct* [*consumes 1*, *case-names Nil Neq Eq*]:  
**assumes** *np*:  $\neg \text{suffix } ps \text{ } ls$   
**and** *base*:  $\bigwedge x \text{ } xs. P \text{ } (xs@[x]) []$   
**and** *r1*:  $\bigwedge x \text{ } xs \text{ } y \text{ } ys. x \neq y \implies P \text{ } (xs@[x]) (ys@[y])$   
**and** *r2*:  $\bigwedge x \text{ } xs \text{ } y \text{ } ys. \llbracket x = y; \neg \text{suffix } xs \text{ } ys; P \text{ } xs \text{ } ys \rrbracket \implies P \text{ } (xs@[x]) (ys@[y])$   
**shows** *P ps ls* ⟨*proof*⟩

**lemma** *parallelD1*: *x*  $\parallel$  *y*  $\implies \neg \text{prefix } x \text{ } y$   
 ⟨*proof*⟩

**lemma** *parallelD2*: *x*  $\parallel$  *y*  $\implies \neg \text{prefix } y \text{ } x$   
 ⟨*proof*⟩

**lemma** *parallel-Nil1* [*simp*]:  $\neg x \parallel []$   
 ⟨*proof*⟩

**lemma** *parallel-Nil2* [*simp*]:  $\neg [] \parallel x$   
 ⟨*proof*⟩

**lemma** *Cons-parallelI1*: *a*  $\neq b \implies a \# as \parallel b \# bs$

*<proof>*

**lemma** *Cons-parallelI2*:  $\llbracket a = b; as \parallel bs \rrbracket \implies a \# as \parallel b \# bs$   
*<proof>*

**lemma** *not-equal-is-parallel*:  
**assumes** *neg*:  $xs \neq ys$   
**and** *len*:  $length\ xs = length\ ys$   
**shows**  $xs \parallel ys$   
*<proof>*

## 57.7 Suffixes

**primrec** *suffixes* **where**  
 $suffixes\ [] = [[]]$   
 $| suffixes\ (x\#\ xs) = suffixes\ xs\ @\ [x\ \#\ xs]$

**lemma** *in-set-suffixes* [*simp*]:  $xs \in set\ (suffixes\ ys) \longleftrightarrow suffix\ xs\ ys$   
*<proof>*

**lemma** *distinct-suffixes* [*intro*]:  $distinct\ (suffixes\ xs)$   
*<proof>*

**lemma** *length-suffixes* [*simp*]:  $length\ (suffixes\ xs) = Suc\ (length\ xs)$   
*<proof>*

**lemma** *suffixes-snoc* [*simp*]:  $suffixes\ (xs\ @\ [x]) = []\ \#\ map\ (\lambda ys.\ ys\ @\ [x])\ (suffixes\ xs)$   
*<proof>*

**lemma** *suffixes-not-Nil* [*simp*]:  $suffixes\ xs \neq []$   
*<proof>*

**lemma** *hd-suffixes* [*simp*]:  $hd\ (suffixes\ xs) = []$   
*<proof>*

**lemma** *last-suffixes* [*simp*]:  $last\ (suffixes\ xs) = xs$   
*<proof>*

**lemma** *suffixes-append*:  
 $suffixes\ (xs\ @\ ys) = suffixes\ ys\ @\ map\ (\lambda xs'.\ xs'\ @\ ys)\ (tl\ (suffixes\ xs))$   
*<proof>*

**lemma** *suffixes-eq-snoc*:  
 $suffixes\ ys = xs\ @\ [x] \longleftrightarrow$   
 $(ys = [] \wedge xs = [] \vee (\exists z\ zs.\ ys = z\#\ zs \wedge xs = suffixes\ zs)) \wedge x = ys$   
*<proof>*

**lemma** *suffixes-tailrec* [*code*]:

$suffixes\ xs = rev\ (snd\ (foldl\ (\lambda(acc1, acc2)\ x.\ (x\#\ acc1, (x\#\ acc1)\#\ acc2))\ ([], []))\ (rev\ xs))$   
 <proof>

**lemma** *set-suffixes-eq*:  $set\ (suffixes\ xs) = \{ys.\ suffix\ ys\ xs\}$   
 <proof>

**lemma** *card-set-suffixes* [simp]:  $card\ (set\ (suffixes\ xs)) = Suc\ (length\ xs)$   
 <proof>

**lemma** *set-suffixes-append*:  
 $set\ (suffixes\ (xs\ @\ ys)) = set\ (suffixes\ ys) \cup \{xs'\ @\ ys\ |\ xs'.\ xs' \in set\ (suffixes\ xs)\}$   
 <proof>

**lemma** *suffixes-conv-prefixes*:  $suffixes\ xs = map\ rev\ (prefixes\ (rev\ xs))$   
 <proof>

**lemma** *prefixes-conv-suffixes*:  $prefixes\ xs = map\ rev\ (suffixes\ (rev\ xs))$   
 <proof>

**lemma** *prefixes-rev*:  $prefixes\ (rev\ xs) = map\ rev\ (suffixes\ xs)$   
 <proof>

**lemma** *suffixes-rev*:  $suffixes\ (rev\ xs) = map\ rev\ (prefixes\ xs)$   
 <proof>

## 57.8 Homeomorphic embedding on lists

**inductive** *list-emb* :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool  
 for *P* :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)

**where**

*list-emb-Nil* [intro, simp]:  $list-emb\ P\ []\ ys$   
 | *list-emb-Cons* [intro]:  $list-emb\ P\ xs\ ys \Longrightarrow list-emb\ P\ xs\ (y\#\ ys)$   
 | *list-emb-Cons2* [intro]:  $P\ x\ y \Longrightarrow list-emb\ P\ xs\ ys \Longrightarrow list-emb\ P\ (x\#\ xs)\ (y\#\ ys)$

**lemma** *list-emb-mono*:  
 assumes  $\bigwedge x\ y.\ P\ x\ y \longrightarrow Q\ x\ y$   
 shows  $list-emb\ P\ xs\ ys \longrightarrow list-emb\ Q\ xs\ ys$   
 <proof>

**lemma** *list-emb-Nil2* [simp]:  
 assumes  $list-emb\ P\ xs\ []$  shows  $xs = []$   
 <proof>

**lemma** *list-emb-refl*:  
 assumes  $\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x\ x$   
 shows  $list-emb\ P\ xs\ xs$

⟨proof⟩

**lemma** *list-emb-Cons-Nil* [simp]:  $list-emb\ P\ (x\#\!xs)\ [] = False$   
 ⟨proof⟩

**lemma** *list-emb-append2* [intro]:  $list-emb\ P\ xs\ ys \implies list-emb\ P\ xs\ (zs\ @\ ys)$   
 ⟨proof⟩

**lemma** *list-emb-prefix* [intro]:  
**assumes**  $list-emb\ P\ xs\ ys$  **shows**  $list-emb\ P\ xs\ (ys\ @\ zs)$   
 ⟨proof⟩

**lemma** *list-emb-ConsD*:  
**assumes**  $list-emb\ P\ (x\#\!xs)\ ys$   
**shows**  $\exists\ us\ v\ vs.\ ys = us\ @\ v\ \#\! vs \wedge P\ x\ v \wedge list-emb\ P\ xs\ vs$   
 ⟨proof⟩

**lemma** *list-emb-appendD*:  
**assumes**  $list-emb\ P\ (xs\ @\ ys)\ zs$   
**shows**  $\exists\ us\ vs.\ zs = us\ @\ vs \wedge list-emb\ P\ xs\ us \wedge list-emb\ P\ ys\ vs$   
 ⟨proof⟩

**lemma** *list-emb-strict-suffix*:  
**assumes**  $list-emb\ P\ xs\ ys$  **and** *strict-suffix*  $ys\ zs$   
**shows**  $list-emb\ P\ xs\ zs$   
 ⟨proof⟩

**lemma** *list-emb-suffix*:  
**assumes**  $list-emb\ P\ xs\ ys$  **and** *suffix*  $ys\ zs$   
**shows**  $list-emb\ P\ xs\ zs$   
 ⟨proof⟩

**lemma** *list-emb-length*:  $list-emb\ P\ xs\ ys \implies length\ xs \leq length\ ys$   
 ⟨proof⟩

**lemma** *list-emb-trans*:  
**assumes**  $\bigwedge x\ y\ z.\ [x \in set\ xs; y \in set\ ys; z \in set\ zs; P\ x\ y; P\ y\ z] \implies P\ x\ z$   
**shows**  $[list-emb\ P\ xs\ ys; list-emb\ P\ ys\ zs] \implies list-emb\ P\ xs\ zs$   
 ⟨proof⟩

**lemma** *list-emb-set*:  
**assumes**  $list-emb\ P\ xs\ ys$  **and**  $x \in set\ xs$   
**obtains**  $y$  **where**  $y \in set\ ys$  **and**  $P\ x\ y$   
 ⟨proof⟩

**lemma** *list-emb-Cons-iff1* [simp]:  
**assumes**  $P\ x\ y$   
**shows**  $list-emb\ P\ (x\#\!xs)\ (y\#\!ys) \longleftrightarrow list-emb\ P\ xs\ ys$   
 ⟨proof⟩

**lemma** *list-emb-Cons-iff2* [simp]:

**assumes**  $\neg P\ x\ y$

**shows**  $list-emb\ P\ (x\#xs)\ (y\#ys) \longleftrightarrow list-emb\ P\ (x\#xs)\ ys$

*<proof>*

**lemma** *list-emb-code* [code]:

$list-emb\ P\ []\ ys \longleftrightarrow True$

$list-emb\ P\ (x\#xs)\ [] \longleftrightarrow False$

$list-emb\ P\ (x\#xs)\ (y\#ys) \longleftrightarrow (if\ P\ x\ y\ then\ list-emb\ P\ xs\ ys\ else\ list-emb\ P\ (x\#xs)\ ys)$

*<proof>*

## 57.9 Subsequences (special case of homeomorphic embedding)

**abbreviation** *subseq* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool

**where**  $subseq\ xs\ ys \equiv list-emb\ (=)\ xs\ ys$

**definition** *strict-subseq* **where**  $strict-subseq\ xs\ ys \longleftrightarrow xs \neq ys \wedge subseq\ xs\ ys$

**lemma** *subseq-Cons2*:  $subseq\ xs\ ys \Longrightarrow subseq\ (x\#xs)\ (x\#ys)$  *<proof>*

**lemma** *subseq-same-length*:

**assumes**  $subseq\ xs\ ys$  **and**  $length\ xs = length\ ys$  **shows**  $xs = ys$

*<proof>*

**lemma** *not-subseq-length* [simp]:  $length\ ys < length\ xs \Longrightarrow \neg subseq\ xs\ ys$

*<proof>*

**lemma** *subseq-Cons'*:  $subseq\ (x\#xs)\ ys \Longrightarrow subseq\ xs\ ys$

*<proof>*

**lemma** *subseq-Cons2'*:

**assumes**  $subseq\ (x\#xs)\ (x\#ys)$  **shows**  $subseq\ xs\ ys$

*<proof>*

**lemma** *subseq-Cons2-neg*:

**assumes**  $subseq\ (x\#xs)\ (y\#ys)$

**shows**  $x \neq y \Longrightarrow subseq\ (x\#xs)\ ys$

*<proof>*

**lemma** *subseq-Cons2-iff* [simp]:

$subseq\ (x\#xs)\ (y\#ys) = (if\ x = y\ then\ subseq\ xs\ ys\ else\ subseq\ (x\#xs)\ ys)$

*<proof>*

**lemma** *subseq-append'*:  $subseq\ (zs\ @\ xs)\ (zs\ @\ ys) \longleftrightarrow subseq\ xs\ ys$

*<proof>*

**global-interpretation** *subseq-order: ordering subseq strict-subseq*  
 ⟨proof⟩

**interpretation** *subseq-order: order subseq strict-subseq*  
 ⟨proof⟩

**lemma** *in-set-subseqs [simp]:*  $xs \in \text{set } (\text{subseqs } ys) \longleftrightarrow \text{subseq } xs \text{ } ys$   
 ⟨proof⟩

**lemma** *set-subseqs-eq:*  $\text{set } (\text{subseqs } ys) = \{xs. \text{subseq } xs \text{ } ys\}$   
 ⟨proof⟩

**lemma** *subseq-append-le-same-iff:*  $\text{subseq } (xs @ ys) \text{ } ys \longleftrightarrow xs = []$   
 ⟨proof⟩

**lemma** *subseq-singleton-left:*  $\text{subseq } [x] \text{ } ys \longleftrightarrow x \in \text{set } ys$   
 ⟨proof⟩

**lemma** *list-emb-append-mono:*  
 $\llbracket \text{list-emb } P \text{ } xs \text{ } xs'; \text{list-emb } P \text{ } ys \text{ } ys' \rrbracket \implies \text{list-emb } P \text{ } (xs @ ys) \text{ } (xs' @ ys')$   
 ⟨proof⟩

**lemma** *prefix-imp-subseq [intro]:*  $\text{prefix } xs \text{ } ys \implies \text{subseq } xs \text{ } ys$   
 ⟨proof⟩

**lemma** *suffix-imp-subseq [intro]:*  $\text{suffix } xs \text{ } ys \implies \text{subseq } xs \text{ } ys$   
 ⟨proof⟩

## 57.10 Appending elements

**lemma** *subseq-append [simp]:*  
 $\text{subseq } (xs @ zs) \text{ } (ys @ zs) \longleftrightarrow \text{subseq } xs \text{ } ys \text{ (is ?l = ?r)}$   
 ⟨proof⟩

**lemma** *subseq-append-iff:*  
 $\text{subseq } xs \text{ } (ys @ zs) \longleftrightarrow (\exists xs1 \text{ } xs2. xs = xs1 @ xs2 \wedge \text{subseq } xs1 \text{ } ys \wedge \text{subseq } xs2 \text{ } zs)$   
 (is ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *subseq-appendE [case-names append]:*  
**assumes**  $\text{subseq } xs \text{ } (ys @ zs)$   
**obtains**  $xs1 \text{ } xs2$  **where**  $xs = xs1 @ xs2 \text{subseq } xs1 \text{ } ys \text{subseq } xs2 \text{ } zs$   
 ⟨proof⟩

**lemma** *subseq-drop-many:*  $\text{subseq } xs \text{ } ys \implies \text{subseq } xs \text{ } (zs @ ys)$   
 ⟨proof⟩

**lemma** *subseq-rev-drop-many:*  $\text{subseq } xs \text{ } ys \implies \text{subseq } xs \text{ } (ys @ zs)$



*<proof>*

### 57.11 Relation to standard list operations

**lemma** *subseq-map*:

**assumes** *subseq xs ys* **shows** *subseq (map f xs) (map f ys)*  
*<proof>*

**lemma** *subseq-filter-left [simp]*: *subseq (filter P xs) xs*

*<proof>*

**lemma** *subseq-filter [simp]*:

**assumes** *subseq xs ys* **shows** *subseq (filter P xs) (filter P ys)*  
*<proof>*

**lemma** *subseq-conv-nths*: *subseq xs ys*  $\longleftrightarrow$   $(\exists N. xs = nth\ ys\ N)$

(**is** *?L = ?R*)

*<proof>*

### 57.12 Contiguous sublists

#### 57.12.1 *sublist*

**definition** *sublist* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool **where**

*sublist xs ys* =  $(\exists ps\ ss. ys = ps @ xs @ ss)$

**definition** *strict-sublist* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool **where**

*strict-sublist xs ys*  $\longleftrightarrow$  *sublist xs ys*  $\wedge$  *xs*  $\neq$  *ys*

**interpretation** *sublist-order*: *order sublist strict-sublist*

*<proof>*

**lemma** *sublist-Nil-left [simp, intro]*: *sublist [] ys*

*<proof>*

**lemma** *sublist-Cons-Nil [simp]*:  $\neg$ *sublist (x#xs) []*

*<proof>*

**lemma** *sublist-Nil-right [simp]*: *sublist xs []*  $\longleftrightarrow$  *xs* = []

*<proof>*

**lemma** *sublist-appendI [simp, intro]*: *sublist xs (ps @ xs @ ss)*

*<proof>*

**lemma** *sublist-append-leftI [simp, intro]*: *sublist xs (ps @ xs)*

*<proof>*

**lemma** *sublist-append-rightI [simp, intro]*: *sublist xs (xs @ ss)*

*<proof>*

**lemma** *sublist-altdef*:  $sublist\ xs\ ys \longleftrightarrow (\exists\ ys'.\ prefix\ ys'\ ys \wedge suffix\ xs\ ys')$   
 ⟨proof⟩

**lemma** *sublist-altdef'*:  $sublist\ xs\ ys \longleftrightarrow (\exists\ ys'.\ suffix\ ys'\ ys \wedge prefix\ xs\ ys')$   
 ⟨proof⟩

**lemma** *sublist-Cons-right*:  $sublist\ xs\ (y\ \# \ ys) \longleftrightarrow prefix\ xs\ (y\ \# \ ys) \vee sublist\ xs\ ys$   
 ⟨proof⟩

**lemma** *sublist-code* [code]:  
 $sublist\ []\ ys \longleftrightarrow True$   
 $sublist\ (x\ \# \ xs)\ [] \longleftrightarrow False$   
 $sublist\ (x\ \# \ xs)\ (y\ \# \ ys) \longleftrightarrow prefix\ (x\ \# \ xs)\ (y\ \# \ ys) \vee sublist\ (x\ \# \ xs)\ ys$   
 ⟨proof⟩

**lemma** *sublist-append*:  
 $sublist\ xs\ (ys\ @ \ zs) \longleftrightarrow$   
 $sublist\ xs\ ys \vee sublist\ xs\ zs \vee (\exists\ xs1\ xs2.\ xs = xs1\ @ \ xs2 \wedge suffix\ xs1\ ys \wedge prefix\ xs2\ zs)$   
 ⟨proof⟩

**lemma** *map-mono-sublist*:  
**assumes**  $sublist\ xs\ ys$   
**shows**  $sublist\ (map\ f\ xs)\ (map\ f\ ys)$   
 ⟨proof⟩

**lemma** *sublist-length-le*:  $sublist\ xs\ ys \implies length\ xs \leq length\ ys$   
 ⟨proof⟩

**lemma** *set-mono-sublist*:  $sublist\ xs\ ys \implies set\ xs \subseteq set\ ys$   
 ⟨proof⟩

**lemma** *prefix-imp-sublist* [simp, intro]:  $prefix\ xs\ ys \implies sublist\ xs\ ys$   
 ⟨proof⟩

**lemma** *suffix-imp-sublist* [simp, intro]:  $suffix\ xs\ ys \implies sublist\ xs\ ys$   
 ⟨proof⟩

**lemma** *sublist-take* [simp, intro]:  $sublist\ (take\ n\ xs)\ xs$   
 ⟨proof⟩

**lemma** *sublist-takeWhile* [simp, intro]:  $sublist\ (takeWhile\ P\ xs)\ xs$   
 ⟨proof⟩

**lemma** *sublist-drop* [simp, intro]:  $sublist\ (drop\ n\ xs)\ xs$   
 ⟨proof⟩

**lemma** *sublist-dropWhile* [simp, intro]:  $sublist\ (dropWhile\ P\ xs)\ xs$

$\langle \text{proof} \rangle$

**lemma** *sublist-tl* [*simp*, *intro*]: *sublist (tl xs) xs*  
 $\langle \text{proof} \rangle$

**lemma** *sublist-butlast* [*simp*, *intro*]: *sublist (butlast xs) xs*  
 $\langle \text{proof} \rangle$

**lemma** *sublist-rev* [*simp*]: *sublist (rev xs) (rev ys) = sublist xs ys*  
 $\langle \text{proof} \rangle$

**lemma** *sublist-rev-left*: *sublist (rev xs) ys = sublist xs (rev ys)*  
 $\langle \text{proof} \rangle$

**lemma** *sublist-rev-right*: *sublist xs (rev ys) = sublist (rev xs) ys*  
 $\langle \text{proof} \rangle$

**lemma** *snoc-sublist-snoc*:  
*sublist (xs @ [x]) (ys @ [y])  $\longleftrightarrow$*   
*(x = y  $\wedge$  suffix xs ys  $\vee$  sublist (xs @ [x]) ys)*  
 $\langle \text{proof} \rangle$

**lemma** *sublist-snoc*:  
*sublist xs (ys @ [y])  $\longleftrightarrow$  suffix xs (ys @ [y])  $\vee$  sublist xs ys*  
 $\langle \text{proof} \rangle$

**lemma** *sublist-imp-subseq* [*intro*]: *sublist xs ys  $\implies$  subseq xs ys*  
 $\langle \text{proof} \rangle$

**lemma** *sublist-map-rightE*:  
**assumes** *sublist xs (map f ys)*  
**shows**  $\exists xs'. \text{sublist } xs' \text{ ys} \wedge xs = \text{map } f \text{ } xs'$   
 $\langle \text{proof} \rangle$

**lemma** *sublist-remdups-adj*:  
**assumes** *sublist xs ys*  
**shows** *sublist (remdups-adj xs) (remdups-adj ys)*  
 $\langle \text{proof} \rangle$

### 57.12.2 *sublists*

**primrec** *sublists* :: 'a list  $\Rightarrow$  'a list list **where**  
*sublists [] = [[]]*  
 $| \text{sublists } (x \# xs) = \text{sublists } xs @ \text{map } ((\#) \ x) \ (\text{prefixes } xs)$

**lemma** *in-set-sublists* [*simp*]: *xs  $\in$  set (sublists ys)  $\longleftrightarrow$  sublist xs ys*  
 $\langle \text{proof} \rangle$

**lemma** *set-sublists-eq*: *set (sublists xs) = {ys. sublist ys xs}*

*<proof>*

**lemma** *length-sublists [simp]*:  $\text{length} (\text{sublists } xs) = \text{Suc} (\text{length } xs * \text{Suc} (\text{length } xs) \text{ div } 2)$

*<proof>*

### 57.13 Parametricity

**context includes** *lifting-syntax*

**begin**

**private lemma** *prefix-primrec*:

*prefix* = *rec-list* ( $\lambda xs. \text{True}$ ) ( $\lambda x xs xsa ys. \text{case } ys \text{ of } [] \Rightarrow \text{False} \mid y \# ys \Rightarrow x = y \wedge xsa \ ys$ )

*<proof>* **lemma** *sublist-primrec*:

*sublist* = ( $\lambda xs \ ys. \text{rec-list} (\lambda xs. xs = [])$ ) ( $\lambda y \ ys \ ysa \ xs. \text{prefix } xs \ (y \# ys) \vee ysa \ xs$ ) *ys xs*)

*<proof>* **lemma** *list-emb-primrec*:

*list-emb* = ( $\lambda uu \ uua \ uuaa. \text{rec-list} (\lambda P \ xs. \text{List.null } xs)$ ) ( $\lambda y \ ys \ ysa \ P \ xs. \text{case } xs \text{ of } [] \Rightarrow \text{True} \mid x \# xs \Rightarrow \text{if } P \ x \ y \text{ then } ysa \ P \ xs \text{ else } ysa \ P \ (x \# xs)$ ) *uuaa uu uua*)

*<proof>*

**lemma** *prefix-transfer [transfer-rule]*:

**assumes** [*transfer-rule*]: *bi-unique A*

**shows** (*list-all2 A*  $\implies$  *list-all2 A*  $\implies$  (=)) *prefix prefix*

*<proof>*

**lemma** *suffix-transfer [transfer-rule]*:

**assumes** [*transfer-rule*]: *bi-unique A*

**shows** (*list-all2 A*  $\implies$  *list-all2 A*  $\implies$  (=)) *suffix suffix*

*<proof>*

**lemma** *sublist-transfer [transfer-rule]*:

**assumes** [*transfer-rule*]: *bi-unique A*

**shows** (*list-all2 A*  $\implies$  *list-all2 A*  $\implies$  (=)) *sublist sublist*

*<proof>*

**lemma** *parallel-transfer [transfer-rule]*:

**assumes** [*transfer-rule*]: *bi-unique A*

**shows** (*list-all2 A*  $\implies$  *list-all2 A*  $\implies$  (=)) *parallel parallel*

*<proof>*

**lemma** *list-emb-transfer [transfer-rule]*:

((*A*  $\implies$  *A*  $\implies$  (=))  $\implies$  *list-all2 A*  $\implies$  *list-all2 A*  $\implies$  (=))

*list-emb list-emb*

*<proof>*

```

lemma strict-prefix-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 A ===> (=)) strict-prefix strict-prefix
  ⟨proof⟩

lemma strict-suffix-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 A ===> (=)) strict-suffix strict-suffix
  ⟨proof⟩

lemma strict-subseq-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 A ===> (=)) strict-subseq strict-subseq
  ⟨proof⟩

lemma strict-sublist-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 A ===> (=)) strict-sublist strict-sublist
  ⟨proof⟩

lemma prefixes-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 (list-all2 A)) prefixes prefixes
  ⟨proof⟩

lemma suffixes-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 (list-all2 A)) suffixes suffixes
  ⟨proof⟩

lemma sublists-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A ===> list-all2 (list-all2 A)) sublists sublists
  ⟨proof⟩

```

**end**

**end**

## 58 Linear Temporal Logic on Streams

```

theory Linear-Temporal-Logic-on-Streams
  imports Stream Sublist Extended-Nat Infinite-Set
begin

```

## 59 Preliminaries

**lemma** *shift-prefix*:

**assumes**  $xl @- xs = yl @- ys$  **and**  $length\ xl \leq length\ yl$

**shows**  $prefix\ xl\ yl$

*<proof>*

**lemma** *shift-prefix-cases*:

**assumes**  $xl @- xs = yl @- ys$

**shows**  $prefix\ xl\ yl \vee prefix\ yl\ xl$

*<proof>*

## 60 Linear temporal logic

Propositional connectives:

**abbreviation** (*input*) *IMPL* (**infix**  $\langle impl \rangle$  60)

**where**  $\varphi\ impl\ \psi \equiv \lambda\ xs.\ \varphi\ xs \longrightarrow \psi\ xs$

**abbreviation** (*input*) *OR* (**infix**  $\langle or \rangle$  60)

**where**  $\varphi\ or\ \psi \equiv \lambda\ xs.\ \varphi\ xs \vee \psi\ xs$

**abbreviation** (*input*) *AND* (**infix**  $\langle aand \rangle$  60)

**where**  $\varphi\ aand\ \psi \equiv \lambda\ xs.\ \varphi\ xs \wedge \psi\ xs$

**abbreviation** (*input*) *not* **where**  $not\ \varphi \equiv \lambda\ xs.\ \neg\ \varphi\ xs$

**abbreviation** (*input*) *true*  $\equiv \lambda\ xs.\ True$

**abbreviation** (*input*) *false*  $\equiv \lambda\ xs.\ False$

**lemma** *impl-not-or*:  $\varphi\ impl\ \psi = (not\ \varphi)\ or\ \psi$

*<proof>*

**lemma** *not-or*:  $not\ (\varphi\ or\ \psi) = (not\ \varphi)\ aand\ (not\ \psi)$

*<proof>*

**lemma** *not-aand*:  $not\ (\varphi\ aand\ \psi) = (not\ \varphi)\ or\ (not\ \psi)$

*<proof>*

**lemma** *non-not[simp]*:  $not\ (not\ \varphi) = \varphi$  *<proof>*

Temporal (LTL) connectives:

**fun** *holds* **where**  $holds\ P\ xs \longleftrightarrow P\ (shd\ xs)$

**fun** *next* **where**  $next\ \varphi\ xs = \varphi\ (stl\ xs)$

**definition** *HLD*  $s = holds\ (\lambda x.\ x \in s)$

**abbreviation** *HLD-next* (**infixr**  $\langle \cdot \rangle$  65) **where**

$s \cdot P \equiv \text{HLD } s \text{ aand } \text{next } P$

**context**

**notes**  $[[\text{inductive-internals}]]$

**begin**

**inductive**  $ev$  **for**  $\varphi$  **where**

$base: \varphi \text{ } xs \implies ev \varphi \text{ } xs$

|

$step: ev \varphi \text{ } (stl \text{ } xs) \implies ev \varphi \text{ } xs$

**coinductive**  $alw$  **for**  $\varphi$  **where**

$alw: [[\varphi \text{ } xs; alw \varphi \text{ } (stl \text{ } xs)]] \implies alw \varphi \text{ } xs$

— weak until:

**coinductive**  $UNTIL$  (**infix**  $\langle \text{until} \rangle$  60) **for**  $\varphi \psi$  **where**

$base: \psi \text{ } xs \implies (\varphi \text{ until } \psi) \text{ } xs$

|

$step: [[\varphi \text{ } xs; (\varphi \text{ until } \psi) \text{ } (stl \text{ } xs)]] \implies (\varphi \text{ until } \psi) \text{ } xs$

**end**

**lemma**  $holds\text{-mono}$ :

**assumes**  $holds: holds \text{ } P \text{ } xs$  **and**  $0: \bigwedge x. P \text{ } x \implies Q \text{ } x$

**shows**  $holds \text{ } Q \text{ } xs$

$\langle \text{proof} \rangle$

**lemma**  $holds\text{-aand}$ :

$(holds \text{ } P \text{ } \text{aand} \text{ } holds \text{ } Q) \text{ } steps \longleftrightarrow holds \text{ } (\lambda \text{ step. } P \text{ } step \wedge Q \text{ } step) \text{ } steps \langle \text{proof} \rangle$

**lemma**  $HLD\text{-iff}$ :  $HLD \text{ } s \ \omega \longleftrightarrow shd \ \omega \in s$

$\langle \text{proof} \rangle$

**lemma**  $HLD\text{-Stream}[simp]$ :  $HLD \text{ } X \text{ } (x \ \#\# \ \omega) \longleftrightarrow x \in X$

$\langle \text{proof} \rangle$

**lemma**  $next\text{-mono}$ :

**assumes**  $next: next \varphi \text{ } xs$  **and**  $0: \bigwedge xs. \varphi \text{ } xs \implies \psi \text{ } xs$

**shows**  $next \ \psi \text{ } xs$

$\langle \text{proof} \rangle$

**declare**  $ev.intros[\text{intro}]$

**declare**  $alw.cases[\text{elim}]$

**lemma**  $ev\text{-induct-strong}[\text{consumes } 1, \text{ case-names base step}]$ :

$ev \varphi \text{ } x \implies (\bigwedge xs. \varphi \text{ } xs \implies P \text{ } xs) \implies (\bigwedge xs. ev \varphi \text{ } (stl \text{ } xs) \implies \neg \varphi \text{ } xs \implies P \text{ } (stl \text{ } xs)) \implies P \text{ } x$

$\langle \text{proof} \rangle$

**lemma** *alw-coinduct*[*consumes 1, case-names alw stl*]:

$X x \Longrightarrow (\bigwedge x. X x \Longrightarrow \varphi x) \Longrightarrow (\bigwedge x. X x \Longrightarrow \neg \text{alw } \varphi (\text{stl } x) \Longrightarrow X (\text{stl } x))$   
 $\Longrightarrow \text{alw } \varphi x$   
 ⟨*proof*⟩

**lemma** *ev-mono*:

**assumes** *ev*:  $\text{ev } \varphi \text{ } xs$  **and** *0*:  $\bigwedge xs. \varphi \text{ } xs \Longrightarrow \psi \text{ } xs$   
**shows**  $\text{ev } \psi \text{ } xs$   
 ⟨*proof*⟩

**lemma** *alw-mono*:

**assumes** *alw*:  $\text{alw } \varphi \text{ } xs$  **and** *0*:  $\bigwedge xs. \varphi \text{ } xs \Longrightarrow \psi \text{ } xs$   
**shows**  $\text{alw } \psi \text{ } xs$   
 ⟨*proof*⟩

**lemma** *until-monoL*:

**assumes** *until*:  $(\varphi 1 \text{ until } \psi) \text{ } xs$  **and** *0*:  $\bigwedge xs. \varphi 1 \text{ } xs \Longrightarrow \varphi 2 \text{ } xs$   
**shows**  $(\varphi 2 \text{ until } \psi) \text{ } xs$   
 ⟨*proof*⟩

**lemma** *until-monoR*:

**assumes** *until*:  $(\varphi \text{ until } \psi 1) \text{ } xs$  **and** *0*:  $\bigwedge xs. \psi 1 \text{ } xs \Longrightarrow \psi 2 \text{ } xs$   
**shows**  $(\varphi \text{ until } \psi 2) \text{ } xs$   
 ⟨*proof*⟩

**lemma** *until-mono*:

**assumes** *until*:  $(\varphi 1 \text{ until } \psi 1) \text{ } xs$  **and**  
*0*:  $\bigwedge xs. \varphi 1 \text{ } xs \Longrightarrow \varphi 2 \text{ } xs \bigwedge xs. \psi 1 \text{ } xs \Longrightarrow \psi 2 \text{ } xs$   
**shows**  $(\varphi 2 \text{ until } \psi 2) \text{ } xs$   
 ⟨*proof*⟩

**lemma** *until-false*:  $\varphi \text{ until false} = \text{alw } \varphi$

⟨*proof*⟩

**lemma** *ev-nxt*:  $\text{ev } \varphi = (\varphi \text{ or } \text{nxt } (\text{ev } \varphi))$

⟨*proof*⟩

**lemma** *alw-nxt*:  $\text{alw } \varphi = (\varphi \text{ aand } \text{nxt } (\text{alw } \varphi))$

⟨*proof*⟩

**lemma** *ev-ev*[*simp*]:  $\text{ev } (\text{ev } \varphi) = \text{ev } \varphi$

⟨*proof*⟩

**lemma** *alw-alw*[*simp*]:  $\text{alw } (\text{alw } \varphi) = \text{alw } \varphi$

⟨*proof*⟩

**lemma** *ev-shift*:

**assumes**  $\text{ev } \varphi \text{ } xs$

**shows**  $\text{ev } \varphi \text{ } (xl @- \text{ } xs)$



$\langle proof \rangle$

**lemma** *ev-imp-shift*:

**assumes**  $ev \ \varphi \ xs$  **shows**  $\exists \ xl \ xs2. \ xs = xl \ @- \ xs2 \wedge \varphi \ xs2$

$\langle proof \rangle$

**lemma** *alw-ev-shift*:  $alw \ \varphi \ xs1 \implies ev \ (alw \ \varphi) \ (xl \ @- \ xs1)$

$\langle proof \rangle$

**lemma** *alw-shift*:

**assumes**  $alw \ \varphi \ (xl \ @- \ xs)$

**shows**  $alw \ \varphi \ xs$

$\langle proof \rangle$

**lemma** *ev-ex-nxt*:

**assumes**  $ev \ \varphi \ xs$

**shows**  $\exists \ n. \ (nxt \ \overset{\sim}{\sim} \ n) \ \varphi \ xs$

$\langle proof \rangle$

**lemma** *alw-sdrop*:

**assumes**  $alw \ \varphi \ xs$  **shows**  $alw \ \varphi \ (sdrop \ n \ xs)$

$\langle proof \rangle$

**lemma** *nxt-sdrop*:  $(nxt \ \overset{\sim}{\sim} \ n) \ \varphi \ xs \longleftrightarrow \varphi \ (sdrop \ n \ xs)$

$\langle proof \rangle$

**definition**  $wait \ \varphi \ xs \equiv LEAST \ n. \ (nxt \ \overset{\sim}{\sim} \ n) \ \varphi \ xs$

**lemma** *nxt-wait*:

**assumes**  $ev \ \varphi \ xs$  **shows**  $(nxt \ \overset{\sim}{\sim} \ (wait \ \varphi \ xs)) \ \varphi \ xs$

$\langle proof \rangle$

**lemma** *nxt-wait-least*:

**assumes**  $ev: ev \ \varphi \ xs$  **and**  $nxt: (nxt \ \overset{\sim}{\sim} \ n) \ \varphi \ xs$  **shows**  $wait \ \varphi \ xs \leq n$

$\langle proof \rangle$

**lemma** *sdrop-wait*:

**assumes**  $ev \ \varphi \ xs$  **shows**  $\varphi \ (sdrop \ (wait \ \varphi \ xs) \ xs)$

$\langle proof \rangle$

**lemma** *sdrop-wait-least*:

**assumes**  $ev: ev \ \varphi \ xs$  **and**  $nxt: \varphi \ (sdrop \ n \ xs)$  **shows**  $wait \ \varphi \ xs \leq n$

$\langle proof \rangle$

**lemma** *nxt-ev*:  $(nxt \ \overset{\sim}{\sim} \ n) \ \varphi \ xs \implies ev \ \varphi \ xs$

$\langle proof \rangle$

**lemma** *not-ev*:  $not \ (ev \ \varphi) = alw \ (not \ \varphi)$

$\langle proof \rangle$

**lemma** *not-alw*:  $\text{not } (\text{alw } \varphi) = \text{ev } (\text{not } \varphi)$   
 ⟨*proof*⟩

**lemma** *not-ev-not[simp]*:  $\text{not } (\text{ev } (\text{not } \varphi)) = \text{alw } \varphi$   
 ⟨*proof*⟩

**lemma** *not-alw-not[simp]*:  $\text{not } (\text{alw } (\text{not } \varphi)) = \text{ev } \varphi$   
 ⟨*proof*⟩

**lemma** *alw-ev-sdrop*:  
**assumes**  $\text{alw } (\text{ev } \varphi) \text{ (sdrop } m \text{ } xs)$   
**shows**  $\text{alw } (\text{ev } \varphi) \text{ } xs$   
 ⟨*proof*⟩

**lemma** *ev-alw-imp-alw-ev*:  
**assumes**  $\text{ev } (\text{alw } \varphi) \text{ } xs$  **shows**  $\text{alw } (\text{ev } \varphi) \text{ } xs$   
 ⟨*proof*⟩

**lemma** *alw-aand*:  $\text{alw } (\varphi \text{ aand } \psi) = \text{alw } \varphi \text{ aand } \text{alw } \psi$   
 ⟨*proof*⟩

**lemma** *ev-or*:  $\text{ev } (\varphi \text{ or } \psi) = \text{ev } \varphi \text{ or } \text{ev } \psi$   
 ⟨*proof*⟩

**lemma** *ev-alw-aand*:  
**assumes**  $\varphi: \text{ev } (\text{alw } \varphi) \text{ } xs$  **and**  $\psi: \text{ev } (\text{alw } \psi) \text{ } xs$   
**shows**  $\text{ev } (\text{alw } (\varphi \text{ aand } \psi)) \text{ } xs$   
 ⟨*proof*⟩

**lemma** *ev-alw-alw-impl*:  
**assumes**  $\text{ev } (\text{alw } \varphi) \text{ } xs$  **and**  $\text{alw } (\text{alw } \varphi \text{ impl } \text{ev } \psi) \text{ } xs$   
**shows**  $\text{ev } \psi \text{ } xs$   
 ⟨*proof*⟩

**lemma** *ev-alw-stl[simp]*:  $\text{ev } (\text{alw } \varphi) \text{ (stl } x) \longleftrightarrow \text{ev } (\text{alw } \varphi) \text{ } x$   
 ⟨*proof*⟩

**lemma** *alw-alw-impl-ev*:  
 $\text{alw } (\text{alw } \varphi \text{ impl } \text{ev } \psi) = (\text{ev } (\text{alw } \varphi) \text{ impl } \text{alw } (\text{ev } \psi)) \text{ (is ?A = ?B)}$   
 ⟨*proof*⟩

**lemma** *ev-alw-impl*:  
**assumes**  $\text{ev } \varphi \text{ } xs$  **and**  $\text{alw } (\varphi \text{ impl } \psi) \text{ } xs$  **shows**  $\text{ev } \psi \text{ } xs$   
 ⟨*proof*⟩

**lemma** *ev-alw-impl-ev*:  
**assumes**  $\text{ev } \varphi \text{ } xs$  **and**  $\text{alw } (\varphi \text{ impl } \text{ev } \psi) \text{ } xs$  **shows**  $\text{ev } \psi \text{ } xs$   
 ⟨*proof*⟩

**lemma** *alw-mp*:

**assumes** *alw*  $\varphi$  *xs* **and** *alw* ( $\varphi$  *impl*  $\psi$ ) *xs*

**shows** *alw*  $\psi$  *xs*

*<proof>*

**lemma** *all-imp-alw*:

**assumes**  $\bigwedge$  *xs*.  $\varphi$  *xs* **shows** *alw*  $\varphi$  *xs*

*<proof>*

**lemma** *alw-impl-ev-alw*:

**assumes** *alw* ( $\varphi$  *impl* *ev*  $\psi$ ) *xs*

**shows** *alw* (*ev*  $\varphi$  *impl* *ev*  $\psi$ ) *xs*

*<proof>*

**lemma** *ev-holds-sset*:

*ev* (*holds* *P*) *xs*  $\longleftrightarrow$  ( $\exists$  *x*  $\in$  *sset xs*. *P x*) (**is** ?*L*  $\longleftrightarrow$  ?*R*)

*<proof>*

LTL as a program logic:

**lemma** *alw-invar*:

**assumes**  $\varphi$  *xs* **and** *alw* ( $\varphi$  *impl* *next*  $\varphi$ ) *xs*

**shows** *alw*  $\varphi$  *xs*

*<proof>*

**lemma** *variance*:

**assumes** 1:  $\varphi$  *xs* **and** 2: *alw* ( $\varphi$  *impl* ( $\psi$  *or* *next*  $\varphi$ )) *xs*

**shows** (*alw*  $\varphi$  *or* *ev*  $\psi$ ) *xs*

*<proof>*

**lemma** *ev-alw-imp-next*:

**assumes** *e*: *ev*  $\varphi$  *xs* **and** *a*: *alw* ( $\varphi$  *impl* (*next*  $\varphi$ )) *xs*

**shows** *ev* (*alw*  $\varphi$ ) *xs*

*<proof>*

**inductive** *ev-at* :: ('*a* *stream*  $\Rightarrow$  *bool*)  $\Rightarrow$  *nat*  $\Rightarrow$  '*a* *stream*  $\Rightarrow$  *bool* **for** *P* :: '*a* *stream*  $\Rightarrow$  *bool* **where**

*base*: *P*  $\omega \Longrightarrow$  *ev-at* *P* 0  $\omega$

| *step*:  $\neg$  *P*  $\omega \Longrightarrow$  *ev-at* *P* *n* (*stl*  $\omega$ )  $\Longrightarrow$  *ev-at* *P* (*Suc* *n*)  $\omega$

**inductive-simps** *ev-at-0*[*simp*]: *ev-at* *P* 0  $\omega$

**inductive-simps** *ev-at-Suc*[*simp*]: *ev-at* *P* (*Suc* *n*)  $\omega$

**lemma** *ev-at-imp-snth*: *ev-at* *P* *n*  $\omega \Longrightarrow$  *P* (*sdrop* *n*  $\omega$ )

*<proof>*

**lemma** *ev-at-HLD-imp-snth*: *ev-at* (*HLD* *X*) *n*  $\omega \Longrightarrow$   $\omega !!$  *n*  $\in$  *X*

*<proof>*

**lemma** *ev-at-HLD-single-imp-snth*:  $ev\text{-at } (HLD \{x\}) n \omega \implies \omega !! n = x$   
 ⟨proof⟩

**lemma** *ev-at-unique*:  $ev\text{-at } P n \omega \implies ev\text{-at } P m \omega \implies n = m$   
 ⟨proof⟩

**lemma** *ev-iff-ev-at*:  $ev P \omega \longleftrightarrow (\exists n. ev\text{-at } P n \omega)$   
 ⟨proof⟩

**lemma** *ev-at-shift*:  $ev\text{-at } (HLD X) i (stake (Suc i) \omega @- \omega' :: 's\ stream) \longleftrightarrow ev\text{-at } (HLD X) i \omega$   
 ⟨proof⟩

**lemma** *ev-iff-ev-at-unique*:  $ev P \omega \longleftrightarrow (\exists! n. ev\text{-at } P n \omega)$   
 ⟨proof⟩

**lemma** *alw-HLD-iff-streams*:  $alw (HLD X) \omega \longleftrightarrow \omega \in streams X$   
 ⟨proof⟩

**lemma** *not-HLD*:  $not (HLD X) = HLD (- X)$   
 ⟨proof⟩

**lemma** *not-alw-iff*:  $\neg (alw P \omega) \longleftrightarrow ev (not P) \omega$   
 ⟨proof⟩

**lemma** *not-ev-iff*:  $\neg (ev P \omega) \longleftrightarrow alw (not P) \omega$   
 ⟨proof⟩

**lemma** *ev-Stream*:  $ev P (x \#\# s) \longleftrightarrow P (x \#\# s) \vee ev P s$   
 ⟨proof⟩

**lemma** *alw-ev-imp-ev-alw*:  
**assumes**  $alw (ev P) \omega$  **shows**  $ev (P \text{ aand } alw (ev P)) \omega$   
 ⟨proof⟩

**lemma** *ev-False*:  $ev (\lambda x. False) \omega \longleftrightarrow False$   
 ⟨proof⟩

**lemma** *alw-False*:  $alw (\lambda x. False) \omega \longleftrightarrow False$   
 ⟨proof⟩

**lemma** *ev-iff-sdrop*:  $ev P \omega \longleftrightarrow (\exists m. P (sdrop m \omega))$   
 ⟨proof⟩

**lemma** *alw-iff-sdrop*:  $alw P \omega \longleftrightarrow (\forall m. P (sdrop m \omega))$   
 ⟨proof⟩

**lemma** *infinite-iff-alw-ev*:  $infinite \{m. P (sdrop m \omega)\} \longleftrightarrow alw (ev P) \omega$

*<proof>*

**lemma** *alw-inv*:

**assumes** *stl*:  $\bigwedge s. f (stl\ s) = stl (f\ s)$

**shows**  $alw\ P (f\ s) \longleftrightarrow alw (\lambda x. P (f\ x))\ s$

*<proof>*

**lemma** *ev-inv*:

**assumes** *stl*:  $\bigwedge s. f (stl\ s) = stl (f\ s)$

**shows**  $ev\ P (f\ s) \longleftrightarrow ev (\lambda x. P (f\ x))\ s$

*<proof>*

**lemma** *alw-smap*:  $alw\ P (smap\ f\ s) \longleftrightarrow alw (\lambda x. P (smap\ f\ x))\ s$

*<proof>*

**lemma** *ev-smap*:  $ev\ P (smap\ f\ s) \longleftrightarrow ev (\lambda x. P (smap\ f\ x))\ s$

*<proof>*

**lemma** *alw-cong*:

**assumes** *P*:  $alw\ P\ \omega$  **and** *eq*:  $\bigwedge \omega. P\ \omega \implies Q1\ \omega \longleftrightarrow Q2\ \omega$

**shows**  $alw\ Q1\ \omega \longleftrightarrow alw\ Q2\ \omega$

*<proof>*

**lemma** *ev-cong*:

**assumes** *P*:  $alw\ P\ \omega$  **and** *eq*:  $\bigwedge \omega. P\ \omega \implies Q1\ \omega \longleftrightarrow Q2\ \omega$

**shows**  $ev\ Q1\ \omega \longleftrightarrow ev\ Q2\ \omega$

*<proof>*

**lemma** *alwD*:  $alw\ P\ x \implies P\ x$

*<proof>*

**lemma** *alw-alwD*:  $alw\ P\ \omega \implies alw (alw\ P)\ \omega$

*<proof>*

**lemma** *alw-ev-stl*:  $alw (ev\ P) (stl\ \omega) \longleftrightarrow alw (ev\ P)\ \omega$

*<proof>*

**lemma** *holds-Stream*:  $holds\ P (x\ \#\#\ s) \longleftrightarrow P\ x$

*<proof>*

**lemma** *holds-eq1[simp]*:  $holds ((=)\ x) = HLD\ \{x\}$

*<proof>*

**lemma** *holds-eq2[simp]*:  $holds (\lambda y. y = x) = HLD\ \{x\}$

*<proof>*

**lemma** *not-holds-eq[simp]*:  $holds (- (=)\ x) = not (HLD\ \{x\})$

*<proof>*

Strong until

**context**

**notes**  $[[\text{inductive-internals}]]$

**begin**

**inductive** *suntil* (**infix**  $\langle \text{suntil} \rangle$  60) **for**  $\varphi \psi$  **where**

*base:*  $\psi \omega \implies (\varphi \text{ suntil } \psi) \omega$

| *step:*  $\varphi \omega \implies (\varphi \text{ suntil } \psi) (\text{stl } \omega) \implies (\varphi \text{ suntil } \psi) \omega$

**inductive-simps** *suntil-Stream:*  $(\varphi \text{ suntil } \psi) (x \#\# s)$

**end**

**lemma** *suntil-induct-strong* $[\text{consumes } 1, \text{case-names base step}]$ :

$(\varphi \text{ suntil } \psi) x \implies$

$(\bigwedge \omega. \psi \omega \implies P \omega) \implies$

$(\bigwedge \omega. \varphi \omega \implies \neg \psi \omega \implies (\varphi \text{ suntil } \psi) (\text{stl } \omega) \implies P (\text{stl } \omega) \implies P \omega) \implies P x$

$\langle \text{proof} \rangle$

**lemma** *ev-suntil:*  $(\varphi \text{ suntil } \psi) \omega \implies \text{ev } \psi \omega$

$\langle \text{proof} \rangle$

**lemma** *suntil-inv:*

**assumes** *stl:*  $\bigwedge s. f (\text{stl } s) = \text{stl } (f s)$

**shows**  $(P \text{ suntil } Q) (f s) \longleftrightarrow ((\lambda x. P (f x)) \text{ suntil } (\lambda x. Q (f x))) s$

$\langle \text{proof} \rangle$

**lemma** *suntil-smap:*  $(P \text{ suntil } Q) (\text{smap } f s) \longleftrightarrow ((\lambda x. P (\text{smap } f x)) \text{ suntil } (\lambda x.$

$Q (\text{smap } f x))) s$

$\langle \text{proof} \rangle$

**lemma** *hld-smap:*  $\text{HLD } x (\text{smap } f s) = \text{holds } (\lambda y. f y \in x) s$

$\langle \text{proof} \rangle$

**lemma** *suntil-mono:*

**assumes** *eq:*  $\bigwedge \omega. P \omega \implies Q1 \omega \implies Q2 \omega \bigwedge \omega. P \omega \implies R1 \omega \implies R2 \omega$

**assumes** *\**:  $(Q1 \text{ suntil } R1) \omega \text{ alw } P \omega$  **shows**  $(Q2 \text{ suntil } R2) \omega$

$\langle \text{proof} \rangle$

**lemma** *suntil-cong:*

$\text{alw } P \omega \implies (\bigwedge \omega. P \omega \implies Q1 \omega \longleftrightarrow Q2 \omega) \implies (\bigwedge \omega. P \omega \implies R1 \omega \longleftrightarrow R2 \omega) \implies$

$(Q1 \text{ suntil } R1) \omega \longleftrightarrow (Q2 \text{ suntil } R2) \omega$

$\langle \text{proof} \rangle$

**lemma** *ev-suntil-iff:*  $\text{ev } (P \text{ suntil } Q) \omega \longleftrightarrow \text{ev } Q \omega$

$\langle \text{proof} \rangle$

**lemma** *true-suntil:*  $((\lambda -. \text{True}) \text{ suntil } P) = \text{ev } P$

$\langle \text{proof} \rangle$

**lemma** *suntil-lfp*:  $(\varphi \text{ until } \psi) = \text{lfp } (\lambda P s. \psi s \vee (\varphi s \wedge P (\text{stl } s)))$   
 ⟨proof⟩

**lemma** *sfilter-P[simp]*:  $P (\text{shd } s) \implies \text{sfilter } P s = \text{shd } s \#\#\text{sfilter } P (\text{stl } s)$   
 ⟨proof⟩

**lemma** *sfilter-not-P[simp]*:  $\neg P (\text{shd } s) \implies \text{sfilter } P s = \text{sfilter } P (\text{stl } s)$   
 ⟨proof⟩

**lemma** *sfilter-eq*:  
**assumes** *ev* (*holds* *P*) *s*  
**shows**  $\text{sfilter } P s = x \#\#\text{sfilter } P s' \longleftrightarrow$   
 $P x \wedge (\text{not } (\text{holds } P) \text{ until } (\text{HLD } \{x\} \text{ aand next } (\lambda s. \text{sfilter } P s = s'))) s$   
 ⟨proof⟩

**lemma** *sfilter-streams*:  
 $\text{alw } (\text{ev } (\text{holds } P)) \omega \implies \omega \in \text{streams } A \implies \text{sfilter } P \omega \in \text{streams } \{x \in A. P x\}$   
 ⟨proof⟩

**lemma** *alw-sfilter*:  
**assumes** \*:  $\text{alw } (\text{ev } (\text{holds } P)) s$   
**shows**  $\text{alw } Q (\text{sfilter } P s) \longleftrightarrow \text{alw } (\lambda x. Q (\text{sfilter } P x)) s$   
 ⟨proof⟩

**lemma** *ev-sfilter*:  
**assumes** \*:  $\text{alw } (\text{ev } (\text{holds } P)) s$   
**shows**  $\text{ev } Q (\text{sfilter } P s) \longleftrightarrow \text{ev } (\lambda x. Q (\text{sfilter } P x)) s$   
 ⟨proof⟩

**lemma** *holds-sfilter*:  
**assumes** *ev* (*holds* *Q*) *s* **shows**  $\text{holds } P (\text{sfilter } Q s) \longleftrightarrow (\text{not } (\text{holds } Q) \text{ until } (\text{holds } (Q \text{ aand } P))) s$   
 ⟨proof⟩

**lemma** *suntil-aand-next*:  
 $(\varphi \text{ until } (\varphi \text{ aand next } \psi)) \omega \longleftrightarrow (\varphi \text{ aand next } (\varphi \text{ until } \psi)) \omega$   
 ⟨proof⟩

**lemma** *alw-sconst*:  $\text{alw } P (\text{sconst } x) \longleftrightarrow P (\text{sconst } x)$   
 ⟨proof⟩

**lemma** *ev-sconst*:  $\text{ev } P (\text{sconst } x) \longleftrightarrow P (\text{sconst } x)$   
 ⟨proof⟩

**lemma** *suntil-sconst*:  $(\varphi \text{ until } \psi) (\text{sconst } x) \longleftrightarrow \psi (\text{sconst } x)$   
 ⟨proof⟩

**lemma** *hld-smap'*:  $\text{HLD } x (\text{smap } f s) = \text{HLD } (f \text{ -' } x) s$

*<proof>*

**lemma** *pigeonhole-stream*:

**assumes**  $alw (HLD s) \omega$

**assumes** *finite s*

**shows**  $\exists x \in s. alw (ev (HLD \{x\})) \omega$

*<proof>*

**lemma** *ev-eq-suntil*:  $ev P \omega \longleftrightarrow (not P \text{ until } P) \omega$

*<proof>*

## 61 Weak vs. strong until (contributed by Michael Foster, University of Sheffield)

**lemma** *suntil-implies-until*:  $(\varphi \text{ until } \psi) \omega \implies (\varphi \text{ until } \psi) \omega$

*<proof>*

**lemma** *alw-implies-until*:  $alw \varphi \omega \implies (\varphi \text{ until } \psi) \omega$

*<proof>*

**lemma** *until-ev-suntil*:  $(\varphi \text{ until } \psi) \omega \implies ev \psi \omega \implies (\varphi \text{ until } \psi) \omega$

*<proof>*

**lemma** *suntil-as-until*:  $(\varphi \text{ until } \psi) \omega = ((\varphi \text{ until } \psi) \omega \wedge ev \psi \omega)$

*<proof>*

**lemma** *until-not-released-now*:  $(\varphi \text{ until } \psi) \omega \implies \neg \psi \omega \implies \varphi \omega$

*<proof>*

**lemma** *until-must-release-ev*:  $(\varphi \text{ until } \psi) \omega \implies ev (not \varphi) \omega \implies ev \psi \omega$

*<proof>*

**lemma** *until-as-suntil*:  $(\varphi \text{ until } \psi) \omega = ((\varphi \text{ until } \psi) \text{ or } (alw \varphi)) \omega$

*<proof>*

**lemma** *alw-holds*:  $alw (holds P) (h\#\#t) = (P h \wedge alw (holds P) t)$

*<proof>*

**lemma** *alw-holds2*:  $alw (holds P) ss = (P (shd ss) \wedge alw (holds P) (stl ss))$

*<proof>*

**lemma** *alw-eq-sconst*:  $(alw (HLD \{h\}) t) = (t = sconst h)$

*<proof>*

**lemma** *sdrop-if-suntil*:  $(p \text{ until } q) \omega \implies \exists j. q (sdrop j \omega) \wedge (\forall k < j. p (sdrop k \omega))$

*<proof>*



**lemma not-suntil:**  $(\neg (p \text{ suntil } q) \omega) = (\neg (p \text{ until } q) \omega \vee \text{alw } (\text{not } q) \omega)$   
 ⟨proof⟩

**lemma sdrop-until:**  $q (\text{sdrop } j \omega) \implies \forall k < j. p (\text{sdrop } k \omega) \implies (p \text{ until } q) \omega$   
 ⟨proof⟩

**lemma sdrop-suntil:**  $q (\text{sdrop } j \omega) \implies (\forall k < j. p (\text{sdrop } k \omega)) \implies (p \text{ suntil } q) \omega$   
 ⟨proof⟩

**lemma suntil-iff-sdrop:**  $(p \text{ suntil } q) \omega = (\exists j. q (\text{sdrop } j \omega) \wedge (\forall k < j. p (\text{sdrop } k \omega)))$   
 ⟨proof⟩

end

## 62 Lists as vectors

**theory ListVector**  
**imports Main**  
**begin**

A vector-space like structure of lists and arithmetic operations on them. Is only a vector space if restricted to lists of the same length.

Multiplication with a scalar:

**abbreviation scale** ::  $('a::\text{times}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  (**infix**  $\langle *_{\text{s}} \rangle$  70)  
**where**  $x *_{\text{s}} xs \equiv \text{map } ((*) x) xs$

**lemma scale1** [*simp*]:  $(1::'a::\text{monoid-mult}) *_{\text{s}} xs = xs$   
 ⟨proof⟩

### 62.1 + and -

**fun zipwith0** ::  $('a::\text{zero} \Rightarrow 'b::\text{zero} \Rightarrow 'c) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow 'c \text{ list}$   
**where**

$\text{zipwith0 } f \ [] \ [] = [] \ |$   
 $\text{zipwith0 } f (x\#xs) (y\#ys) = f x y \# \text{zipwith0 } f xs ys \ |$   
 $\text{zipwith0 } f (x\#xs) [] = f x 0 \# \text{zipwith0 } f xs [] \ |$   
 $\text{zipwith0 } f [] (y\#ys) = f 0 y \# \text{zipwith0 } f [] ys$

**instantiation list** ::  $(\{\text{zero}, \text{plus}\}) \text{ plus}$   
**begin**

**definition**  
*list-add-def*:  $(+) = \text{zipwith0 } (+)$

**instance** ⟨proof⟩

end

**instantiation** *list* :: (*{zero, uminus}*) *uminus*  
**begin**

**definition**

*list-uminus-def*: *uminus* = *map uminus*

**instance** *<proof>*

**end**

**instantiation** *list* :: (*{zero, minus}*) *minus*  
**begin**

**definition**

*list-diff-def*: *(-)* = *zipwith0 (-)*

**instance** *<proof>*

**end**

**lemma** *zipwith0-Nil[simp]*: *zipwith0 f [] ys* = *map (f 0) ys*  
*<proof>*

**lemma** *list-add-Nil[simp]*: *[] + xs* = *(xs::'a::monoid-add list)*  
*<proof>*

**lemma** *list-add-Nil2[simp]*: *xs + []* = *(xs::'a::monoid-add list)*  
*<proof>*

**lemma** *list-add-Cons[simp]*: *(x#xs) + (y#ys)* = *(x+y)#(xs+ys)*  
*<proof>*

**lemma** *list-diff-Nil[simp]*: *[] - xs* = *-(xs::'a::group-add list)*  
*<proof>*

**lemma** *list-diff-Nil2[simp]*: *xs - []* = *(xs::'a::group-add list)*  
*<proof>*

**lemma** *list-diff-Cons-Cons[simp]*: *(x#xs) - (y#ys)* = *(x-y)#(xs-ys)*  
*<proof>*

**lemma** *list-uminus-Cons[simp]*: *-(x#xs)* = *(-x)#(-xs)*  
*<proof>*

**lemma** *self-list-diff*:

*xs - xs* = *replicate (length(xs::'a::group-add list)) 0*  
*<proof>*

**lemma** *list-add-assoc*:  
**fixes**  $xs :: 'a::monoid-add\ list$   
**shows**  $(xs+ys)+zs = xs+(ys+zs)$   
 $\langle proof \rangle$

## 62.2 Inner product

**definition** *iprod* ::  $'a::ring\ list \Rightarrow 'a\ list \Rightarrow 'a$  ( $\langle \langle open\ block\ notation = \langle mixfix\ iprod \rangle \langle -, - \rangle \rangle \rangle$ )  
**where**  $\langle xs, ys \rangle = (\sum (x, y) \leftarrow zip\ xs\ ys.\ x*y)$

**lemma** *iprod-Nil[simp]*:  $\langle [], ys \rangle = 0$   
 $\langle proof \rangle$

**lemma** *iprod-Nil2[simp]*:  $\langle xs, [] \rangle = 0$   
 $\langle proof \rangle$

**lemma** *iprod-Cons[simp]*:  $\langle x\#\ xs, y\#\ ys \rangle = x*y + \langle xs, ys \rangle$   
 $\langle proof \rangle$

**lemma** *iprod0-if-coeffs0*:  $\forall c \in set\ cs.\ c = 0 \implies \langle cs, xs \rangle = 0$   
 $\langle proof \rangle$

**lemma** *iprod-uminus[simp]*:  $\langle -xs, ys \rangle = -\langle xs, ys \rangle$   
 $\langle proof \rangle$

**lemma** *iprod-left-add-distrib*:  $\langle xs + ys, zs \rangle = \langle xs, zs \rangle + \langle ys, zs \rangle$   
 $\langle proof \rangle$

**lemma** *iprod-left-diff-distrib*:  $\langle xs - ys, zs \rangle = \langle xs, zs \rangle - \langle ys, zs \rangle$   
 $\langle proof \rangle$

**lemma** *iprod-assoc*:  $\langle x *_s xs, ys \rangle = x * \langle xs, ys \rangle$   
 $\langle proof \rangle$

**end**

## 63 Definitions of Least Upper Bounds and Greatest Lower Bounds

**theory** *Lub-Glb*  
**imports** *Complex-Main*  
**begin**

Thanks to suggestions by James Margetson

**definition** *setle* ::  $'a\ set \Rightarrow 'a::ord \Rightarrow bool$  (**infixl**  $\langle * \leq \rangle$  70)  
**where**  $S * \leq x = (\forall y \in S.\ y \leq x)$

**definition**  $setge :: 'a::ord \Rightarrow 'a\ set \Rightarrow bool$  (**infixl**  $\langle \leq^* \rangle$  70)  
**where**  $x \leq^* S = (\forall y \in S. x \leq y)$

### 63.1 Rules for the Relations $\leq^*$ and $\leq$

**lemma**  $setleI: \forall y \in S. y \leq x \Longrightarrow S \leq^* x$   
 $\langle proof \rangle$

**lemma**  $setleD: S \leq^* x \Longrightarrow y \in S \Longrightarrow y \leq x$   
 $\langle proof \rangle$

**lemma**  $setgeI: \forall y \in S. x \leq y \Longrightarrow x \leq^* S$   
 $\langle proof \rangle$

**lemma**  $setgeD: x \leq^* S \Longrightarrow y \in S \Longrightarrow x \leq y$   
 $\langle proof \rangle$

**definition**  $leastP :: ('a \Rightarrow bool) \Rightarrow 'a::ord \Rightarrow bool$   
**where**  $leastP\ P\ x = (P\ x \wedge x \leq^* Collect\ P)$

**definition**  $isUb :: 'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$   
**where**  $isUb\ R\ S\ x = (S \leq^* x \wedge x \in R)$

**definition**  $isLub :: 'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$   
**where**  $isLub\ R\ S\ x = leastP\ (isUb\ R\ S)\ x$

**definition**  $ubs :: 'a\ set \Rightarrow 'a::ord\ set \Rightarrow 'a\ set$   
**where**  $ubs\ R\ S = Collect\ (isUb\ R\ S)$

### 63.2 Rules about the Operators $leastP$ , $ub$ and $lub$

**lemma**  $leastPD1: leastP\ P\ x \Longrightarrow P\ x$   
 $\langle proof \rangle$

**lemma**  $leastPD2: leastP\ P\ x \Longrightarrow x \leq^* Collect\ P$   
 $\langle proof \rangle$

**lemma**  $leastPD3: leastP\ P\ x \Longrightarrow y \in Collect\ P \Longrightarrow x \leq y$   
 $\langle proof \rangle$

**lemma**  $isLubD1: isLub\ R\ S\ x \Longrightarrow S \leq^* x$   
 $\langle proof \rangle$

**lemma**  $isLubD1a: isLub\ R\ S\ x \Longrightarrow x \in R$   
 $\langle proof \rangle$

**lemma**  $isLub-isUb: isLub\ R\ S\ x \Longrightarrow isUb\ R\ S\ x$   
 $\langle proof \rangle$

**lemma** *isLubD2*:  $isLub\ R\ S\ x \implies y \in S \implies y \leq x$   
*<proof>*

**lemma** *isLubD3*:  $isLub\ R\ S\ x \implies leastP\ (isUb\ R\ S)\ x$   
*<proof>*

**lemma** *isLubI1*:  $leastP(isUb\ R\ S)\ x \implies isLub\ R\ S\ x$   
*<proof>*

**lemma** *isLubI2*:  $isUb\ R\ S\ x \implies x \leq_*\ Collect\ (isUb\ R\ S) \implies isLub\ R\ S\ x$   
*<proof>*

**lemma** *isUbD*:  $isUb\ R\ S\ x \implies y \in S \implies y \leq x$   
*<proof>*

**lemma** *isUbD2*:  $isUb\ R\ S\ x \implies S \leq_*\ x$   
*<proof>*

**lemma** *isUbD2a*:  $isUb\ R\ S\ x \implies x \in R$   
*<proof>*

**lemma** *isUbI*:  $S \leq_*\ x \implies x \in R \implies isUb\ R\ S\ x$   
*<proof>*

**lemma** *isLub-le-isUb*:  $isLub\ R\ S\ x \implies isUb\ R\ S\ y \implies x \leq y$   
*<proof>*

**lemma** *isLub-ubs*:  $isLub\ R\ S\ x \implies x \leq_*\ ubs\ R\ S$   
*<proof>*

**lemma** *isLub-unique*:  $[ [ isLub\ R\ S\ x; isLub\ R\ S\ y ] ] \implies x = (y::'a::linorder)$   
*<proof>*

**lemma** *isUb-UNIV-I*:  $(\bigwedge y. y \in S \implies y \leq u) \implies isUb\ UNIV\ S\ u$   
*<proof>*

**definition** *greatestP* ::  $('a \Rightarrow bool) \Rightarrow 'a::ord \Rightarrow bool$   
**where** *greatestP*  $P\ x = (P\ x \wedge Collect\ P\ \leq_*\ x)$

**definition** *isLb* ::  $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$   
**where** *isLb*  $R\ S\ x = (x \leq_*\ S \wedge x \in R)$

**definition** *isGlb* ::  $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$   
**where** *isGlb*  $R\ S\ x = greatestP\ (isLb\ R\ S)\ x$

**definition** *lbs* ::  $'a\ set \Rightarrow 'a::ord\ set \Rightarrow 'a\ set$   
**where** *lbs*  $R\ S = Collect\ (isLb\ R\ S)$

### 63.3 Rules about the Operators *greatestP*, *isLb* and *isGlb*

**lemma** *greatestPD1*:  $\text{greatestP } P \ x \Longrightarrow P \ x$   
*<proof>*

**lemma** *greatestPD2*:  $\text{greatestP } P \ x \Longrightarrow \text{Collect } P \ * \leq x$   
*<proof>*

**lemma** *greatestPD3*:  $\text{greatestP } P \ x \Longrightarrow y \in \text{Collect } P \Longrightarrow x \geq y$   
*<proof>*

**lemma** *isGlbD1*:  $\text{isGlb } R \ S \ x \Longrightarrow x \leq * S$   
*<proof>*

**lemma** *isGlbD1a*:  $\text{isGlb } R \ S \ x \Longrightarrow x \in R$   
*<proof>*

**lemma** *isGlb-isLb*:  $\text{isGlb } R \ S \ x \Longrightarrow \text{isLb } R \ S \ x$   
*<proof>*

**lemma** *isGlbD2*:  $\text{isGlb } R \ S \ x \Longrightarrow y \in S \Longrightarrow y \geq x$   
*<proof>*

**lemma** *isGlbD3*:  $\text{isGlb } R \ S \ x \Longrightarrow \text{greatestP } (\text{isLb } R \ S) \ x$   
*<proof>*

**lemma** *isGlbI1*:  $\text{greatestP } (\text{isLb } R \ S) \ x \Longrightarrow \text{isGlb } R \ S \ x$   
*<proof>*

**lemma** *isGlbI2*:  $\text{isLb } R \ S \ x \Longrightarrow \text{Collect } (\text{isLb } R \ S) \ * \leq x \Longrightarrow \text{isGlb } R \ S \ x$   
*<proof>*

**lemma** *isLbD*:  $\text{isLb } R \ S \ x \Longrightarrow y \in S \Longrightarrow y \geq x$   
*<proof>*

**lemma** *isLbD2*:  $\text{isLb } R \ S \ x \Longrightarrow x \leq * S$   
*<proof>*

**lemma** *isLbD2a*:  $\text{isLb } R \ S \ x \Longrightarrow x \in R$   
*<proof>*

**lemma** *isLbI*:  $x \leq * S \Longrightarrow x \in R \Longrightarrow \text{isLb } R \ S \ x$   
*<proof>*

**lemma** *isGlb-le-isLb*:  $\text{isGlb } R \ S \ x \Longrightarrow \text{isLb } R \ S \ y \Longrightarrow x \geq y$   
*<proof>*

**lemma** *isGlb-ubs*:  $\text{isGlb } R \ S \ x \Longrightarrow \text{lbs } R \ S \ * \leq x$   
*<proof>*

**lemma** *isGlb-unique*:  $[[ \text{isGlb } R \ S \ x; \text{isGlb } R \ S \ y ]] \implies x = (y::'a::\text{linorder})$   
 ⟨proof⟩

**lemma** *bdd-above-setle*:  $\text{bdd-above } A \longleftrightarrow (\exists a. A \ *<= \ a)$   
 ⟨proof⟩

**lemma** *bdd-below-setge*:  $\text{bdd-below } A \longleftrightarrow (\exists a. a \ <=* \ A)$   
 ⟨proof⟩

**lemma** *isLub-cSup*:  
 $(S::'a :: \text{conditionally-complete-lattice set}) \neq \{\} \implies (\exists b. S \ *<= \ b) \implies \text{isLub } UNIV \ S \ (\text{Sup } S)$   
 ⟨proof⟩

**lemma** *isGlb-cInf*:  
 $(S::'a :: \text{conditionally-complete-lattice set}) \neq \{\} \implies (\exists b. b \ <=* \ S) \implies \text{isGlb } UNIV \ S \ (\text{Inf } S)$   
 ⟨proof⟩

**lemma** *cSup-le*:  $(S::'a::\text{conditionally-complete-lattice set}) \neq \{\} \implies S \ *<= \ b \implies \text{Sup } S \leq b$   
 ⟨proof⟩

**lemma** *cInf-ge*:  $(S::'a :: \text{conditionally-complete-lattice set}) \neq \{\} \implies b \ <=* \ S \implies \text{Inf } S \geq b$   
 ⟨proof⟩

**lemma** *cSup-bounds*:  
**fixes**  $S :: 'a :: \text{conditionally-complete-lattice set}$   
**shows**  $S \neq \{\} \implies a \ <=* \ S \implies S \ *<= \ b \implies a \leq \text{Sup } S \wedge \text{Sup } S \leq b$   
 ⟨proof⟩

**lemma** *cSup-unique*:  $(S::'a :: \{\text{conditionally-complete-linorder, no-bot}\} \text{ set}) \ *<= \ b \implies (\forall b' < b. \exists x \in S. b' < x) \implies \text{Sup } S = b$   
 ⟨proof⟩

**lemma** *cInf-unique*:  $b \ <=* \ (S::'a :: \{\text{conditionally-complete-linorder, no-top}\} \text{ set}) \implies (\forall b' > b. \exists x \in S. b' > x) \implies \text{Inf } S = b$   
 ⟨proof⟩

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

**lemma** *reals-complete*:  $\exists X. X \in S \implies \exists Y. \text{isUb } (UNIV::\text{real set}) \ S \ Y \implies \exists t. \text{isLub } (UNIV :: \text{real set}) \ S \ t$   
 ⟨proof⟩

**lemma** *Bseq-isUb*:  $\bigwedge X :: \text{nat} \Rightarrow \text{real}. \text{Bseq } X \implies \exists U. \text{isUb } (UNIV::\text{real set}) \ \{x. \exists n. X \ n = x\} \ U$   
 ⟨proof⟩

**lemma** *Bseq-isLub*:  $\bigwedge X :: \text{nat} \Rightarrow \text{real}. \text{Bseq } X \Longrightarrow \exists U. \text{isLub } (UNIV :: \text{real set})$   
 $\{x. \exists n. X \ n = x\} \ U$   
 ⟨proof⟩

**lemma** *isLub-mono-imp-LIMSEQ*:  
**fixes**  $X :: \text{nat} \Rightarrow \text{real}$   
**assumes**  $u: \text{isLub } UNIV \ \{x. \exists n. X \ n = x\} \ u$   
**assumes**  $X: \forall m \ n. m \leq n \longrightarrow X \ m \leq X \ n$   
**shows**  $X \longrightarrow u$   
 ⟨proof⟩

**lemmas** *real-isGlb-unique* = *isGlb-unique*[**where** 'a=real]

**lemma** *real-le-inf-subset*:  $t \neq \{\}$   $\Longrightarrow t \subseteq s \Longrightarrow \exists b. b \leq^* s \Longrightarrow \text{Inf } s \leq \text{Inf } t$   
 (t::real set)  
 ⟨proof⟩

**lemma** *real-ge-sup-subset*:  $t \neq \{\}$   $\Longrightarrow t \subseteq s \Longrightarrow \exists b. s \leq^* b \Longrightarrow \text{Sup } s \geq \text{Sup } t$   
 (t::real set)  
 ⟨proof⟩

end

## 64 An abstract view on maps for code generation.

**theory** *Mapping*  
**imports** *Main AList*  
**begin**

### 64.1 Parametricity transfer rules

**lemma** *map-of-foldr*:  $\text{map-of } xs = \text{foldr } (\lambda(k, v) \ m. \ m(k \mapsto v)) \ xs \ \text{Map.empty}$   
 ⟨proof⟩

**context includes** *lifting-syntax*  
**begin**

**lemma** *empty-parametric*:  $(A \text{ ===> } \text{rel-option } B) \ \text{Map.empty} \ \text{Map.empty}$   
 ⟨proof⟩

**lemma** *lookup-parametric*:  $((A \text{ ===> } B) \text{ ===> } A \text{ ===> } B) \ (\lambda m \ k. \ m \ k) \ (\lambda m \ k. \ m \ k)$   
 ⟨proof⟩

**lemma** *update-parametric*:  
**assumes** [*transfer-rule*]: *bi-unique*  $A$   
**shows**  $(A \text{ ===> } B \text{ ===> } (A \text{ ===> } \text{rel-option } B) \text{ ===> } A \text{ ===> } \text{rel-option } B)$



( $\lambda k v m. m(k \mapsto v)$ ) ( $\lambda k v m. m(k \mapsto v)$ )  
 ⟨proof⟩

**lemma** *delete-parametric*:

**assumes** [*transfer-rule*]: *bi-unique A*

**shows** ( $A \text{====>} (A \text{====>} \textit{rel-option B}) \text{====>} A \text{====>} \textit{rel-option B}$ )

( $\lambda k m. m(k := \textit{None})$ ) ( $\lambda k m. m(k := \textit{None})$ )

⟨proof⟩

**lemma** *is-none-parametric* [*transfer-rule*]:

( $\textit{rel-option A} \text{====>} \textit{HOL.eq} \textit{Option.is-none Option.is-none}$ )

⟨proof⟩

**lemma** *dom-parametric*:

**assumes** [*transfer-rule*]: *bi-total A*

**shows** ( $(A \text{====>} \textit{rel-option B}) \text{====>} \textit{rel-set A} \textit{dom dom}$ )

⟨proof⟩

**lemma** *graph-parametric*:

**assumes** *bi-total A*

**shows** ( $(A \text{====>} \textit{rel-option B}) \text{====>} \textit{rel-set (rel-prod A B)} \textit{Map.graph Map.graph}$ )

⟨proof⟩

**lemma** *map-of-parametric* [*transfer-rule*]:

**assumes** [*transfer-rule*]: *bi-unique R1*

**shows** ( $\textit{list-all2 (rel-prod R1 R2)} \text{====>} R1 \text{====>} \textit{rel-option R2}$ ) *map-of*

⟨proof⟩

**lemma** *map-entry-parametric* [*transfer-rule*]:

**assumes** [*transfer-rule*]: *bi-unique A*

**shows** ( $A \text{====>} (B \text{====>} B) \text{====>} (A \text{====>} \textit{rel-option B}) \text{====>} A$   
 $\text{====>} \textit{rel-option B}$ )

( $\lambda k f m. (\textit{case m k of None} \Rightarrow m$   
 |  $\textit{Some v} \Rightarrow m (k \mapsto (f v))))$  ( $\lambda k f m. (\textit{case m k of None} \Rightarrow m$   
 |  $\textit{Some v} \Rightarrow m (k \mapsto (f v))))$ )

⟨proof⟩

**lemma** *tabulate-parametric*:

**assumes** [*transfer-rule*]: *bi-unique A*

**shows** ( $\textit{list-all2 A} \text{====>} (A \text{====>} B) \text{====>} A \text{====>} \textit{rel-option B}$ )

( $\lambda ks f. (\textit{map-of (map (\lambda k. (k, f k)) ks)})$ ) ( $\lambda ks f. (\textit{map-of (map (\lambda k. (k, f k))$   
 $ks))$ )

⟨proof⟩

**lemma** *bulkload-parametric*:

( $\textit{list-all2 A} \text{====>} \textit{HOL.eq} \text{====>} \textit{rel-option A}$ )

( $\lambda xs k. \textit{if } k < \textit{length xs} \textit{ then Some (xs ! } k) \textit{ else None}$ )

( $\lambda xs k. \textit{if } k < \textit{length xs} \textit{ then Some (xs ! } k) \textit{ else None}$ )

⟨proof⟩

**lemma** *map-parametric*:

((A ==> B) ==> (C ==> D) ==> (B ==> rel-option C) ==> A  
 ==> rel-option D)  
 (λf g m. (map-option g ∘ m ∘ f)) (λf g m. (map-option g ∘ m ∘ f))  
 ⟨proof⟩

**lemma** *combine-with-key-parametric*:

((A ==> B ==> B ==> B) ==> (A ==> rel-option B) ==> (A  
 ==> rel-option B) ==>  
 (A ==> rel-option B)) (λf m1 m2 x. combine-options (f x) (m1 x) (m2 x))  
 (λf m1 m2 x. combine-options (f x) (m1 x) (m2 x))  
 ⟨proof⟩

**lemma** *combine-parametric*:

((B ==> B ==> B) ==> (A ==> rel-option B) ==> (A ==>  
 rel-option B) ==>  
 (A ==> rel-option B)) (λf m1 m2 x. combine-options f (m1 x) (m2 x))  
 (λf m1 m2 x. combine-options f (m1 x) (m2 x))  
 ⟨proof⟩

end

## 64.2 Type definition and primitive operations

**typedef** ('a, 'b) *mapping* = UNIV :: ('a → 'b) set  
**morphisms** *rep Mapping* ⟨proof⟩

**setup-lifting** *type-definition-mapping*

**lift-definition** *empty* :: ('a, 'b) *mapping*  
**is** *Map.empty* **parametric** *empty-parametric* ⟨proof⟩

**lift-definition** *lookup* :: ('a, 'b) *mapping* ⇒ 'a ⇒ 'b *option*  
**is** λm k. m k **parametric** *lookup-parametric* ⟨proof⟩

**definition** *lookup-default* d m k = (case *Mapping.lookup* m k of None ⇒ d | Some  
 v ⇒ v)

**lift-definition** *update* :: 'a ⇒ 'b ⇒ ('a, 'b) *mapping* ⇒ ('a, 'b) *mapping*  
**is** λk v m. m(k ↦ v) **parametric** *update-parametric* ⟨proof⟩

**lift-definition** *delete* :: 'a ⇒ ('a, 'b) *mapping* ⇒ ('a, 'b) *mapping*  
**is** λk m. m(k := None) **parametric** *delete-parametric* ⟨proof⟩

**lift-definition** *filter* :: ('a ⇒ 'b ⇒ bool) ⇒ ('a, 'b) *mapping* ⇒ ('a, 'b) *mapping*  
**is** λP m k. case m k of None ⇒ None | Some v ⇒ if P k v then Some v else None  
 ⟨proof⟩

**lift-definition** *keys* :: ('a, 'b) mapping  $\Rightarrow$  'a set  
 is **dom parametric dom-parametric** <proof>

**lift-definition** *entries* :: ('a, 'b) mapping  $\Rightarrow$  ('a  $\times$  'b) set  
 is **Map.graph parametric graph-parametric** <proof>

**lift-definition** *tabulate* :: 'a list  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a, 'b) mapping  
 is  $\lambda ks f. (map-of (List.map (\lambda k. (k, f k)) ks))$  **parametric tabulate-parametric**  
 <proof>

**lift-definition** *bulkload* :: 'a list  $\Rightarrow$  (nat, 'a) mapping  
 is  $\lambda xs k. \text{if } k < \text{length } xs \text{ then } \text{Some } (xs ! k) \text{ else } \text{None}$  **parametric bulk-**  
**load-parametric** <proof>

**lift-definition** *map* :: ('c  $\Rightarrow$  'a)  $\Rightarrow$  ('b  $\Rightarrow$  'd)  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('c, 'd) mapping  
 is  $\lambda f g m. (map-option g \circ m \circ f)$  **parametric map-parametric** <proof>

**lift-definition** *map-values* :: ('c  $\Rightarrow$  'a  $\Rightarrow$  'b)  $\Rightarrow$  ('c, 'a) mapping  $\Rightarrow$  ('c, 'b) mapping  
 is  $\lambda f m x. map-option (f x) (m x)$  <proof>

**lift-definition** *combine-with-key* ::  
 ('a  $\Rightarrow$  'b  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('a, 'b) mapping  
 is  $\lambda f m1 m2 x. combine-options (f x) (m1 x) (m2 x)$  **parametric combine-with-key-parametric**  
 <proof>

**lift-definition** *combine* ::  
 ('b  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('a, 'b) mapping  $\Rightarrow$  ('a, 'b) mapping  
 is  $\lambda f m1 m2 x. combine-options f (m1 x) (m2 x)$  **parametric combine-parametric**  
 <proof>

**definition** *All-mapping*  $m P \longleftrightarrow$   
 ( $\forall x. \text{case } Mapping.lookup m x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } y \Rightarrow P x y$ )

**declare** [[code drop: map]]

### 64.3 Functorial structure

**functor** *map*: map  
 <proof>

### 64.4 Derived operations

**definition** *ordered-keys* :: ('a::linorder, 'b) mapping  $\Rightarrow$  'a list  
 where *ordered-keys*  $m = (\text{if finite } (keys m) \text{ then sorted-list-of-set } (keys m) \text{ else } [])$

**definition** *ordered-entries* :: ('a::linorder, 'b) mapping  $\Rightarrow$  ('a  $\times$  'b) list  
 where *ordered-entries*  $m = (\text{if finite } (entries m) \text{ then sorted-key-list-of-set fst } (entries m))$

else [])

**definition** *fold* :: ('a::linorder ⇒ 'b ⇒ 'c ⇒ 'c) ⇒ ('a, 'b) mapping ⇒ 'c ⇒ 'c  
**where** *fold* f m a = List.fold (case-prod f) (ordered-entries m) a

**definition** *is-empty* :: ('a, 'b) mapping ⇒ bool  
**where** *is-empty* m ↔ keys m = {}

**definition** *size* :: ('a, 'b) mapping ⇒ nat  
**where** *size* m = (if finite (keys m) then card (keys m) else 0)

**definition** *replace* :: 'a ⇒ 'b ⇒ ('a, 'b) mapping ⇒ ('a, 'b) mapping  
**where** *replace* k v m = (if k ∈ keys m then update k v m else m)

**definition** *default* :: 'a ⇒ 'b ⇒ ('a, 'b) mapping ⇒ ('a, 'b) mapping  
**where** *default* k v m = (if k ∈ keys m then m else update k v m)

Manual derivation of transfer rule is non-trivial

**lift-definition** *map-entry* :: 'a ⇒ ('b ⇒ 'b) ⇒ ('a, 'b) mapping ⇒ ('a, 'b) mapping  
**is**

λk f m.  
 (case m k of  
   None ⇒ m  
   | Some v ⇒ m (k ↦ (f v))) **parametric** *map-entry-parametric* ⟨proof⟩

**lemma** *map-entry-code* [code]:

*map-entry* k f m =  
 (case lookup m k of  
   None ⇒ m  
   | Some v ⇒ update k (f v) m)  
 ⟨proof⟩

**definition** *map-default* :: 'a ⇒ 'b ⇒ ('b ⇒ 'b) ⇒ ('a, 'b) mapping ⇒ ('a, 'b) mapping  
**where** *map-default* k v f m = *map-entry* k f (default k v m)

**definition** *of-alist* :: ('k × 'v) list ⇒ ('k, 'v) mapping  
**where** *of-alist* xs = foldr (λ(k, v) m. update k v m) xs empty

**instantiation** *mapping* :: (type, type) equal  
**begin**

**definition** *HOL.equal* m1 m2 ↔ (∀ k. lookup m1 k = lookup m2 k)

**instance**  
 ⟨proof⟩

**end**

**context includes** *lifting-syntax*  
**begin**

**lemma** [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *bi-total A*  
**and** [*transfer-rule*]: *bi-unique B*  
**shows** (*pcr-mapping A B*  $\implies$  *pcr-mapping A B*  $\implies$  (=)) *HOL.eq HOL.equal*  
 ⟨*proof*⟩

**lemma** *of-alist-transfer* [*transfer-rule*]:  
**assumes** [*transfer-rule*]: *bi-unique R1*  
**shows** (*list-all2 (rel-prod R1 R2)*  $\implies$  *pcr-mapping R1 R2*) *map-of of-alist*  
 ⟨*proof*⟩

**end**

## 64.5 Properties

**lemma** *mapping-eqI*:  $(\bigwedge x. \text{lookup } m \ x = \text{lookup } m' \ x) \implies m = m'$   
 ⟨*proof*⟩

**lemma** *mapping-eqI'*:  
**assumes**  $\bigwedge x. x \in \text{Mapping.keys } m \implies \text{Mapping.lookup-default } d \ m \ x = \text{Mapping.lookup-default } d \ m' \ x$   
**and**  $\text{Mapping.keys } m = \text{Mapping.keys } m'$   
**shows**  $m = m'$   
 ⟨*proof*⟩

**lemma** *lookup-update[simp]*:  $\text{lookup } (\text{update } k \ v \ m) \ k = \text{Some } v$   
 ⟨*proof*⟩

**lemma** *lookup-update-neq[simp]*:  $k \neq k' \implies \text{lookup } (\text{update } k \ v \ m) \ k' = \text{lookup } m \ k'$   
 ⟨*proof*⟩

**lemma** *lookup-update'*:  $\text{lookup } (\text{update } k \ v \ m) \ k' = (\text{if } k = k' \ \text{then } \text{Some } v \ \text{else } \text{lookup } m \ k')$   
 ⟨*proof*⟩

**lemma** *lookup-empty[simp]*:  $\text{lookup } \text{empty} \ k = \text{None}$   
 ⟨*proof*⟩

**lemma** *lookup-delete[simp]*:  $\text{lookup } (\text{delete } k \ m) \ k = \text{None}$   
 ⟨*proof*⟩

**lemma** *lookup-delete-neq[simp]*:  $k \neq k' \implies \text{lookup } (\text{delete } k \ m) \ k' = \text{lookup } m \ k'$   
 ⟨*proof*⟩

**lemma** *lookup-filter*:

*lookup* (*filter P m*) *k* =  
 (case *lookup m k* of  
   None  $\Rightarrow$  None  
   | Some *v*  $\Rightarrow$  if *P k v* then Some *v* else None)  
 ⟨proof⟩

**lemma** *lookup-map-values*: *lookup* (*map-values f m*) *k* = *map-option* (*f k*) (*lookup m k*)  
 ⟨proof⟩

**lemma** *lookup-default-empty*: *lookup-default d empty k* = *d*  
 ⟨proof⟩

**lemma** *lookup-default-update*: *lookup-default d* (*update k v m*) *k* = *v*  
 ⟨proof⟩

**lemma** *lookup-default-update-neq*:  
*k*  $\neq$  *k'*  $\implies$  *lookup-default d* (*update k v m*) *k'* = *lookup-default d m k'*  
 ⟨proof⟩

**lemma** *lookup-default-update'*:  
*lookup-default d* (*update k v m*) *k'* = (if *k* = *k'* then *v* else *lookup-default d m k'*)  
 ⟨proof⟩

**lemma** *lookup-default-filter*:  
*lookup-default d* (*filter P m*) *k* =  
 (if *P k* (*lookup-default d m k*) then *lookup-default d m k* else *d*)  
 ⟨proof⟩

**lemma** *lookup-default-map-values*:  
*lookup-default* (*f k d*) (*map-values f m*) *k* = *f k* (*lookup-default d m k*)  
 ⟨proof⟩

**lemma** *lookup-combine-with-key*:  
*Mapping.lookup* (*combine-with-key f m1 m2*) *x* =  
*combine-options* (*f x*) (*Mapping.lookup m1 x*) (*Mapping.lookup m2 x*)  
 ⟨proof⟩

**lemma** *combine-altdef*: *combine f m1 m2* = *combine-with-key* ( $\lambda\cdot$ . *f*) *m1 m2*  
 ⟨proof⟩

**lemma** *lookup-combine*:  
*Mapping.lookup* (*combine f m1 m2*) *x* =  
*combine-options* *f* (*Mapping.lookup m1 x*) (*Mapping.lookup m2 x*)  
 ⟨proof⟩

**lemma** *lookup-default-neutral-combine-with-key*:  
**assumes**  $\bigwedge x. f k d x = x \bigwedge x. f k x d = x$   
**shows** *Mapping.lookup-default d* (*combine-with-key f m1 m2*) *k* =

$f\ k\ (Mapping.lookup-default\ d\ m1\ k)\ (Mapping.lookup-default\ d\ m2\ k)$   
 ⟨proof⟩

**lemma** *lookup-default-neutral-combine*:

**assumes**  $\bigwedge x. f\ d\ x = x \wedge x. f\ x\ d = x$

**shows**  $Mapping.lookup-default\ d\ (combine\ f\ m1\ m2)\ x =$

$f\ (Mapping.lookup-default\ d\ m1\ x)\ (Mapping.lookup-default\ d\ m2\ x)$   
 ⟨proof⟩

**lemma** *lookup-map-entry*:  $lookup\ (map-entry\ x\ f\ m)\ x = map-option\ f\ (lookup\ m\ x)$

⟨proof⟩

**lemma** *lookup-map-entry-neq*:  $x \neq y \implies lookup\ (map-entry\ x\ f\ m)\ y = lookup\ m\ y$

⟨proof⟩

**lemma** *lookup-map-entry'*:

$lookup\ (map-entry\ x\ f\ m)\ y =$

$(if\ x = y\ then\ map-option\ f\ (lookup\ m\ y)\ else\ lookup\ m\ y)$

⟨proof⟩

**lemma** *lookup-default*:  $lookup\ (default\ x\ d\ m)\ x = Some\ (lookup-default\ d\ m\ x)$

⟨proof⟩

**lemma** *lookup-default-neq*:  $x \neq y \implies lookup\ (default\ x\ d\ m)\ y = lookup\ m\ y$

⟨proof⟩

**lemma** *lookup-default'*:

$lookup\ (default\ x\ d\ m)\ y =$

$(if\ x = y\ then\ Some\ (lookup-default\ d\ m\ x)\ else\ lookup\ m\ y)$

⟨proof⟩

**lemma** *lookup-map-default*:  $lookup\ (map-default\ x\ d\ f\ m)\ x = Some\ (f\ (lookup-default\ d\ m\ x))$

⟨proof⟩

**lemma** *lookup-map-default-neq*:  $x \neq y \implies lookup\ (map-default\ x\ d\ f\ m)\ y = lookup\ m\ y$

⟨proof⟩

**lemma** *lookup-map-default'*:

$lookup\ (map-default\ x\ d\ f\ m)\ y =$

$(if\ x = y\ then\ Some\ (f\ (lookup-default\ d\ m\ x))\ else\ lookup\ m\ y)$

⟨proof⟩

**lemma** *lookup-tabulate*:

**assumes** *distinct xs*

**shows**  $Mapping.lookup\ (Mapping.tabulate\ xs\ f)\ x = (if\ x \in set\ xs\ then\ Some\ (f$

$x$ ) else None)  
 ⟨proof⟩

**lemma** *lookup-of-alist*:  $\text{lookup (of-alist xs) k} = \text{map-of xs k}$   
 ⟨proof⟩

**lemma** *keys-is-none-rep* [code-unfold]:  $k \in \text{keys } m \iff \neg (\text{Option.is-none (lookup } m \text{ k)})$   
 ⟨proof⟩

**lemma** *update-update*:  
 $\text{update } k \text{ } v \text{ (update } k \text{ } w \text{ } m) = \text{update } k \text{ } v \text{ } m$   
 $k \neq l \implies \text{update } k \text{ } v \text{ (update } l \text{ } w \text{ } m) = \text{update } l \text{ } w \text{ (update } k \text{ } v \text{ } m)$   
 ⟨proof⟩

**lemma** *update-delete* [simp]:  $\text{update } k \text{ } v \text{ (delete } k \text{ } m) = \text{update } k \text{ } v \text{ } m$   
 ⟨proof⟩

**lemma** *delete-update*:  
 $\text{delete } k \text{ (update } k \text{ } v \text{ } m) = \text{delete } k \text{ } m$   
 $k \neq l \implies \text{delete } k \text{ (update } l \text{ } v \text{ } m) = \text{update } l \text{ } v \text{ (delete } k \text{ } m)$   
 ⟨proof⟩

**lemma** *delete-empty* [simp]:  $\text{delete } k \text{ empty} = \text{empty}$   
 ⟨proof⟩

**lemma** *Mapping-delete-if-notin-keys*[simp]:  
 $k \notin \text{keys } m \implies \text{delete } k \text{ } m = m$   
 ⟨proof⟩

**lemma** *replace-update*:  
 $k \notin \text{keys } m \implies \text{replace } k \text{ } v \text{ } m = m$   
 $k \in \text{keys } m \implies \text{replace } k \text{ } v \text{ } m = \text{update } k \text{ } v \text{ } m$   
 ⟨proof⟩

**lemma** *map-values-update*:  $\text{map-values } f \text{ (update } k \text{ } v \text{ } m) = \text{update } k \text{ (} f \text{ } v \text{) (map-values } f \text{ } m)$   
 ⟨proof⟩

**lemma** *size-mono*:  $\text{finite (keys } m') \implies \text{keys } m \subseteq \text{keys } m' \implies \text{size } m \leq \text{size } m'$   
 ⟨proof⟩

**lemma** *size-empty* [simp]:  $\text{size empty} = 0$   
 ⟨proof⟩

**lemma** *size-update*:  
 $\text{finite (keys } m) \implies \text{size (update } k \text{ } v \text{ } m) =$   
 (if  $k \in \text{keys } m$  then  $\text{size } m$  else  $\text{Suc (size } m)$ )  
 ⟨proof⟩



**lemma** *size-delete*:  $\text{size} (\text{delete } k \ m) = (\text{if } k \in \text{keys } m \text{ then } \text{size } m - 1 \text{ else } \text{size } m)$   
 ⟨proof⟩

**lemma** *size-tabulate* [simp]:  $\text{size} (\text{tabulate } ks \ f) = \text{length} (\text{remdups } ks)$   
 ⟨proof⟩

**lemma** *keys-filter*:  $\text{keys} (\text{filter } P \ m) \subseteq \text{keys } m$   
 ⟨proof⟩

**lemma** *size-filter*:  $\text{finite} (\text{keys } m) \implies \text{size} (\text{filter } P \ m) \leq \text{size } m$   
 ⟨proof⟩

**lemma** *bulkload-tabulate*:  $\text{bulkload } xs = \text{tabulate } [0..<\text{length } xs] (\text{nth } xs)$   
 ⟨proof⟩

**lemma** *is-empty-empty* [simp]:  $\text{is-empty } \text{empty}$   
 ⟨proof⟩

**lemma** *is-empty-update* [simp]:  $\neg \text{is-empty} (\text{update } k \ v \ m)$   
 ⟨proof⟩

**lemma** *is-empty-delete*:  $\text{is-empty} (\text{delete } k \ m) \longleftrightarrow \text{is-empty } m \vee \text{keys } m = \{k\}$   
 ⟨proof⟩

**lemma** *is-empty-replace* [simp]:  $\text{is-empty} (\text{replace } k \ v \ m) \longleftrightarrow \text{is-empty } m$   
 ⟨proof⟩

**lemma** *is-empty-default* [simp]:  $\neg \text{is-empty} (\text{default } k \ v \ m)$   
 ⟨proof⟩

**lemma** *is-empty-map-entry* [simp]:  $\text{is-empty} (\text{map-entry } k \ f \ m) \longleftrightarrow \text{is-empty } m$   
 ⟨proof⟩

**lemma** *is-empty-map-values* [simp]:  $\text{is-empty} (\text{map-values } f \ m) \longleftrightarrow \text{is-empty } m$   
 ⟨proof⟩

**lemma** *is-empty-map-default* [simp]:  $\neg \text{is-empty} (\text{map-default } k \ v \ f \ m)$   
 ⟨proof⟩

**lemma** *keys-dom-lookup*:  $\text{keys } m = \text{dom} (\text{Mapping.lookup } m)$   
 ⟨proof⟩

**lemma** *keys-empty* [simp]:  $\text{keys } \text{empty} = \{\}$   
 ⟨proof⟩

**lemma** *in-keysD*:  $k \in \text{keys } m \implies \exists v. \text{lookup } m \ k = \text{Some } v$   
 ⟨proof⟩

**lemma** *keys-update* [*simp*]:  $keys (update\ k\ v\ m) = insert\ k\ (keys\ m)$   
 ⟨*proof*⟩

**lemma** *keys-delete* [*simp*]:  $keys (delete\ k\ m) = keys\ m - \{k\}$   
 ⟨*proof*⟩

**lemma** *keys-replace* [*simp*]:  $keys (replace\ k\ v\ m) = keys\ m$   
 ⟨*proof*⟩

**lemma** *keys-default* [*simp*]:  $keys (default\ k\ v\ m) = insert\ k\ (keys\ m)$   
 ⟨*proof*⟩

**lemma** *keys-map-entry* [*simp*]:  $keys (map-entry\ k\ f\ m) = keys\ m$   
 ⟨*proof*⟩

**lemma** *keys-map-default* [*simp*]:  $keys (map-default\ k\ v\ f\ m) = insert\ k\ (keys\ m)$   
 ⟨*proof*⟩

**lemma** *keys-map-values* [*simp*]:  $keys (map-values\ f\ m) = keys\ m$   
 ⟨*proof*⟩

**lemma** *keys-combine-with-key* [*simp*]:  
 $Mapping.keys (combine-with-key\ f\ m1\ m2) = Mapping.keys\ m1 \cup Mapping.keys\ m2$   
 ⟨*proof*⟩

**lemma** *keys-combine* [*simp*]:  $Mapping.keys (combine\ f\ m1\ m2) = Mapping.keys\ m1 \cup Mapping.keys\ m2$   
 ⟨*proof*⟩

**lemma** *keys-tabulate* [*simp*]:  $keys (tabulate\ ks\ f) = set\ ks$   
 ⟨*proof*⟩

**lemma** *keys-of-alist* [*simp*]:  $keys (of-alist\ xs) = set (List.map\ fst\ xs)$   
 ⟨*proof*⟩

**lemma** *keys-bulkload* [*simp*]:  $keys (bulkload\ xs) = \{0..<length\ xs\}$   
 ⟨*proof*⟩

**lemma** *finite-keys-update*[*simp*]:  
 $finite (keys (update\ k\ v\ m)) = finite (keys\ m)$   
 ⟨*proof*⟩

**lemma** *set-ordered-keys*[*simp*]:  
 $finite (Mapping.keys\ m) \implies set (Mapping.ordered-keys\ m) = Mapping.keys\ m$   
 ⟨*proof*⟩

**lemma** *distinct-ordered-keys* [*simp*]:  $distinct (ordered-keys\ m)$   
 ⟨*proof*⟩

**lemma** *ordered-keys-infinite* [simp]:  $\neg \text{finite } (\text{keys } m) \implies \text{ordered-keys } m = []$   
 ⟨proof⟩

**lemma** *ordered-keys-empty* [simp]:  $\text{ordered-keys } \text{empty} = []$   
 ⟨proof⟩

**lemma** *sorted-ordered-keys*[simp]:  $\text{sorted } (\text{ordered-keys } m)$   
 ⟨proof⟩

**lemma** *ordered-keys-update* [simp]:  
 $k \in \text{keys } m \implies \text{ordered-keys } (\text{update } k \ v \ m) = \text{ordered-keys } m$   
 $\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies$   
 $\text{ordered-keys } (\text{update } k \ v \ m) = \text{insert } k \ (\text{ordered-keys } m)$   
 ⟨proof⟩

**lemma** *ordered-keys-delete* [simp]:  $\text{ordered-keys } (\text{delete } k \ m) = \text{remove1 } k \ (\text{ordered-keys } m)$   
 ⟨proof⟩

**lemma** *ordered-keys-replace* [simp]:  $\text{ordered-keys } (\text{replace } k \ v \ m) = \text{ordered-keys } m$   
 ⟨proof⟩

**lemma** *ordered-keys-default* [simp]:  
 $k \in \text{keys } m \implies \text{ordered-keys } (\text{default } k \ v \ m) = \text{ordered-keys } m$   
 $\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies \text{ordered-keys } (\text{default } k \ v \ m) = \text{insert } k \ (\text{ordered-keys } m)$   
 ⟨proof⟩

**lemma** *ordered-keys-map-entry* [simp]:  $\text{ordered-keys } (\text{map-entry } k \ f \ m) = \text{ordered-keys } m$   
 ⟨proof⟩

**lemma** *ordered-keys-map-default* [simp]:  
 $k \in \text{keys } m \implies \text{ordered-keys } (\text{map-default } k \ v \ f \ m) = \text{ordered-keys } m$   
 $\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies \text{ordered-keys } (\text{map-default } k \ v \ f \ m) = \text{insert } k \ (\text{ordered-keys } m)$   
 ⟨proof⟩

**lemma** *ordered-keys-tabulate* [simp]:  $\text{ordered-keys } (\text{tabulate } ks \ f) = \text{sort } (\text{remdups } ks)$   
 ⟨proof⟩

**lemma** *ordered-keys-bulkload* [simp]:  $\text{ordered-keys } (\text{bulkload } ks) = [0..<\text{length } ks]$   
 ⟨proof⟩

**lemma** *tabulate-fold*:  $\text{tabulate } xs \ f = \text{List.fold } (\lambda k \ m. \text{update } k \ (f \ k) \ m) \ xs \ \text{empty}$   
 ⟨proof⟩

**lemma** *All-mapping-mono:*

$(\bigwedge k v. k \in \text{keys } m \implies P k v \implies Q k v) \implies \text{All-mapping } m P \implies \text{All-mapping } m Q$   
 ⟨proof⟩

**lemma** *All-mapping-empty [simp]: All-mapping Mapping.empty P*

⟨proof⟩

**lemma** *All-mapping-update-iff:*

$\text{All-mapping } (\text{Mapping.update } k v m) P \longleftrightarrow P k v \wedge \text{All-mapping } m (\lambda k' v'. k = k' \vee P k' v')$   
 ⟨proof⟩

**lemma** *All-mapping-update:*

$P k v \implies \text{All-mapping } m (\lambda k' v'. k = k' \vee P k' v') \implies \text{All-mapping } (\text{Mapping.update } k v m) P$   
 ⟨proof⟩

**lemma** *All-mapping-filter-iff: All-mapping (filter P m) Q  $\longleftrightarrow$  All-mapping m ( $\lambda k v. P k v \longrightarrow Q k v$ )*

⟨proof⟩

**lemma** *All-mapping-filter: All-mapping m Q  $\implies$  All-mapping (filter P m) Q*

⟨proof⟩

**lemma** *All-mapping-map-values: All-mapping (map-values f m) P  $\longleftrightarrow$  All-mapping m ( $\lambda k v. P k (f k v)$ )*

⟨proof⟩

**lemma** *All-mapping-tabulate: ( $\forall x \in \text{set } xs. P x (f x)$ )  $\implies$  All-mapping (Mapping.tabulate xs f) P*

⟨proof⟩

**lemma** *All-mapping-alist:*

$(\bigwedge k v. (k, v) \in \text{set } xs \implies P k v) \implies \text{All-mapping } (\text{Mapping.of-alist } xs) P$   
 ⟨proof⟩

**lemma** *combine-empty [simp]: combine f Mapping.empty y = y combine f y Mapping.empty = y*

⟨proof⟩

**lemma** (*in abel-semigroup*) *comm-monoid-set-combine: comm-monoid-set (combine f) Mapping.empty*

⟨proof⟩

**locale** *combine-mapping-abel-semigroup = abel-semigroup*

**begin**

**sublocale** *combine: comm-monoid-set combine f Mapping.empty*

⟨proof⟩

**lemma** *fold-combine-code*:

$combine.F\ g\ (set\ xs) = foldr\ (\lambda x. combine\ f\ (g\ x))\ (remdups\ xs)\ Mapping.empty$   
 ⟨proof⟩

**lemma** *keys-fold-combine*:  $finite\ A \implies Mapping.keys\ (combine.F\ g\ A) = (\bigcup_{x \in A} Mapping.keys\ (g\ x))$

⟨proof⟩

**end**

### 64.5.1 entries, ordered-entries, and fold

**context** *linorder*

**begin**

**sublocale** *folding-Map-graph*: *folding-insort-key* ( $\leq$ ) ( $<$ ) *Map.graph* *m* **fst** **for** *m*  
 ⟨proof⟩

**end**

**lemma** *sorted-fst-list-of-set-insort-Map-graph[simp]*:

**assumes**  $finite\ (dom\ m)\ fst\ x \notin dom\ m$

**shows**  $sorted\_key\_list\_of\_set\ fst\ (insert\ x\ (Map.graph\ m))$

$= insort\_key\ fst\ x\ (sorted\_key\_list\_of\_set\ fst\ (Map.graph\ m))$

⟨proof⟩

**lemma** *sorted-fst-list-of-set-insort-insert-Map-graph[simp]*:

**assumes**  $finite\ (dom\ m)\ fst\ x \notin dom\ m$

**shows**  $sorted\_key\_list\_of\_set\ fst\ (insert\ x\ (Map.graph\ m))$

$= insort\_insert\_key\ fst\ x\ (sorted\_key\_list\_of\_set\ fst\ (Map.graph\ m))$

⟨proof⟩

**lemma** *linorder-finite-Map-induct[consumes 1, case-names empty update]*:

**fixes**  $m :: 'a::linorder \rightarrow 'b$

**assumes**  $finite\ (dom\ m)$

**assumes**  $P\ Map.empty$

**assumes**  $\bigwedge k\ v\ m. \llbracket finite\ (dom\ m); k \notin dom\ m; (\bigwedge k'. k' \in dom\ m \implies k' \leq k); P\ m \rrbracket$

$\implies P\ (m(k \mapsto v))$

**shows**  $P\ m$

⟨proof⟩

**lemma** *delete-insort-fst[simp]*:  $AList.delete\ k\ (insort\_key\ fst\ (k, v)\ xs) = AList.delete\ k\ xs$

⟨proof⟩

**lemma** *insort-fst-delete*:  $\llbracket fst\ x \neq k2; sorted\ (List.map\ fst\ xs) \rrbracket$

$\implies \text{insort-key fst } x \ (\text{AList.delete } k2 \ xs) = \text{AList.delete } k2 \ (\text{insort-key fst } x \ xs)$   
 ⟨proof⟩

**lemma** *sorted-fst-list-of-set-Map-graph-fun-upd-None[simp]*:  
 $\text{sorted-key-list-of-set fst } (\text{Map.graph } (m(k := \text{None})))$   
 $= \text{AList.delete } k \ (\text{sorted-key-list-of-set fst } (\text{Map.graph } m))$   
 ⟨proof⟩

**lemma** *entries-empty[simp]*:  $\text{entries empty} = \{\}$   
 ⟨proof⟩

**lemma** *entries-lookup*:  $\text{entries } m = \text{Map.graph } (\text{lookup } m)$   
 ⟨proof⟩

**lemma** *in-entriesI*:  $\text{lookup } m \ k = \text{Some } v \implies (k, v) \in \text{entries } m$   
 ⟨proof⟩

**lemma** *in-entriesD*:  $(k, v) \in \text{entries } m \implies \text{lookup } m \ k = \text{Some } v$   
 ⟨proof⟩

**lemma** *fst-image-entries-eq-keys[simp]*:  $\text{fst } \text{ ` Mapping.entries } m = \text{Mapping.keys } m$   
 ⟨proof⟩

**lemma** *finite-entries-iff-finite-keys[simp]*:  
 $\text{finite } (\text{entries } m) = \text{finite } (\text{keys } m)$   
 ⟨proof⟩

**lemma** *entries-update*:  
 $\text{entries } (\text{update } k \ v \ m) = \text{insert } (k, v) \ (\text{entries } (\text{delete } k \ m))$   
 ⟨proof⟩

**lemma** *entries-delete*:  
 $\text{entries } (\text{delete } k \ m) = \{e \in \text{entries } m. \text{fst } e \neq k\}$   
 ⟨proof⟩

**lemma** *entries-of-alist[simp]*:  
 $\text{distinct } (\text{List.map fst } xs) \implies \text{entries } (\text{of-alist } xs) = \text{set } xs$   
 ⟨proof⟩

**lemma** *entries-keysD*:  
 $x \in \text{entries } m \implies \text{fst } x \in \text{keys } m$   
 ⟨proof⟩

**lemma** *set-ordered-entries[simp]*:  
 $\text{finite } (\text{keys } m) \implies \text{set } (\text{ordered-entries } m) = \text{entries } m$   
 ⟨proof⟩

**lemma** *distinct-ordered-entries[simp]*:  $\text{distinct } (\text{List.map fst } (\text{ordered-entries } m))$

*<proof>*

**lemma** *sorted-ordered-entries[simp]*: *sorted (List.map fst (ordered-entries m))*  
*<proof>*

**lemma** *ordered-entries-infinite[simp]*:  
 $\neg \text{finite (Mapping.keys } m) \implies \text{ordered-entries } m = []$   
*<proof>*

**lemma** *ordered-entries-empty[simp]*: *ordered-entries empty = []*  
*<proof>*

**lemma** *ordered-entries-update[simp]*:  
**assumes** *finite (keys m)*  
**shows** *ordered-entries (update k v m)*  
 $= \text{insort-insert-key fst (k, v) (AList.delete k (ordered-entries m))}$   
*<proof>*

**lemma** *ordered-entries-delete[simp]*:  
 $\text{ordered-entries (delete k m)} = \text{AList.delete k (ordered-entries m)}$   
*<proof>*

**lemma** *map-fst-ordered-entries[simp]*:  
 $\text{List.map fst (ordered-entries m)} = \text{ordered-keys m}$   
*<proof>*

**lemma** *fold-empty[simp]*: *fold f empty a = a*  
*<proof>*

**lemma** *insort-key-is-snoc-if-sorted-and-distinct*:  
**assumes** *sorted (List.map f xs) f y  $\notin$  f ' set xs  $\forall x \in$  set xs. f x  $\leq$  f y*  
**shows** *insort-key f y xs = xs @ [y]*  
*<proof>*

**lemma** *fold-update*:  
**assumes** *finite (keys m)*  
**assumes**  $k \notin \text{keys } m \wedge k'. k' \in \text{keys } m \implies k' \leq k$   
**shows**  $\text{fold } f \text{ (update } k \text{ v } m) a = f \text{ k v (fold } f \text{ m } a)$   
*<proof>*

**lemma** *linorder-finite-Mapping-induct[consumes 1, case-names empty update]*:  
**fixes**  $m :: ('a::\text{linorder}, 'b) \text{ mapping}$   
**assumes** *finite (keys m)*  
**assumes** *P empty*  
**assumes**  $\wedge k \text{ v } m.$   
 $\llbracket \text{finite (keys } m); k \notin \text{keys } m; (\wedge k'. k' \in \text{keys } m \implies k' \leq k); P \text{ m} \rrbracket$   
 $\implies P \text{ (update } k \text{ v } m)$   
**shows** *P m*  
*<proof>*

## 64.6 Code generator setup

**hide-const** (**open**) *empty is-empty rep lookup lookup-default filter update delete ordered-keys*

*keys size replace default map-entry map-default tabulate bulkload map map-values combine of-alist*

*entries ordered-entries fold*

**end**

## 65 Monad notation for arbitrary types

**theory** *Monad-Syntax*

**imports** *Main*

**begin**

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

**consts**

*bind* :: 'a ⇒ ('b ⇒ 'c) ⇒ 'd (**infixl** <≫=> 54)

**notation** (*ASCII*)

*bind* (**infixl** <>>=> 54)

**abbreviation** (*do-notation*)

*bind-do* :: 'a ⇒ ('b ⇒ 'c) ⇒ 'd

**where** *bind-do* ≡ *bind*

**notation** (**output**)

*bind-do* (**infixl** <≫=> 54)

**notation** (*ASCII output*)

*bind-do* (**infixl** <>>=> 54)

**nonterminal** *do-binds and do-bind*

**syntax**

*-do-block* :: *do-binds* ⇒ 'a

(⟨⟨*open-block notation*=⟨*mixfix do block*⟩⟩do {/(2 -)/}) [12] 62)

*-do-bind* :: [*pttrn*, 'a] ⇒ *do-bind*

(⟨⟨*indent*=2 *notation*=⟨*infix do bind*⟩⟩- ←/ -) 13)

*-do-let* :: [*pttrn*, 'a] ⇒ *do-bind*

(⟨⟨*indent*=2 *notation*=⟨*infix do let*⟩⟩let - =/ -) [1000, 13] 13)

*-do-then* :: 'a ⇒ *do-bind* (⟨-⟩ [14] 13)

*-do-final* :: 'a ⇒ *do-binds* (⟨-⟩)

*-do-cons* :: [*do-bind*, *do-binds*] ⇒ *do-binds*

(⟨⟨*open-block notation*=⟨*infix do next*⟩⟩-;/-⟩ [13, 12] 12)

*-thenM* :: ['a, 'b] ⇒ 'c (**infixl** <≫=> 54)



**syntax** (ASCII)

-do-bind :: [pttrn, 'a] ⇒ do-bind  
 (⟨⟨indent=2 notation=infix do bind⟩⟩- <- / -⟩ 13)  
 -thenM :: ['a, 'b] ⇒ 'c (infixl <>> 54)

**syntax-consts**

-do-block -do-cons -do-bind -do-then ⇒ bind **and**  
 -do-let ⇒ Let

**translations**

-do-block (-do-cons (-do-then t) (-do-final e))  
 ⇒ CONST bind-do t (λ-. e)  
 -do-block (-do-cons (-do-bind p t) (-do-final e))  
 ⇒ CONST bind-do t (λp. e)  
 -do-block (-do-cons (-do-let p t) bs)  
 ⇒ let p = t in -do-block bs  
 -do-block (-do-cons b (-do-cons c cs))  
 ⇒ -do-block (-do-cons b (-do-final (-do-block (-do-cons c cs))))  
 -do-cons (-do-let p t) (-do-final s)  
 ⇒ -do-final (let p = t in s)  
 -do-block (-do-final e) → e  
 (m ≫ n) → (m ≫= (λ-. n))

**adhoc-overloading**

bind ⇒ Set.bind Predicate.bind Option.bind List.bind

end

## 66 Less common functions on lists

**theory** More-List

**imports** Main

**begin**

**definition** strip-while :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list

**where**

strip-while P = rev ◦ dropWhile P ◦ rev

**lemma** strip-while-rev [simp]:

strip-while P (rev xs) = rev (dropWhile P xs)

⟨proof⟩

**lemma** strip-while-Nil [simp]:

strip-while P [] = []

⟨proof⟩

**lemma** strip-while-append [simp]:

¬ P x ⇒ strip-while P (xs @ [x]) = xs @ [x]

*<proof>*

**lemma** *strip-while-append-rec* [simp]:

$P\ x \implies \text{strip-while}\ P\ (xs\ @\ [x]) = \text{strip-while}\ P\ xs$

*<proof>*

**lemma** *strip-while-Cons* [simp]:

$\neg\ P\ x \implies \text{strip-while}\ P\ (x\ \#\ xs) = x\ \#\ \text{strip-while}\ P\ xs$

*<proof>*

**lemma** *strip-while-eq-Nil* [simp]:

$\text{strip-while}\ P\ xs = [] \iff (\forall x \in \text{set}\ xs.\ P\ x)$

*<proof>*

**lemma** *strip-while-eq-Cons-rec*:

$\text{strip-while}\ P\ (x\ \#\ xs) = x\ \#\ \text{strip-while}\ P\ xs \iff \neg\ (P\ x \wedge (\forall x \in \text{set}\ xs.\ P\ x))$

*<proof>*

**lemma** *split-strip-while-append*:

**fixes**  $xs :: 'a\ \text{list}$

**obtains**  $ys\ zs :: 'a\ \text{list}$

**where**  $\text{strip-while}\ P\ xs = ys$  **and**  $\forall x \in \text{set}\ zs.\ P\ x$  **and**  $xs = ys\ @\ zs$

*<proof>*

**lemma** *strip-while-snoc* [simp]:

$\text{strip-while}\ P\ (xs\ @\ [x]) = (\text{if}\ P\ x\ \text{then}\ \text{strip-while}\ P\ xs\ \text{else}\ xs\ @\ [x])$

*<proof>*

**lemma** *strip-while-map*:

$\text{strip-while}\ P\ (\text{map}\ f\ xs) = \text{map}\ f\ (\text{strip-while}\ (P\ \circ\ f)\ xs)$

*<proof>*

**lemma** *strip-while-dropWhile-commute*:

$\text{strip-while}\ P\ (\text{dropWhile}\ Q\ xs) = \text{dropWhile}\ Q\ (\text{strip-while}\ P\ xs)$

*<proof>*

**lemma** *dropWhile-strip-while-commute*:

$\text{dropWhile}\ P\ (\text{strip-while}\ Q\ xs) = \text{strip-while}\ Q\ (\text{dropWhile}\ P\ xs)$

*<proof>*

**definition** *no-leading* ::  $('a \Rightarrow \text{bool}) \Rightarrow 'a\ \text{list} \Rightarrow \text{bool}$

**where**

$\text{no-leading}\ P\ xs \iff (xs \neq [] \longrightarrow \neg\ P\ (\text{hd}\ xs))$

**lemma** *no-leading-Nil* [iff]:

$\text{no-leading}\ P\ []$

*<proof>*

**lemma** *no-leading-Cons* [iff]:

*no-leading*  $P$  ( $x \# xs$ )  $\longleftrightarrow \neg P x$

*<proof>*

**lemma** *no-leading-append* [simp]:

*no-leading*  $P$  ( $xs @ ys$ )  $\longleftrightarrow$  *no-leading*  $P$   $xs \wedge (xs = [] \longrightarrow$  *no-leading*  $P$   $ys)$

*<proof>*

**lemma** *no-leading-dropWhile* [simp]:

*no-leading*  $P$  (*dropWhile*  $P$   $xs$ )

*<proof>*

**lemma** *dropWhile-eq-obtain-leading*:

**assumes** *dropWhile*  $P$   $xs = ys$

**obtains**  $zs$  **where**  $xs = zs @ ys$  **and**  $\bigwedge z. z \in \text{set } zs \implies P z$  **and** *no-leading*  $P$

$ys$

*<proof>*

**lemma** *dropWhile-idem-iff*:

*dropWhile*  $P$   $xs = xs \longleftrightarrow$  *no-leading*  $P$   $xs$

*<proof>*

**abbreviation** *no-trailing* :: ( $'a \Rightarrow \text{bool}$ )  $\Rightarrow$   $'a$  list  $\Rightarrow \text{bool}$

**where**

*no-trailing*  $P$   $xs \equiv$  *no-leading*  $P$  (*rev*  $xs$ )

**lemma** *no-trailing-unfold*:

*no-trailing*  $P$   $xs \longleftrightarrow (xs \neq [] \longrightarrow \neg P (\text{last } xs))$

*<proof>*

**lemma** *no-trailing-Nil* [iff]:

*no-trailing*  $P$  []

*<proof>*

**lemma** *no-trailing-Cons* [simp]:

*no-trailing*  $P$  ( $x \# xs$ )  $\longleftrightarrow$  *no-trailing*  $P$   $xs \wedge (xs = [] \longrightarrow \neg P x)$

*<proof>*

**lemma** *no-trailing-append*:

*no-trailing*  $P$  ( $xs @ ys$ )  $\longleftrightarrow$  *no-trailing*  $P$   $ys \wedge (ys = [] \longrightarrow$  *no-trailing*  $P$   $xs)$

*<proof>*

**lemma** *no-trailing-append-Cons* [simp]:

*no-trailing*  $P$  ( $xs @ y \# ys$ )  $\longleftrightarrow$  *no-trailing*  $P$  ( $y \# ys$ )

*<proof>*

**lemma** *no-trailing-strip-while* [simp]:

*no-trailing*  $P$  (*strip-while*  $P$   $xs$ )

*<proof>*

**lemma** *strip-while-idem* [*simp*]:  
*no-trailing*  $P$   $xs \implies \text{strip-while } P \text{ } xs = xs$   
*<proof>*

**lemma** *strip-while-eq-obtain-trailing*:  
**assumes** *strip-while*  $P$   $xs = ys$   
**obtains**  $zs$  **where**  $xs = ys @ zs$  **and**  $\bigwedge z. z \in \text{set } zs \implies P z$  **and** *no-trailing*  $P$   $ys$   
*<proof>*

**lemma** *strip-while-idem-iff*:  
*strip-while*  $P$   $xs = xs \longleftrightarrow \text{no-trailing } P \text{ } xs$   
*<proof>*

**lemma** *no-trailing-map*:  
*no-trailing*  $P$  (*map*  $f$   $xs$ )  $\longleftrightarrow \text{no-trailing } (P \circ f) \text{ } xs$   
*<proof>*

**lemma** *no-trailing-drop* [*simp*]:  
*no-trailing*  $P$  (*drop*  $n$   $xs$ ) **if** *no-trailing*  $P$   $xs$   
*<proof>*

**lemma** *no-trailing-upt* [*simp*]:  
*no-trailing*  $P$  [ $n..<m$ ]  $\longleftrightarrow (n < m \longrightarrow \neg P (m - 1))$   
*<proof>*

**definition** *nth-default* ::  $'a \Rightarrow 'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a$   
**where**  
*nth-default*  $dflt$   $xs$   $n = (\text{if } n < \text{length } xs \text{ then } xs ! n \text{ else } dflt)$

**lemma** *nth-default-nth*:  
 $n < \text{length } xs \implies \text{nth-default } dflt \text{ } xs \text{ } n = xs ! n$   
*<proof>*

**lemma** *nth-default-beyond*:  
 $\text{length } xs \leq n \implies \text{nth-default } dflt \text{ } xs \text{ } n = dflt$   
*<proof>*

**lemma** *nth-default-Nil* [*simp*]:  
*nth-default*  $dflt$  []  $n = dflt$   
*<proof>*

**lemma** *nth-default-Cons*:  
*nth-default*  $dflt$  ( $x \# xs$ )  $n = (\text{case } n \text{ of } 0 \Rightarrow x \mid \text{Suc } n' \Rightarrow \text{nth-default } dflt \text{ } xs \text{ } n')$   
*<proof>*

**lemma** *nth-default-Cons-0* [*simp*]:

$nth\_default\ dflt\ (x\ \#\ xs)\ 0 = x$

*<proof>*

**lemma** *nth-default-Cons-Suc* [*simp*]:

$nth\_default\ dflt\ (x\ \#\ xs)\ (Suc\ n) = nth\_default\ dflt\ xs\ n$

*<proof>*

**lemma** *nth-default-replicate-dflt* [*simp*]:

$nth\_default\ dflt\ (replicate\ n\ dflt)\ m = dflt$

*<proof>*

**lemma** *nth-default-append*:

$nth\_default\ dflt\ (xs\ @\ ys)\ n =$

*(if*  $n < length\ xs$  *then*  $nth\ xs\ n$  *else*  $nth\_default\ dflt\ ys\ (n - length\ xs)$ *)*

*<proof>*

**lemma** *nth-default-append-trailing* [*simp*]:

$nth\_default\ dflt\ (xs\ @\ replicate\ n\ dflt) = nth\_default\ dflt\ xs$

*<proof>*

**lemma** *nth-default-snoc-default* [*simp*]:

$nth\_default\ dflt\ (xs\ @\ [dflt]) = nth\_default\ dflt\ xs$

*<proof>*

**lemma** *nth-default-eq-dflt-iff*:

$nth\_default\ dflt\ xs\ k = dflt \iff (k < length\ xs \implies xs\ !\ k = dflt)$

*<proof>*

**lemma** *nth-default-take-eq*:

$nth\_default\ dflt\ (take\ m\ xs)\ n =$

*(if*  $n < m$  *then*  $nth\_default\ dflt\ xs\ n$  *else*  $dflt$ *)*

*<proof>*

**lemma** *in-enumerate-iff-nth-default-eq*:

$x \neq dflt \implies (n, x) \in set\ (enumerate\ 0\ xs) \iff nth\_default\ dflt\ xs\ n = x$

*<proof>*

**lemma** *last-conv-nth-default*:

**assumes**  $xs \neq []$

**shows**  $last\ xs = nth\_default\ dflt\ xs\ (length\ xs - 1)$

*<proof>*

**lemma** *nth-default-map-eq*:

$f\ dflt' = dflt \implies nth\_default\ dflt\ (map\ f\ xs)\ n = f\ (nth\_default\ dflt'\ xs\ n)$

*<proof>*

**lemma** *finite-nth-default-neq-default* [*simp*]:

$finite\ \{k.\ nth\_default\ dflt\ xs\ k \neq dflt\}$

⟨proof⟩

**lemma** *sorted-list-of-set-nth-default*:

*sorted-list-of-set* { $k$ . *nth-default dflt xs k*  $\neq$  *dflt*} = *map fst* (*filter* ( $\lambda(-, x).$   $x \neq$  *dflt*) (*enumerate 0 xs*))

⟨proof⟩

**lemma** *map-nth-default*:

*map* (*nth-default x xs*) [ $0..<$ *length xs*] = *xs*

⟨proof⟩

**lemma** *range-nth-default* [*simp*]:

*range* (*nth-default dflt xs*) = *insert dflt* (*set xs*)

⟨proof⟩

**lemma** *nth-strip-while*:

**assumes**  $n <$  *length* (*strip-while P xs*)

**shows** *strip-while P xs* !  $n$  = *xs* !  $n$

⟨proof⟩

**lemma** *length-strip-while-le*:

*length* (*strip-while P xs*)  $\leq$  *length xs*

⟨proof⟩

**lemma** *nth-default-strip-while-dflt* [*simp*]:

*nth-default dflt* (*strip-while* ( $(=)$  *dflt*) *xs*) = *nth-default dflt xs*

⟨proof⟩

**lemma** *nth-default-eq-iff*:

*nth-default dflt xs* = *nth-default dflt ys*

$\longleftrightarrow$  *strip-while* (*HOL.eq dflt*) *xs* = *strip-while* (*HOL.eq dflt*) *ys* (**is**  $?P \longleftrightarrow$

$?Q$ )

⟨proof⟩

**lemma** *nth-default-map2*:

$\langle$ *nth-default d* (*map2 f xs ys*)  $n$  = *f* (*nth-default d1 xs n*) (*nth-default d2 ys n*) $\rangle$

**if**  $\langle$ *length xs* = *length ys* $\rangle$  **and**  $\langle$ *f d1 d2* = *d* $\rangle$  **for** *bs cs*

⟨proof⟩

**end**

**theory** *Cancellation*

**imports** *Main*

**begin**

**named-theorems** *cancelation-simproc-pre*  $\langle$ *These theorems are here to normalise the term. Special*

*handling of constructors should be here. Remark that only the simproc @{term*

*NO-MATCH* } is also  
included.}

**named-theorems** *cancellation-simproc-post* ‹These theorems are here to normalise the term, after the cancellation simproc. Normalisation of ‹iterate-add› back to the normale representation should be put here.›

**named-theorems** *cancellation-simproc-eq-elim* ‹These theorems are here to help deriving contradiction (e.g., ‹Suc - = 0›).›

**definition** *iterate-add* :: ‹nat  $\Rightarrow$  'a::cancel-comm-monoid-add  $\Rightarrow$  'a› **where**  
‹iterate-add n a = (((+) a)  $\widehat{\widehat{}}$  n) 0›

**lemma** *iterate-add-simps[simp]*:  
‹iterate-add 0 a = 0›  
‹iterate-add (Suc n) a = a + iterate-add n a›  
‹proof›

**lemma** *iterate-add-empty[simp]*: ‹iterate-add n 0 = 0›  
‹proof›

**lemma** *iterate-add-distrib[simp]*: ‹iterate-add (m+n) a = iterate-add m a + iterate-add n a›  
‹proof›

**lemma** *iterate-add-Numeral1*: ‹iterate-add n Numeral1 = of-nat n›  
‹proof›

**lemma** *iterate-add-1*: ‹iterate-add n 1 = of-nat n›  
‹proof›

**lemma** *iterate-add-eq-add-iff1*:  
‹i  $\leq$  j  $\implies$  (iterate-add j u + m = iterate-add i u + n) = (iterate-add (j - i) u + m = n)›  
‹proof›

**lemma** *iterate-add-eq-add-iff2*:  
‹i  $\leq$  j  $\implies$  (iterate-add i u + m = iterate-add j u + n) = (m = iterate-add (j - i) u + n)›  
‹proof›

**lemma** *iterate-add-less-iff1*:  
j  $\leq$  (i::nat)  $\implies$  (iterate-add i (u::'a :: {cancel-comm-monoid-add, ordered-ab-semigroup-add-imp-le}) + m < iterate-add j u + n) = (iterate-add (i-j) u + m < n)  
‹proof›

**lemma** *iterate-add-less-iff2*:

$i \leq (j::nat) \implies (iterate-add\ i\ (u::'a::\{\cancel{comm}\ monoid\ add,\ ordered\ ab\ semigroup\ add\ imp\ le\}))$   
 $+ m < iterate-add\ j\ u + n = (m < iterate-add\ (j - i)\ u + n)$   
 ⟨proof⟩

**lemma** *iterate-add-less-eq-iff1*:

$j \leq (i::nat) \implies (iterate-add\ i\ (u::'a::\{\cancel{comm}\ monoid\ add,\ ordered\ ab\ semigroup\ add\ imp\ le\}))$   
 $+ m \leq iterate-add\ j\ u + n = (iterate-add\ (i-j)\ u + m \leq n)$   
 ⟨proof⟩

**lemma** *iterate-add-less-eq-iff2*:

$i \leq (j::nat) \implies (iterate-add\ i\ (u::'a::\{\cancel{comm}\ monoid\ add,\ ordered\ ab\ semigroup\ add\ imp\ le\}))$   
 $+ m \leq iterate-add\ j\ u + n = (m \leq iterate-add\ (j - i)\ u + n)$   
 ⟨proof⟩

**lemma** *iterate-add-add-eq1*:

$j \leq (i::nat) \implies ((iterate-add\ i\ u + m) - (iterate-add\ j\ u + n)) = ((iterate-add\ (i-j)\ u + m) - n)$   
 ⟨proof⟩

**lemma** *iterate-add-diff-add-eq2*:

$i \leq (j::nat) \implies ((iterate-add\ i\ u + m) - (iterate-add\ j\ u + n)) = (m - (iterate-add\ (j-i)\ u + n))$   
 ⟨proof⟩

Simproc Set-Up

⟨ML⟩

end

## 67 (Finite) Multisets

**theory** *Multiset*

**imports** *Cancellation*

**begin**

### 67.1 The type of multisets

**typedef**  $'a\ multiset = \langle \{f :: 'a \Rightarrow nat.\ finite\ \{x.\ f\ x > 0\}\} \rangle$

**morphisms** *count Abs-multiset*

⟨proof⟩

**setup-lifting** *type-definition-multiset*

**lemma** *count-Abs-multiset*:

$\langle count\ (Abs-multiset\ f) = f \rangle$  **if**  $\langle finite\ \{x.\ f\ x > 0\} \rangle$

⟨proof⟩

**lemma** *multiset-eq-iff*:  $M = N \iff (\forall a.\ count\ M\ a = count\ N\ a)$



⟨proof⟩

**lemma** *multiset-eqI*:  $(\bigwedge x. \text{count } A \ x = \text{count } B \ x) \implies A = B$

⟨proof⟩

Preservation of the representing set *multiset*.

**lemma** *diff-preserves-multiset*:

⟨finite  $\{x. 0 < M \ x - N \ x\}$ ⟩ **if** ⟨finite  $\{x. 0 < M \ x\}$ ⟩ **for**  $M \ N :: \langle 'a \Rightarrow \text{nat} \rangle$

⟨proof⟩

**lemma** *filter-preserves-multiset*:

⟨finite  $\{x. 0 < (\text{if } P \ x \ \text{then } M \ x \ \text{else } 0)\}$ ⟩ **if** ⟨finite  $\{x. 0 < M \ x\}$ ⟩ **for**  $M \ N :: \langle 'a \Rightarrow \text{nat} \rangle$

⟨proof⟩

**lemmas** *in-multiset = diff-preserves-multiset filter-preserves-multiset*

## 67.2 Representing multisets

Multiset enumeration

**instantiation** *multiset* :: (type) *cancel-comm-monoid-add*

**begin**

**lift-definition** *zero-multiset* ::  $\langle 'a \ \text{multiset} \rangle$

**is**  $\langle \lambda a. 0 \rangle$

⟨proof⟩

**abbreviation** *empty-mset* ::  $\langle 'a \ \text{multiset} \rangle \ (\langle \{\#\} \rangle)$

**where**  $\langle \text{empty-mset} \equiv 0 \rangle$

**lift-definition** *plus-multiset* ::  $\langle 'a \ \text{multiset} \Rightarrow 'a \ \text{multiset} \Rightarrow 'a \ \text{multiset} \rangle$

**is**  $\langle \lambda M \ N \ a. M \ a + N \ a \rangle$

⟨proof⟩

**lift-definition** *minus-multiset* ::  $\langle 'a \ \text{multiset} \Rightarrow 'a \ \text{multiset} \Rightarrow 'a \ \text{multiset} \rangle$

**is**  $\langle \lambda M \ N \ a. M \ a - N \ a \rangle$

⟨proof⟩

**instance**

⟨proof⟩

**end**

**context**

**begin**

**qualified definition** *is-empty* ::  $\langle 'a \ \text{multiset} \Rightarrow \text{bool} \ \text{where}$

[code-abbrev]:  $\text{is-empty } A \longleftrightarrow A = \{\#\}$

**end**

**lemma** *add-mset-in-multiset*:

$\langle \text{finite } \{x. 0 < (\text{if } x = a \text{ then } \text{Suc } (M x) \text{ else } M x)\} \rangle$   
**if**  $\langle \text{finite } \{x. 0 < M x\} \rangle$   
 $\langle \text{proof} \rangle$

**lift-definition** *add-mset* ::  $'a \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset}$  **is**

$\lambda a M b. \text{if } b = a \text{ then } \text{Suc } (M b) \text{ else } M b$   
 $\langle \text{proof} \rangle$

**syntax**

*-multiset* ::  $\text{args} \Rightarrow 'a \text{ multiset}$  ( $\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix multiset enumeration} \rangle \{ \# \# \} \rangle \rangle$ )

**syntax-consts**

*-multiset*  $\equiv$  *add-mset*

**translations**

$\{ \# x, x \# \} == \text{CONST } \text{add-mset } x \{ \# x \# \}$   
 $\{ \# x \# \} == \text{CONST } \text{add-mset } x \{ \# \}$

**lemma** *count-empty* [*simp*]:  $\text{count } \{ \# \} a = 0$   
 $\langle \text{proof} \rangle$

**lemma** *count-add-mset* [*simp*]:

$\text{count } (\text{add-mset } b A) a = (\text{if } b = a \text{ then } \text{Suc } (\text{count } A a) \text{ else } \text{count } A a)$   
 $\langle \text{proof} \rangle$

**lemma** *count-single*:  $\text{count } \{ \# b \# \} a = (\text{if } b = a \text{ then } 1 \text{ else } 0)$   
 $\langle \text{proof} \rangle$

**lemma**

*add-mset-not-empty* [*simp*]:  $\langle \text{add-mset } a A \neq \{ \# \} \rangle$  **and**  
*empty-not-add-mset* [*simp*]:  $\{ \# \} \neq \text{add-mset } a A$   
 $\langle \text{proof} \rangle$

**lemma** *add-mset-add-mset-same-iff* [*simp*]:

$\text{add-mset } a A = \text{add-mset } a B \longleftrightarrow A = B$   
 $\langle \text{proof} \rangle$

**lemma** *add-mset-commute*:

$\text{add-mset } x (\text{add-mset } y M) = \text{add-mset } y (\text{add-mset } x M)$   
 $\langle \text{proof} \rangle$

## 67.3 Basic operations

### 67.3.1 Conversion to set and membership

**definition** *set-mset* ::  $\langle 'a \text{ multiset} \Rightarrow 'a \text{ set} \rangle$

**where**  $\langle \text{set-mset } M = \{x. \text{count } M x > 0\} \rangle$

**abbreviation** *member-mset* ::  $\langle 'a \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool} \rangle$   
**where**  $\langle \text{member-mset } a \ M \equiv a \in \text{set-mset } M \rangle$

**notation**

*member-mset* ( $\langle '( \in \# ' ) \rangle$ ) **and**  
*member-mset* ( $\langle (\langle \text{notation} = \langle \text{infix } \in \# \rangle \rangle - / \in \# -) \rangle$  [50, 51] 50)

**notation** (ASCII)

*member-mset* ( $\langle '( : \# ' ) \rangle$ ) **and**  
*member-mset* ( $\langle (\langle \text{notation} = \langle \text{infix } : \# \rangle \rangle - / : \# -) \rangle$  [50, 51] 50)

**abbreviation** *not-member-mset* ::  $\langle 'a \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool} \rangle$   
**where**  $\langle \text{not-member-mset } a \ M \equiv a \notin \text{set-mset } M \rangle$

**notation**

*not-member-mset* ( $\langle '( \notin \# ' ) \rangle$ ) **and**  
*not-member-mset* ( $\langle (\langle \text{notation} = \langle \text{infix } \notin \# \rangle \rangle - / \notin \# -) \rangle$  [50, 51] 50)

**notation** (ASCII)

*not-member-mset* ( $\langle '( \sim \# ' ) \rangle$ ) **and**  
*not-member-mset* ( $\langle (\langle \text{notation} = \langle \text{infix } \sim \# \rangle \rangle - / \sim \# -) \rangle$  [50, 51] 50)

**context****begin**

**qualified abbreviation** *Ball* ::  $\langle 'a \text{ multiset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool} \rangle$   
**where**  $\langle \text{Ball } M \equiv \text{Set.Ball } (\text{set-mset } M) \rangle$

**qualified abbreviation** *Bex* ::  $\langle 'a \text{ multiset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool} \rangle$   
**where**  $\langle \text{Bex } M \equiv \text{Set.Bex } (\text{set-mset } M) \rangle$

**end****syntax**

-*MBall* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$   
 $\langle (\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \forall \rangle \rangle \forall - \in \# - / -) \rangle$  [0, 0, 10] 10)  
 -*MBex* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$   
 $\langle (\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \exists \rangle \rangle \exists - \in \# - / -) \rangle$  [0, 0, 10] 10)

**syntax** (ASCII)

-*MBall* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$   
 $\langle (\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \forall \rangle \rangle \forall - : \# - / -) \rangle$  [0, 0, 10] 10)  
 -*MBex* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$   
 $\langle (\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \exists \rangle \rangle \exists - : \# - / -) \rangle$  [0, 0, 10] 10)

**syntax-consts**

-*MBall*  $\equiv$  *Multiset.Ball* **and**  
 -*MBex*  $\equiv$  *Multiset.Bex*

**translations**

$\forall x \in \# A. P \equiv \text{CONST } \text{Multiset.Ball } A (\lambda x. P)$   
 $\exists x \in \# A. P \equiv \text{CONST } \text{Multiset.Bex } A (\lambda x. P)$

⟨ML⟩

**lemma** *count-eq-zero-iff*:

$count\ M\ x = 0 \longleftrightarrow x \notin\# M$

⟨proof⟩

**lemma** *not-in-iff*:

$x \notin\# M \longleftrightarrow count\ M\ x = 0$

⟨proof⟩

**lemma** *count-greater-zero-iff* [simp]:

$count\ M\ x > 0 \longleftrightarrow x \in\# M$

⟨proof⟩

**lemma** *count-inI*:

**assumes**  $count\ M\ x = 0 \implies False$

**shows**  $x \in\# M$

⟨proof⟩

**lemma** *in-countE*:

**assumes**  $x \in\# M$

**obtains**  $n$  **where**  $count\ M\ x = Suc\ n$

⟨proof⟩

**lemma** *count-greater-eq-Suc-zero-iff* [simp]:

$count\ M\ x \geq Suc\ 0 \longleftrightarrow x \in\# M$

⟨proof⟩

**lemma** *count-greater-eq-one-iff* [simp]:

$count\ M\ x \geq 1 \longleftrightarrow x \in\# M$

⟨proof⟩

**lemma** *set-mset-empty* [simp]:

$set\ mset\ \{\#\} = \{\}$

⟨proof⟩

**lemma** *set-mset-single*:

$set\ mset\ \{\#b\# \} = \{b\}$

⟨proof⟩

**lemma** *set-mset-eq-empty-iff* [simp]:

$set\ mset\ M = \{\} \longleftrightarrow M = \{\#\}$

⟨proof⟩

**lemma** *finite-set-mset* [iff]:

$finite\ (set\ mset\ M)$

⟨proof⟩

**lemma** *set-mset-add-mset-insert* [simp]:  $\langle \text{set-mset } (\text{add-mset } a \ A) = \text{insert } a \ (\text{set-mset } A) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *multiset-nonemptyE* [elim]:  
 assumes  $A \neq \{\#\}$   
 obtains  $x$  where  $x \in\# A$   
 $\langle \text{proof} \rangle$

**lemma** *count-gt-imp-in-mset*:  $\text{count } M \ x > n \implies x \in\# M$   
 $\langle \text{proof} \rangle$

### 67.3.2 Union

**lemma** *count-union* [simp]:  
 $\text{count } (M + N) \ a = \text{count } M \ a + \text{count } N \ a$   
 $\langle \text{proof} \rangle$

**lemma** *set-mset-union* [simp]:  
 $\text{set-mset } (M + N) = \text{set-mset } M \cup \text{set-mset } N$   
 $\langle \text{proof} \rangle$

**lemma** *union-mset-add-mset-left* [simp]:  
 $\text{add-mset } a \ A + B = \text{add-mset } a \ (A + B)$   
 $\langle \text{proof} \rangle$

**lemma** *union-mset-add-mset-right* [simp]:  
 $A + \text{add-mset } a \ B = \text{add-mset } a \ (A + B)$   
 $\langle \text{proof} \rangle$

**lemma** *add-mset-add-single*:  $\langle \text{add-mset } a \ A = A + \{\#a\#\} \rangle$   
 $\langle \text{proof} \rangle$

### 67.3.3 Difference

**instance** *multiset* :: (type) *comm-monoid-diff*  
 $\langle \text{proof} \rangle$

**lemma** *count-diff* [simp]:  
 $\text{count } (M - N) \ a = \text{count } M \ a - \text{count } N \ a$   
 $\langle \text{proof} \rangle$

**lemma** *add-mset-diff-bothsides*:  
 $\langle \text{add-mset } a \ M - \text{add-mset } a \ A = M - A \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *in-diff-count*:  
 $a \in\# M - N \longleftrightarrow \text{count } N \ a < \text{count } M \ a$   
 $\langle \text{proof} \rangle$

**lemma** *count-in-diffI*:

**assumes**  $\bigwedge n. \text{count } N \ x = n + \text{count } M \ x \implies \text{False}$

**shows**  $x \in\# M - N$

*<proof>*

**lemma** *in-diff-countE*:

**assumes**  $x \in\# M - N$

**obtains**  $n$  **where**  $\text{count } M \ x = \text{Suc } n + \text{count } N \ x$

*<proof>*

**lemma** *in-diffD*:

**assumes**  $a \in\# M - N$

**shows**  $a \in\# M$

*<proof>*

**lemma** *set-mset-diff*:

*set-mset*  $(M - N) = \{a. \text{count } N \ a < \text{count } M \ a\}$

*<proof>*

**lemma** *diff-empty [simp]*:  $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$

*<proof>*

**lemma** *diff-cancel*:  $A - A = \{\#\}$

*<proof>*

**lemma** *diff-union-cancelR*:  $M + N - N = (M :: 'a \text{ multiset})$

*<proof>*

**lemma** *diff-union-cancelL*:  $N + M - N = (M :: 'a \text{ multiset})$

*<proof>*

**lemma** *diff-right-commute*:

**fixes**  $M \ N \ Q :: 'a \text{ multiset}$

**shows**  $M - N - Q = M - Q - N$

*<proof>*

**lemma** *diff-add*:

**fixes**  $M \ N \ Q :: 'a \text{ multiset}$

**shows**  $M - (N + Q) = M - N - Q$

*<proof>*

**lemma** *insert-DiffM [simp]*:  $x \in\# M \implies \text{add-mset } x \ (M - \{\#x\}) = M$

*<proof>*

**lemma** *insert-DiffM2*:  $x \in\# M \implies (M - \{\#x\}) + \{\#x\} = M$

*<proof>*

**lemma** *diff-union-swap*:  $a \neq b \implies \text{add-mset } b \ (M - \{\#a\}) = \text{add-mset } b \ M -$

$\{\#a\#\}$   
 $\langle proof \rangle$

**lemma** *diff-add-mset-swap* [*simp*]:  $b \notin\# A \implies add\text{-}mset\ b\ M - A = add\text{-}mset\ b\ (M - A)$   
 $\langle proof \rangle$

**lemma** *diff-union-swap2* [*simp*]:  $y \in\# M \implies add\text{-}mset\ x\ M - \{\#y\#\} = add\text{-}mset\ x\ (M - \{\#y\#\})$   
 $\langle proof \rangle$

**lemma** *diff-diff-add-mset* [*simp*]:  $(M::'a\ multiset) - N - P = M - (N + P)$   
 $\langle proof \rangle$

**lemma** *diff-union-single-conv*:  
 $a \in\# J \implies I + J - \{\#a\#\} = I + (J - \{\#a\#\})$   
 $\langle proof \rangle$

**lemma** *mset-add* [*elim?*]:  
**assumes**  $a \in\# A$   
**obtains**  $B$  **where**  $A = add\text{-}mset\ a\ B$   
 $\langle proof \rangle$

**lemma** *union-iff*:  
 $a \in\# A + B \iff a \in\# A \vee a \in\# B$   
 $\langle proof \rangle$

**lemma** *count-minus-inter-lt-count-minus-inter-iff*:  
 $count\ (M2 - M1)\ y < count\ (M1 - M2)\ y \iff y \in\# M1 - M2$   
 $\langle proof \rangle$

**lemma** *minus-inter-eq-minus-inter-iff*:  
 $(M1 - M2) = (M2 - M1) \iff set\text{-}mset\ (M1 - M2) = set\text{-}mset\ (M2 - M1)$   
 $\langle proof \rangle$

### 67.3.4 Min and Max

**abbreviation** *Min-mset* ::  $'a::linorder\ multiset \Rightarrow 'a$  **where**  
 $Min\text{-}mset\ m \equiv Min\ (set\text{-}mset\ m)$

**abbreviation** *Max-mset* ::  $'a::linorder\ multiset \Rightarrow 'a$  **where**  
 $Max\text{-}mset\ m \equiv Max\ (set\text{-}mset\ m)$

**lemma**  
 $Min\text{-}in\text{-}mset: M \neq \{\#\} \implies Min\text{-}mset\ M \in\# M$  **and**  
 $Max\text{-}in\text{-}mset: M \neq \{\#\} \implies Max\text{-}mset\ M \in\# M$   
 $\langle proof \rangle$

**67.3.5 Equality of multisets**

**lemma** *single-eq-single* [*simp*]:  $\{ \#a\# \} = \{ \#b\# \} \longleftrightarrow a = b$   
 ⟨*proof*⟩

**lemma** *union-eq-empty* [*iff*]:  $M + N = \{ \# \} \longleftrightarrow M = \{ \# \} \wedge N = \{ \# \}$   
 ⟨*proof*⟩

**lemma** *empty-eq-union* [*iff*]:  $\{ \# \} = M + N \longleftrightarrow M = \{ \# \} \wedge N = \{ \# \}$   
 ⟨*proof*⟩

**lemma** *multi-self-add-other-not-self* [*simp*]:  $M = \text{add-mset } x \ M \longleftrightarrow \text{False}$   
 ⟨*proof*⟩

**lemma** *add-mset-remove-trivial* [*simp*]:  $\langle \text{add-mset } x \ M - \{ \#x\# \} = M \rangle$   
 ⟨*proof*⟩

**lemma** *diff-single-trivial*:  $\neg x \in\# \ M \Longrightarrow M - \{ \#x\# \} = M$   
 ⟨*proof*⟩

**lemma** *diff-single-eq-union*:  $x \in\# \ M \Longrightarrow M - \{ \#x\# \} = N \longleftrightarrow M = \text{add-mset } x \ N$   
 ⟨*proof*⟩

**lemma** *union-single-eq-diff*:  $\text{add-mset } x \ M = N \Longrightarrow M = N - \{ \#x\# \}$   
 ⟨*proof*⟩

**lemma** *union-single-eq-member*:  $\text{add-mset } x \ M = N \Longrightarrow x \in\# \ N$   
 ⟨*proof*⟩

**lemma** *add-mset-remove-trivial-If*:  
 $\text{add-mset } a \ (N - \{ \#a\# \}) = (\text{if } a \in\# \ N \text{ then } N \text{ else } \text{add-mset } a \ N)$   
 ⟨*proof*⟩

**lemma** *add-mset-remove-trivial-eq*:  $\langle N = \text{add-mset } a \ (N - \{ \#a\# \}) \longleftrightarrow a \in\# \ N \rangle$   
 ⟨*proof*⟩

**lemma** *union-is-single*:  
 $M + N = \{ \#a\# \} \longleftrightarrow M = \{ \#a\# \} \wedge N = \{ \# \} \vee M = \{ \# \} \wedge N = \{ \#a\# \}$   
 (is ?lhs = ?rhs)  
 ⟨*proof*⟩

**lemma** *single-is-union*:  $\{ \#a\# \} = M + N \longleftrightarrow \{ \#a\# \} = M \wedge N = \{ \# \} \vee M = \{ \# \} \wedge \{ \#a\# \} = N$   
 ⟨*proof*⟩

**lemma** *add-eq-conv-diff*:  
 $\text{add-mset } a \ M = \text{add-mset } b \ N \longleftrightarrow M = N \wedge a = b \vee M = \text{add-mset } b \ (N - \{ \#a\# \}) \wedge N = \text{add-mset } a \ (M - \{ \#b\# \})$



(is ?lhs  $\longleftrightarrow$  ?rhs)

$\langle$ proof $\rangle$

**lemma** *add-mset-eq-single* [iff]:  $add-mset\ b\ M = \{\#a\# \} \longleftrightarrow b = a \wedge M = \{\#\}$   
 $\langle$ proof $\rangle$

**lemma** *single-eq-add-mset* [iff]:  $\{\#a\# \} = add-mset\ b\ M \longleftrightarrow b = a \wedge M = \{\#\}$   
 $\langle$ proof $\rangle$

**lemma** *insert-noteq-member*:

**assumes** *BC*:  $add-mset\ b\ B = add-mset\ c\ C$

**and** *bnotc*:  $b \neq c$

**shows**  $c \in\# B$

$\langle$ proof $\rangle$

**lemma** *add-eq-conv-ex*:

$(add-mset\ a\ M = add-mset\ b\ N) =$

$(M = N \wedge a = b \vee (\exists K. M = add-mset\ b\ K \wedge N = add-mset\ a\ K))$

$\langle$ proof $\rangle$

**lemma** *multi-member-split*:  $x \in\# M \implies \exists A. M = add-mset\ x\ A$

$\langle$ proof $\rangle$

**lemma** *multiset-add-sub-el-shuffle*:

**assumes**  $c \in\# B$

**and**  $b \neq c$

**shows**  $add-mset\ b\ (B - \{\#c\# \}) = add-mset\ b\ B - \{\#c\# \}$

$\langle$ proof $\rangle$

**lemma** *add-mset-eq-singleton-iff*[iff]:

$add-mset\ x\ M = \{\#y\# \} \longleftrightarrow M = \{\#\} \wedge x = y$

$\langle$ proof $\rangle$

### 67.3.6 Pointwise ordering induced by count

**definition** *subseteq-mset* :: 'a multiset  $\Rightarrow$  'a multiset  $\Rightarrow$  bool (**infix**  $\langle \subseteq\# \rangle$  50)

**where**  $A \subseteq\# B \longleftrightarrow (\forall a. count\ A\ a \leq count\ B\ a)$

**definition** *subset-mset* :: 'a multiset  $\Rightarrow$  'a multiset  $\Rightarrow$  bool (**infix**  $\langle \subset\# \rangle$  50)

**where**  $A \subset\# B \longleftrightarrow A \subseteq\# B \wedge A \neq B$

**abbreviation** (*input*) *supseteq-mset* :: 'a multiset  $\Rightarrow$  'a multiset  $\Rightarrow$  bool (**infix**  $\langle \supseteq\# \rangle$  50)

**where**  $supseteq-mset\ A\ B \equiv B \subseteq\# A$

**abbreviation** (*input*) *supset-mset* :: 'a multiset  $\Rightarrow$  'a multiset  $\Rightarrow$  bool (**infix**  $\langle \supset\# \rangle$  50)

**where**  $supset-mset\ A\ B \equiv B \subset\# A$

**notation** (*input*)

*subseteq-mset* (**infix**  $\langle \leq \# \rangle$  50) **and**  
*supseteq-mset* (**infix**  $\langle \geq \# \rangle$  50)

**notation** (*ASCII*)

*subseteq-mset* (**infix**  $\langle \leq \# \rangle$  50) **and**  
*subset-mset* (**infix**  $\langle < \# \rangle$  50) **and**  
*supseteq-mset* (**infix**  $\langle \geq \# \rangle$  50) **and**  
*supset-mset* (**infix**  $\langle > \# \rangle$  50)

**global-interpretation** *subset-mset*: ordering  $\langle (\subseteq \#) \rangle \langle (\subset \#) \rangle$   
 $\langle \text{proof} \rangle$

**interpretation** *subset-mset*: ordered-ab-semigroup-add-imp-le  $\langle (+) \rangle \langle (-) \rangle \langle (\subseteq \#) \rangle$   
 $\langle (\subset \#) \rangle$   
 $\langle \text{proof} \rangle$

**interpretation** *subset-mset*: ordered-ab-semigroup-monoid-add-imp-le  $(+) 0 (-)$   
 $(\subseteq \#) (\subset \#)$   
 $\langle \text{proof} \rangle$

**lemma** *mset-subset-eqI*:

$(\bigwedge a. \text{count } A \ a \leq \text{count } B \ a) \implies A \subseteq \# B$   
 $\langle \text{proof} \rangle$

**lemma** *mset-subset-eq-count*:

$A \subseteq \# B \implies \text{count } A \ a \leq \text{count } B \ a$   
 $\langle \text{proof} \rangle$

**lemma** *mset-subset-eq-exists-conv*:  $(A::'a \text{ multiset}) \subseteq \# B \longleftrightarrow (\exists C. B = A + C)$   
 $\langle \text{proof} \rangle$

**interpretation** *subset-mset*: ordered-cancel-comm-monoid-diff  $(+) 0 (\subseteq \#) (\subset \#)$   
 $(-)$   
 $\langle \text{proof} \rangle$

**declare** *subset-mset.add-diff-assoc[simp]* *subset-mset.add-diff-assoc2[simp]*

**lemma** *mset-subset-eq-mono-add-right-cancel*:  $(A::'a \text{ multiset}) + C \subseteq \# B + C$   
 $\longleftrightarrow A \subseteq \# B$   
 $\langle \text{proof} \rangle$

**lemma** *mset-subset-eq-mono-add-left-cancel*:  $C + (A::'a \text{ multiset}) \subseteq \# C + B \longleftrightarrow$   
 $A \subseteq \# B$   
 $\langle \text{proof} \rangle$

**lemma** *mset-subset-eq-mono-add*:  $(A::'a \text{ multiset}) \subseteq \# B \implies C \subseteq \# D \implies A +$   
 $C \subseteq \# B + D$

⟨proof⟩

**lemma** *mset-subset-eq-add-left*:  $(A::'a \text{ multiset}) \subseteq\# A + B$   
 ⟨proof⟩

**lemma** *mset-subset-eq-add-right*:  $B \subseteq\# (A::'a \text{ multiset}) + B$   
 ⟨proof⟩

**lemma** *single-subset-iff [simp]*:  
 $\{\#a\# \} \subseteq\# M \longleftrightarrow a \in\# M$   
 ⟨proof⟩

**lemma** *mset-subset-eq-single*:  $a \in\# B \implies \{\#a\# \} \subseteq\# B$   
 ⟨proof⟩

**lemma** *mset-subset-eq-add-mset-cancel*:  $\langle \text{add-mset } a \ A \subseteq\# \text{ add-mset } a \ B \longleftrightarrow A \subseteq\# B \rangle$   
 ⟨proof⟩

**lemma** *multiset-diff-union-assoc*:  
**fixes**  $A \ B \ C \ D :: 'a \text{ multiset}$   
**shows**  $C \subseteq\# B \implies A + B - C = A + (B - C)$   
 ⟨proof⟩

**lemma** *mset-subset-eq-multiset-union-diff-commute*:  
**fixes**  $A \ B \ C \ D :: 'a \text{ multiset}$   
**shows**  $B \subseteq\# A \implies A - B + C = A + C - B$   
 ⟨proof⟩

**lemma** *diff-subset-eq-self [simp]*:  
 $(M::'a \text{ multiset}) - N \subseteq\# M$   
 ⟨proof⟩

**lemma** *mset-subset-eqD*:  
**assumes**  $A \subseteq\# B$  **and**  $x \in\# A$   
**shows**  $x \in\# B$   
 ⟨proof⟩

**lemma** *mset-subsetD*:  
 $A \subset\# B \implies x \in\# A \implies x \in\# B$   
 ⟨proof⟩

**lemma** *set-mset-mono*:  
 $A \subseteq\# B \implies \text{set-mset } A \subseteq \text{set-mset } B$   
 ⟨proof⟩

**lemma** *mset-subset-eq-insertD*:  
**assumes**  $\text{add-mset } x \ A \subseteq\# B$   
**shows**  $x \in\# B \wedge A \subset\# B$

*<proof>*

**lemma** *mset-subset-insertD*:

$$\text{add-mset } x \ A \ C\# \ B \Longrightarrow x \in\# \ B \wedge A \ C\# \ B$$

*<proof>*

**lemma** *mset-subset-of-empty[simp]*:  $A \ C\# \ \{\#\} \longleftrightarrow \text{False}$

*<proof>*

**lemma** *empty-subset-add-mset[simp]*:  $\{\#\} \ C\# \ \text{add-mset } x \ M$

*<proof>*

**lemma** *empty-le*:  $\{\#\} \subseteq\# \ A$

*<proof>*

**lemma** *insert-subset-eq-iff*:

$$\text{add-mset } a \ A \subseteq\# \ B \longleftrightarrow a \in\# \ B \wedge A \subseteq\# \ B - \{\#a\}$$

*<proof>*

**lemma** *insert-union-subset-iff*:

$$\text{add-mset } a \ A \ C\# \ B \longleftrightarrow a \in\# \ B \wedge A \ C\# \ B - \{\#a\}$$

*<proof>*

**lemma** *subset-eq-diff-conv*:

$$A - C \subseteq\# \ B \longleftrightarrow A \subseteq\# \ B + C$$

*<proof>*

**lemma** *multi-psub-of-add-self [simp]*:  $A \ C\# \ \text{add-mset } x \ A$

*<proof>*

**lemma** *multi-psub-self*:  $A \ C\# \ A = \text{False}$

*<proof>*

**lemma** *mset-subset-add-mset [simp]*:  $\text{add-mset } x \ N \ C\# \ \text{add-mset } x \ M \longleftrightarrow N \ C\# \ M$

*<proof>*

**lemma** *mset-subset-diff-self*:  $c \in\# \ B \Longrightarrow B - \{\#c\} \ C\# \ B$

*<proof>*

**lemma** *Diff-eq-empty-iff-mset*:  $A - B = \{\#\} \longleftrightarrow A \subseteq\# \ B$

*<proof>*

**lemma** *add-mset-subseteq-single-iff[iff]*:  $\text{add-mset } a \ M \subseteq\# \ \{\#b\} \longleftrightarrow M = \{\#\}$

$$\wedge a = b$$

*<proof>*

**lemma** *nonempty-subseteq-mset-eq-single*:  $M \neq \{\#\} \Longrightarrow M \subseteq\# \ \{\#x\} \Longrightarrow M = \{\#x\}$

⟨proof⟩

**lemma** *nonempty-subseteq-mset-iff-single*:  $(M \neq \{\#\} \wedge M \subseteq\# \{\#x\# \} \wedge P) \longleftrightarrow$   
 $M = \{\#x\# \} \wedge P$   
 ⟨proof⟩

### 67.3.7 Intersection and bounded union

**definition** *inter-mset* ::  $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$  (**infixl**  $\langle \cap\# \rangle$   
 70)  
 where  $\langle A \cap\# B = A - (A - B) \rangle$

**lemma** *count-inter-mset* [*simp*]:  
 $\langle \text{count } (A \cap\# B) x = \min (\text{count } A x) (\text{count } B x) \rangle$   
 ⟨proof⟩

**interpretation** *subset-mset*: *semilattice-inf*  $\langle (\cap\#) \rangle \langle (\subseteq\#) \rangle \langle (\subset\#) \rangle$   
 ⟨proof⟩

**definition** *union-mset* ::  $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$  (**infixl**  $\langle \cup\# \rangle$   
 70)  
 where  $\langle A \cup\# B = A + (B - A) \rangle$

**lemma** *count-union-mset* [*simp*]:  
 $\langle \text{count } (A \cup\# B) x = \max (\text{count } A x) (\text{count } B x) \rangle$   
 ⟨proof⟩

**global-interpretation** *subset-mset*: *semilattice-neutr-order*  $\langle (\cup\#) \rangle \langle \{\#\} \rangle \langle (\supseteq\#) \rangle$   
 $\langle (\supset\#) \rangle$   
 ⟨proof⟩

**interpretation** *subset-mset*: *semilattice-sup*  $\langle (\cup\#) \rangle \langle (\subseteq\#) \rangle \langle (\subset\#) \rangle$   
 ⟨proof⟩

**interpretation** *subset-mset*: *bounded-lattice-bot*  $(\cap\#) (\subseteq\#) (\subset\#)$   
 $(\cup\#) \{\#\}$   
 ⟨proof⟩

### 67.3.8 Additional intersection facts

**lemma** *set-mset-inter* [*simp*]:  
 $\text{set-mset } (A \cap\# B) = \text{set-mset } A \cap \text{set-mset } B$   
 ⟨proof⟩

**lemma** *diff-intersect-left-idem* [*simp*]:  
 $M - M \cap\# N = M - N$   
 ⟨proof⟩

**lemma** *diff-intersect-right-idem* [simp]:

$$M - N \cap\# M = M - N$$

*<proof>*

**lemma** *multiset-inter-single*[simp]:  $a \neq b \implies \{\#a\# \} \cap\# \{\#b\# \} = \{\#\}$

*<proof>*

**lemma** *multiset-union-diff-commute*:

$$\text{assumes } B \cap\# C = \{\#\}$$

$$\text{shows } A + B - C = A - C + B$$

*<proof>*

**lemma** *disjunct-not-in*:

$$A \cap\# B = \{\#\} \longleftrightarrow (\forall a. a \notin\# A \vee a \notin\# B)$$

*<proof>*

**lemma** *inter-mset-empty-distrib-right*:  $A \cap\# (B + C) = \{\#\} \longleftrightarrow A \cap\# B = \{\#\} \wedge A \cap\# C = \{\#\}$

*<proof>*

**lemma** *inter-mset-empty-distrib-left*:  $(A + B) \cap\# C = \{\#\} \longleftrightarrow A \cap\# C = \{\#\} \wedge B \cap\# C = \{\#\}$

*<proof>*

**lemma** *add-mset-inter-add-mset* [simp]:

$$\text{add-mset } a \ A \cap\# \text{ add-mset } a \ B = \text{add-mset } a \ (A \cap\# B)$$

*<proof>*

**lemma** *add-mset-disjoint* [simp]:

$$\text{add-mset } a \ A \cap\# B = \{\#\} \longleftrightarrow a \notin\# B \wedge A \cap\# B = \{\#\}$$

$$\{\#\} = \text{add-mset } a \ A \cap\# B \longleftrightarrow a \notin\# B \wedge \{\#\} = A \cap\# B$$

*<proof>*

**lemma** *disjoint-add-mset* [simp]:

$$B \cap\# \text{ add-mset } a \ A = \{\#\} \longleftrightarrow a \notin\# B \wedge B \cap\# A = \{\#\}$$

$$\{\#\} = A \cap\# \text{ add-mset } b \ B \longleftrightarrow b \notin\# A \wedge \{\#\} = A \cap\# B$$

*<proof>*

**lemma** *inter-add-left1*:  $\neg x \in\# N \implies (\text{add-mset } x \ M) \cap\# N = M \cap\# N$

*<proof>*

**lemma** *inter-add-left2*:  $x \in\# N \implies (\text{add-mset } x \ M) \cap\# N = \text{add-mset } x \ (M \cap\# (N - \{\#x\#}))$

*<proof>*

**lemma** *inter-add-right1*:  $\neg x \in\# N \implies N \cap\# (\text{add-mset } x \ M) = N \cap\# M$

*<proof>*

**lemma** *inter-add-right2*:  $x \in\# N \implies N \cap\# (\text{add-mset } x \ M) = \text{add-mset } x \ ((N$

–  $\{\#x\#\} \cap\# M$   
 $\langle proof \rangle$

**lemma** *disjunct-set-mset-diff*:  
**assumes**  $M \cap\# N = \{\#\}$   
**shows**  $set\text{-}mset (M - N) = set\text{-}mset M$   
 $\langle proof \rangle$

**lemma** *at-most-one-mset-mset-diff*:  
**assumes**  $a \notin\# M - \{\#a\#\}$   
**shows**  $set\text{-}mset (M - \{\#a\#\}) = set\text{-}mset M - \{a\}$   
 $\langle proof \rangle$

**lemma** *more-than-one-mset-mset-diff*:  
**assumes**  $a \in\# M - \{\#a\#\}$   
**shows**  $set\text{-}mset (M - \{\#a\#\}) = set\text{-}mset M$   
 $\langle proof \rangle$

**lemma** *inter-iff*:  
 $a \in\# A \cap\# B \longleftrightarrow a \in\# A \wedge a \in\# B$   
 $\langle proof \rangle$

**lemma** *inter-union-distrib-left*:  
 $A \cap\# B + C = (A + C) \cap\# (B + C)$   
 $\langle proof \rangle$

**lemma** *inter-union-distrib-right*:  
 $C + A \cap\# B = (C + A) \cap\# (C + B)$   
 $\langle proof \rangle$

**lemma** *inter-subset-eq-union*:  
 $A \cap\# B \subseteq\# A + B$   
 $\langle proof \rangle$

### 67.3.9 Additional bounded union facts

**lemma** *set-mset-sup [simp]*:  
 $set\text{-}mset (A \cup\# B) = set\text{-}mset A \cup set\text{-}mset B$   
 $\langle proof \rangle$

**lemma** *sup-union-left1 [simp]*:  $\neg x \in\# N \implies (add\text{-}mset x M) \cup\# N = add\text{-}mset x (M \cup\# N)$   
 $\langle proof \rangle$

**lemma** *sup-union-left2*:  $x \in\# N \implies (add\text{-}mset x M) \cup\# N = add\text{-}mset x (M \cup\# (N - \{\#x\#\}))$   
 $\langle proof \rangle$

**lemma** *sup-union-right1 [simp]*:  $\neg x \in\# N \implies N \cup\# (add\text{-}mset x M) = add\text{-}mset$

$x (N \cup\# M)$   
 ⟨proof⟩

**lemma** *sup-union-right2*:  $x \in\# N \implies N \cup\# (\text{add-mset } x M) = \text{add-mset } x ((N - \{\#x\}) \cup\# M)$   
 ⟨proof⟩

**lemma** *sup-union-distrib-left*:  
 $A \cup\# B + C = (A + C) \cup\# (B + C)$   
 ⟨proof⟩

**lemma** *union-sup-distrib-right*:  
 $C + A \cup\# B = (C + A) \cup\# (C + B)$   
 ⟨proof⟩

**lemma** *union-diff-inter-eq-sup*:  
 $A + B - A \cap\# B = A \cup\# B$   
 ⟨proof⟩

**lemma** *union-diff-sup-eq-inter*:  
 $A + B - A \cup\# B = A \cap\# B$   
 ⟨proof⟩

**lemma** *add-mset-union*:  
 ⟨ $\text{add-mset } a A \cup\# \text{add-mset } a B = \text{add-mset } a (A \cup\# B)$ ⟩  
 ⟨proof⟩

## 67.4 Replicate and repeat operations

**definition** *replicate-mset* ::  $\text{nat} \Rightarrow 'a \Rightarrow 'a \text{ multiset}$  **where**  
 $\text{replicate-mset } n x = (\text{add-mset } x \overset{\sim}{\sim} n) \{\#\}$

**lemma** *replicate-mset-0*[*simp*]:  $\text{replicate-mset } 0 x = \{\#\}$   
 ⟨proof⟩

**lemma** *replicate-mset-Suc* [*simp*]:  $\text{replicate-mset } (\text{Suc } n) x = \text{add-mset } x (\text{replicate-mset } n x)$   
 ⟨proof⟩

**lemma** *count-replicate-mset*[*simp*]:  $\text{count } (\text{replicate-mset } n x) y = (\text{if } y = x \text{ then } n \text{ else } 0)$   
 ⟨proof⟩

**lift-definition** *repeat-mset* ::  $\langle \text{nat} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$   
**is**  $\langle \lambda n M a. n * M a \rangle$  ⟨proof⟩

**lemma** *count-repeat-mset* [*simp*]:  $\text{count } (\text{repeat-mset } i A) a = i * \text{count } A a$   
 ⟨proof⟩



**lemma** *repeat-mset-0* [simp]:

$\langle \text{repeat-mset } 0 \ M = \{\#\} \rangle$

$\langle \text{proof} \rangle$

**lemma** *repeat-mset-Suc* [simp]:

$\langle \text{repeat-mset } (\text{Suc } n) \ M = M + \text{repeat-mset } n \ M \rangle$

$\langle \text{proof} \rangle$

**lemma** *repeat-mset-right* [simp]:  $\text{repeat-mset } a \ (\text{repeat-mset } b \ A) = \text{repeat-mset } (a * b) \ A$

$\langle \text{proof} \rangle$

**lemma** *left-diff-repeat-mset-distrib'*:  $\langle \text{repeat-mset } (i - j) \ u = \text{repeat-mset } i \ u - \text{repeat-mset } j \ u \rangle$

$\langle \text{proof} \rangle$

**lemma** *left-add-mult-distrib-mset*:

$\text{repeat-mset } i \ u + (\text{repeat-mset } j \ u + k) = \text{repeat-mset } (i+j) \ u + k$

$\langle \text{proof} \rangle$

**lemma** *repeat-mset-distrib*:

$\text{repeat-mset } (m + n) \ A = \text{repeat-mset } m \ A + \text{repeat-mset } n \ A$

$\langle \text{proof} \rangle$

**lemma** *repeat-mset-distrib2*[simp]:

$\text{repeat-mset } n \ (A + B) = \text{repeat-mset } n \ A + \text{repeat-mset } n \ B$

$\langle \text{proof} \rangle$

**lemma** *repeat-mset-replicate-mset*[simp]:

$\text{repeat-mset } n \ \{\#a\# \} = \text{replicate-mset } n \ a$

$\langle \text{proof} \rangle$

**lemma** *repeat-mset-distrib-add-mset*[simp]:

$\text{repeat-mset } n \ (\text{add-mset } a \ A) = \text{replicate-mset } n \ a + \text{repeat-mset } n \ A$

$\langle \text{proof} \rangle$

**lemma** *repeat-mset-empty*[simp]:  $\text{repeat-mset } n \ \{\#\} = \{\#\}$

$\langle \text{proof} \rangle$

**lemma** *set-mset-sum*:  $\text{finite } A \implies \text{set-mset } (\sum x \in A. f \ x) = (\bigcup x \in A. \text{set-mset } (f \ x))$

$\langle \text{proof} \rangle$

### 67.4.1 Simprocs

**lemma** *repeat-mset-iterate-add*:  $\langle \text{repeat-mset } n \ M = \text{iterate-add } n \ M \rangle$

$\langle \text{proof} \rangle$

**lemma** *mset-subseteq-add-iff1*:

$j \leq (i::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subseteq\# \text{ repeat-mset } j \ u + n) = (\text{repeat-mset } (i-j) \ u + m \subseteq\# n)$   
 ⟨proof⟩

**lemma** *mset-subseteq-add-iff2*:

$i \leq (j::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subseteq\# \text{ repeat-mset } j \ u + n) = (m \subseteq\# \text{ repeat-mset } (j-i) \ u + n)$   
 ⟨proof⟩

**lemma** *mset-subset-add-iff1*:

$j \leq (i::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subset\# \text{ repeat-mset } j \ u + n) = (\text{repeat-mset } (i-j) \ u + m \subset\# n)$   
 ⟨proof⟩

**lemma** *mset-subset-add-iff2*:

$i \leq (j::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subset\# \text{ repeat-mset } j \ u + n) = (m \subset\# \text{ repeat-mset } (j-i) \ u + n)$   
 ⟨proof⟩

⟨ML⟩

**lemma** *add-mset-replicate-mset-safe[cancelation-simproc-pre]*: ⟨NO-MATCH {#}  
 $M \implies \text{add-mset } a \ M = \{\#a\# \} + M$ ⟩  
 ⟨proof⟩

**declare** *repeat-mset-iterate-add[cancelation-simproc-pre]*

**declare** *iterate-add-distrib[cancelation-simproc-pre]*

**declare** *repeat-mset-iterate-add[symmetric, cancelation-simproc-post]*

**declare** *add-mset-not-empty[cancelation-simproc-eq-elim]*

*empty-not-add-mset[cancelation-simproc-eq-elim]*

*subset-mset.le-zero-eq[cancelation-simproc-eq-elim]*

*empty-not-add-mset[cancelation-simproc-eq-elim]*

*add-mset-not-empty[cancelation-simproc-eq-elim]*

*subset-mset.le-zero-eq[cancelation-simproc-eq-elim]*

*le-zero-eq[cancelation-simproc-eq-elim]*

⟨ML⟩

## 67.4.2 Conditionally complete lattice

**instantiation** *multiset* :: (type) Inf

**begin**

**lift-definition** *Inf-multiset* :: 'a multiset set  $\Rightarrow$  'a multiset is

$\lambda A \ i. \text{if } A = \{\} \text{ then } 0 \text{ else } \text{Inf } ((\lambda f. f \ i) \ 'A)$

⟨proof⟩

**instance**  $\langle proof \rangle$

**end**

**lemma** *Inf-multiset-empty*:  $Inf \ \{\} = \{\#\}$   
 $\langle proof \rangle$

**lemma** *count-Inf-multiset-nonempty*:  $A \neq \{\} \implies count \ (Inf \ A) \ x = Inf \ ((\lambda X. count \ X \ x) \ 'A)$   
 $\langle proof \rangle$

**instantiation** *multiset* :: (type) Sup  
**begin**

**definition** *Sup-multiset* :: 'a multiset set  $\Rightarrow$  'a multiset **where**  
*Sup-multiset*  $A = (if \ A \neq \{\} \wedge subset\text{-mset}.\text{bdd-above} \ A \ then$   
*Abs-multiset*  $(\lambda i. Sup \ ((\lambda X. count \ X \ i) \ 'A)) \ else \ \{\#\})$

**lemma** *Sup-multiset-empty*:  $Sup \ \{\} = \{\#\}$   
 $\langle proof \rangle$

**lemma** *Sup-multiset-unbounded*:  $\neg subset\text{-mset}.\text{bdd-above} \ A \implies Sup \ A = \{\#\}$   
 $\langle proof \rangle$

**instance**  $\langle proof \rangle$

**end**

**lemma** *bdd-above-multiset-imp-bdd-above-count*:  
**assumes**  $subset\text{-mset}.\text{bdd-above} \ (A :: 'a \ multiset \ set)$   
**shows**  $bdd\text{-above} \ ((\lambda X. count \ X \ x) \ 'A)$   
 $\langle proof \rangle$

**lemma** *bdd-above-multiset-imp-finite-support*:  
**assumes**  $A \neq \{\}$   $subset\text{-mset}.\text{bdd-above} \ (A :: 'a \ multiset \ set)$   
**shows**  $finite \ (\bigcup X \in A. \ \{x. \ count \ X \ x > 0\})$   
 $\langle proof \rangle$

**lemma** *Sup-multiset-in-multiset*:  
 $\langle finite \ \{i. \ 0 < (SUP \ M \in A. \ count \ M \ i)\} \rangle$   
**if**  $\langle A \neq \{\} \rangle \langle subset\text{-mset}.\text{bdd-above} \ A \rangle$   
 $\langle proof \rangle$

**lemma** *count-Sup-multiset-nonempty*:  
 $\langle count \ (Sup \ A) \ x = (SUP \ X \in A. \ count \ X \ x) \rangle$   
**if**  $\langle A \neq \{\} \rangle \langle subset\text{-mset}.\text{bdd-above} \ A \rangle$   
 $\langle proof \rangle$

**interpretation** *subset-mset: conditionally-complete-lattice Inf Sup* ( $\cap\#$ ) ( $\subseteq\#$ ) ( $\subset\#$ ) ( $\cup\#$ )  
 $\langle proof \rangle$

**lemma** *set-mset-Inf*:  
**assumes**  $A \neq \{\}$   
**shows**  $set\text{-}mset\ (Inf\ A) = (\bigcap X \in A. set\text{-}mset\ X)$   
 $\langle proof \rangle$

**lemma** *in-Inf-multiset-iff*:  
**assumes**  $A \neq \{\}$   
**shows**  $x \in\# Inf\ A \longleftrightarrow (\forall X \in A. x \in\# X)$   
 $\langle proof \rangle$

**lemma** *in-Inf-multisetD*:  $x \in\# Inf\ A \implies X \in A \implies x \in\# X$   
 $\langle proof \rangle$

**lemma** *set-mset-Sup*:  
**assumes** *subset-mset.bdd-above A*  
**shows**  $set\text{-}mset\ (Sup\ A) = (\bigcup X \in A. set\text{-}mset\ X)$   
 $\langle proof \rangle$

**lemma** *in-Sup-multiset-iff*:  
**assumes** *subset-mset.bdd-above A*  
**shows**  $x \in\# Sup\ A \longleftrightarrow (\exists X \in A. x \in\# X)$   
 $\langle proof \rangle$

**lemma** *in-Sup-multisetD*:  
**assumes**  $x \in\# Sup\ A$   
**shows**  $\exists X \in A. x \in\# X$   
 $\langle proof \rangle$

**interpretation** *subset-mset: distrib-lattice* ( $\cap\#$ ) ( $\subseteq\#$ ) ( $\subset\#$ ) ( $\cup\#$ )  
 $\langle proof \rangle$

### 67.4.3 Filter (with comprehension)

Multiset comprehension

**lift-definition** *filter-mset* ::  $('a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$   
**is**  $\lambda P M. \lambda x. if\ P\ x\ then\ M\ x\ else\ 0$   
 $\langle proof \rangle$

**syntax** (*ASCII*)  
 $-MCollect :: pptrn \Rightarrow 'a\ multiset \Rightarrow bool \Rightarrow 'a\ multiset$   
 $(\langle \langle indent=1\ notation=\langle mixfix\ multiset\ comprehension \rangle \rangle \{ \# - : \# - / - \# \} \rangle)$

**syntax**  
 $-MCollect :: pptrn \Rightarrow 'a\ multiset \Rightarrow bool \Rightarrow 'a\ multiset$   
 $(\langle \langle indent=1\ notation=\langle mixfix\ multiset\ comprehension \rangle \rangle \{ \# - \in \# - / - \# \} \rangle)$

**syntax-consts**

$-MCollect == filter\text{-}mset$

**translations**

$\{\#x \in\# M. P\# \} == CONST\ filter\text{-}mset (\lambda x. P) M$

**lemma** *count-filter-mset* [simp]:

$count (filter\text{-}mset P M) a = (if P a then count M a else 0)$   
 ⟨proof⟩

**lemma** *set-mset-filter* [simp]:

$set\text{-}mset (filter\text{-}mset P M) = \{a \in set\text{-}mset M. P a\}$   
 ⟨proof⟩

**lemma** *filter-empty-mset* [simp]:  $filter\text{-}mset P \{\#\} = \{\#\}$

⟨proof⟩

**lemma** *filter-single-mset*:  $filter\text{-}mset P \{\#x\# \} = (if P x then \{\#x\# \} else \{\#\})$

⟨proof⟩

**lemma** *filter-union-mset* [simp]:  $filter\text{-}mset P (M + N) = filter\text{-}mset P M + filter\text{-}mset P N$

⟨proof⟩

**lemma** *filter-diff-mset* [simp]:  $filter\text{-}mset P (M - N) = filter\text{-}mset P M - filter\text{-}mset P N$

⟨proof⟩

**lemma** *filter-inter-mset* [simp]:  $filter\text{-}mset P (M \cap\# N) = filter\text{-}mset P M \cap\# filter\text{-}mset P N$

⟨proof⟩

**lemma** *filter-sup-mset*[simp]:  $filter\text{-}mset P (A \cup\# B) = filter\text{-}mset P A \cup\# filter\text{-}mset P B$

⟨proof⟩

**lemma** *filter-mset-add-mset* [simp]:

$filter\text{-}mset P (add\text{-}mset x A) =$   
 (if  $P x$  then  $add\text{-}mset x (filter\text{-}mset P A)$  else  $filter\text{-}mset P A$ )  
 ⟨proof⟩

**lemma** *multiset-filter-subset*[simp]:  $filter\text{-}mset f M \subseteq\# M$

⟨proof⟩

**lemma** *multiset-filter-mono*:

**assumes**  $A \subseteq\# B$

**shows**  $filter\text{-}mset f A \subseteq\# filter\text{-}mset f B$

⟨proof⟩

**lemma** *filter-mset-eq-conv*:

$filter\text{-}mset P M = N \longleftrightarrow N \subseteq\# M \wedge (\forall b \in\# N. P b) \wedge (\forall a \in\# M - N. \neg P a)$

(is  $?P \longleftrightarrow ?Q$ )  
 ⟨proof⟩

**lemma** *filter-filter-mset*:  $\text{filter-mset } P (\text{filter-mset } Q M) = \{\#x \in\# M. Q x \wedge P x\#$   
 $\}$   
 ⟨proof⟩

**lemma**  
*filter-mset-True[simp]*:  $\{\#y \in\# M. \text{True}\#\} = M$  **and**  
*filter-mset-False[simp]*:  $\{\#y \in\# M. \text{False}\#\} = \{\#\}$   
 ⟨proof⟩

**lemma** *filter-mset-cong0*:  
**assumes**  $\bigwedge x. x \in\# M \implies f x \longleftrightarrow g x$   
**shows**  $\text{filter-mset } f M = \text{filter-mset } g M$   
 ⟨proof⟩

**lemma** *filter-mset-cong*:  
**assumes**  $M = M'$  **and**  $\bigwedge x. x \in\# M' \implies f x \longleftrightarrow g x$   
**shows**  $\text{filter-mset } f M = \text{filter-mset } g M'$   
 ⟨proof⟩

**lemma** *filter-eq-replicate-mset*:  $\{\#y \in\# D. y = x\#\} = \text{replicate-mset } (\text{count } D x)$   
 $x$   
 ⟨proof⟩

#### 67.4.4 Size

**definition** *wcount* **where**  $wcount f M = (\lambda x. \text{count } M x * \text{Suc } (f x))$

**lemma** *wcount-union*:  $wcount f (M + N) a = wcount f M a + wcount f N a$   
 ⟨proof⟩

**lemma** *wcount-add-mset*:  
 $wcount f (\text{add-mset } x M) a = (\text{if } x = a \text{ then } \text{Suc } (f a) \text{ else } 0) + wcount f M a$   
 ⟨proof⟩

**definition** *size-multiset* ::  $('a \Rightarrow \text{nat}) \Rightarrow 'a \text{ multiset} \Rightarrow \text{nat}$  **where**  
 $\text{size-multiset } f M = \text{sum } (wcount f M) (\text{set-mset } M)$

**lemmas** *size-multiset-eq* = *size-multiset-def*[*unfolded wcount-def*]

**instantiation** *multiset* ::  $(\text{type}) \text{ size}$   
**begin**

**definition** *size-multiset* **where**  
 $\text{size-multiset-overloaded-def}: \text{size-multiset} = \text{Multiset.size-multiset } (\lambda-. 0)$   
**instance** ⟨proof⟩

**end**

**lemmas** *size-multiset-overloaded-eq* =  
*size-multiset-overloaded-def*[*THEN fun-cong, unfolded size-multiset-eq, simplified*]

**lemma** *size-multiset-empty* [*simp*]: *size-multiset*  $f \{\#\} = 0$   
 ⟨*proof*⟩

**lemma** *size-empty* [*simp*]: *size*  $\{\#\} = 0$   
 ⟨*proof*⟩

**lemma** *size-multiset-single* : *size-multiset*  $f \{\#b\# \} = \text{Suc } (f b)$   
 ⟨*proof*⟩

**lemma** *size-single*: *size*  $\{\#b\# \} = 1$   
 ⟨*proof*⟩

**lemma** *sum-wcount-Int*:  
*finite*  $A \implies \text{sum } (\text{wcount } f N) (A \cap \text{set-mset } N) = \text{sum } (\text{wcount } f N) A$   
 ⟨*proof*⟩

**lemma** *size-multiset-union* [*simp*]:  
*size-multiset*  $f (M + N::'a \text{ multiset}) = \text{size-multiset } f M + \text{size-multiset } f N$   
 ⟨*proof*⟩

**lemma** *size-multiset-add-mset* [*simp*]:  
*size-multiset*  $f (\text{add-mset } a M) = \text{Suc } (f a) + \text{size-multiset } f M$   
 ⟨*proof*⟩

**lemma** *size-add-mset* [*simp*]: *size*  $(\text{add-mset } a A) = \text{Suc } (\text{size } A)$   
 ⟨*proof*⟩

**lemma** *size-union* [*simp*]: *size*  $(M + N::'a \text{ multiset}) = \text{size } M + \text{size } N$   
 ⟨*proof*⟩

**lemma** *size-multiset-eq-0-iff-empty* [*iff*]:  
*size-multiset*  $f M = 0 \iff M = \{\#\}$   
 ⟨*proof*⟩

**lemma** *size-eq-0-iff-empty* [*iff*]:  $(\text{size } M = 0) = (M = \{\#\})$   
 ⟨*proof*⟩

**lemma** *nonempty-has-size*:  $(S \neq \{\#\}) = (0 < \text{size } S)$   
 ⟨*proof*⟩

**lemma** *size-eq-Suc-imp-elem*: *size*  $M = \text{Suc } n \implies \exists a. a \in\# M$   
 ⟨*proof*⟩

**lemma** *size-eq-Suc-imp-eq-union*:

**assumes**  $size\ M = Suc\ n$   
**shows**  $\exists a\ N. M = add\text{-}mset\ a\ N$   
 $\langle proof \rangle$

**lemma** *size-mset-mono*:  
**fixes**  $A\ B :: 'a\ multiset$   
**assumes**  $A \subseteq\# B$   
**shows**  $size\ A \leq size\ B$   
 $\langle proof \rangle$

**lemma** *size-filter-mset-lesseq[simp]*:  $size\ (filter\text{-}mset\ f\ M) \leq size\ M$   
 $\langle proof \rangle$

**lemma** *size-Diff-submset*:  
 $M \subseteq\# M' \implies size\ (M' - M) = size\ M' - size\ (M :: 'a\ multiset)$   
 $\langle proof \rangle$

**lemma** *size-lt-imp-ex-count-lt*:  $size\ M < size\ N \implies \exists x \in\# N. count\ M\ x < count\ N\ x$   
 $\langle proof \rangle$

## 67.5 Induction and case splits

**theorem** *multiset-induct* [*case-names empty add, induct type: multiset*]:  
**assumes** *empty*:  $P\ \{\#\}$   
**assumes** *add*:  $\bigwedge x\ M. P\ M \implies P\ (add\text{-}mset\ x\ M)$   
**shows**  $P\ M$   
 $\langle proof \rangle$

**lemma** *multiset-induct-min*[*case-names empty add*]:  
**fixes**  $M :: 'a::linorder\ multiset$   
**assumes**  
*empty*:  $P\ \{\#\}$  **and**  
*add*:  $\bigwedge x\ M. P\ M \implies (\forall y \in\# M. y \geq x) \implies P\ (add\text{-}mset\ x\ M)$   
**shows**  $P\ M$   
 $\langle proof \rangle$

**lemma** *multiset-induct-max*[*case-names empty add*]:  
**fixes**  $M :: 'a::linorder\ multiset$   
**assumes**  
*empty*:  $P\ \{\#\}$  **and**  
*add*:  $\bigwedge x\ M. P\ M \implies (\forall y \in\# M. y \leq x) \implies P\ (add\text{-}mset\ x\ M)$   
**shows**  $P\ M$   
 $\langle proof \rangle$

**lemma** *multi-nonempty-split*:  $M \neq \{\#\} \implies \exists A\ a. M = add\text{-}mset\ a\ A$   
 $\langle proof \rangle$

**lemma** *multiset-cases* [*cases type*]:



**obtains** (*empty*)  $M = \{\#\} \mid (\text{add}) x N$  **where**  $M = \text{add-mset } x N$   
 ⟨*proof*⟩

**lemma** *multi-drop-mem-not-eq*:  $c \in\# B \implies B - \{\#c\} \neq B$   
 ⟨*proof*⟩

**lemma** *union-filter-mset-complement*[*simp*]:  
 $\forall x. P x = (\neg Q x) \implies \text{filter-mset } P M + \text{filter-mset } Q M = M$   
 ⟨*proof*⟩

**lemma** *multiset-partition*:  $M = \{\#x \in\# M. P x\} + \{\#x \in\# M. \neg P x\}$   
 ⟨*proof*⟩

**lemma** *mset-subset-size*:  $A \subset\# B \implies \text{size } A < \text{size } B$   
 ⟨*proof*⟩

**lemma** *size-1-singleton-mset*:  $\text{size } M = 1 \implies \exists a. M = \{\#a\}$   
 ⟨*proof*⟩

**lemma** *set-mset-subset-singletonD*:  
**assumes** *set-mset*  $A \subseteq \{x\}$   
**shows**  $A = \text{replicate-mset } (\text{size } A) x$   
 ⟨*proof*⟩

**lemma** *count-conv-size-mset*:  $\text{count } A x = \text{size } (\text{filter-mset } (\lambda y. y = x) A)$   
 ⟨*proof*⟩

**lemma** *size-conv-count-bool-mset*:  $\text{size } A = \text{count } A \text{ True} + \text{count } A \text{ False}$   
 ⟨*proof*⟩

### 67.5.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

**lemma** *wf-subset-mset-rel*:  $wf \{(M, N :: 'a \text{ multiset}). M \subset\# N\}$   
 ⟨*proof*⟩

**lemma** *wfp-subset-mset*[*simp*]:  $wfp (\subset\#)$   
 ⟨*proof*⟩

**lemma** *full-multiset-induct* [*case-names less*]:  
**assumes** *ih*:  $\bigwedge B. \forall (A :: 'a \text{ multiset}). A \subset\# B \longrightarrow P A \implies P B$   
**shows**  $P B$   
 ⟨*proof*⟩

**lemma** *multi-subset-induct* [*consumes 2, case-names empty add*]:  
**assumes**  $F \subseteq\# A$   
**and** *empty*:  $P \{\#\}$   
**and** *insert*:  $\bigwedge a F. a \in\# A \implies P F \implies P (\text{add-mset } a F)$   
**shows**  $P F$

⟨proof⟩

## 67.6 Least and greatest elements

context begin

qualified lemma

assumes

$M \neq \{\#\}$  and

$\text{transp-on } (\text{set-mset } M) R$  and

$\text{totalp-on } (\text{set-mset } M) R$

shows

$\text{bex-least-element: } (\exists l \in \# M. \forall x \in \# M. x \neq l \longrightarrow R l x)$  and

$\text{bex-greatest-element: } (\exists g \in \# M. \forall x \in \# M. x \neq g \longrightarrow R x g)$

⟨proof⟩

end

## 67.7 The fold combinator

definition  $\text{fold-mset} :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ multiset} \Rightarrow 'b$

where

$\text{fold-mset } f s M = \text{Finite-Set.fold } (\lambda x. f x \sim\sim \text{count } M x) s (\text{set-mset } M)$

lemma  $\text{fold-mset-empty [simp]: fold-mset } f s \{\#\} = s$

⟨proof⟩

lemma  $\text{fold-mset-single [simp]: fold-mset } f s \{\#x\# \} = f x s$

⟨proof⟩

context  $\text{comp-fun-commute}$

begin

lemma  $\text{fold-mset-add-mset [simp]: fold-mset } f s (\text{add-mset } x M) = f x (\text{fold-mset } f s M)$

⟨proof⟩

lemma  $\text{fold-mset-fun-left-comm: } f x (\text{fold-mset } f s M) = \text{fold-mset } f (f x s) M$

⟨proof⟩

lemma  $\text{fold-mset-union [simp]: fold-mset } f s (M + N) = \text{fold-mset } f (\text{fold-mset } f s M) N$

⟨proof⟩

lemma  $\text{fold-mset-fusion:}$

assumes  $\text{comp-fun-commute } g$

and  $*$ :  $\bigwedge x y. h (g x y) = f x (h y)$

shows  $h (\text{fold-mset } g w A) = \text{fold-mset } f (h w) A$

⟨proof⟩

**end**

**lemma** *union-fold-mset-add-mset*:  $A + B = \text{fold-mset } \text{add-mset } A B$   
 ⟨*proof*⟩

A note on code generation: When defining some function containing a subterm *fold-mset*  $F$ , code generation is not automatic. When interpreting locale *left-commutative* with  $F$ , the would be code thms for *fold-mset* become thms like  $\text{fold-mset } F z \{\#\} = z$  where  $F$  is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for  $F$ . See the image operator below.

## 67.8 Image

**definition** *image-mset* ::  $('a \Rightarrow 'b) \Rightarrow 'a \text{ multiset} \Rightarrow 'b \text{ multiset}$  **where**  
 $\text{image-mset } f = \text{fold-mset } (\text{add-mset} \circ f) \{\#\}$

**lemma** *comp-fun-commute-mset-image*:  $\text{comp-fun-commute } (\text{add-mset} \circ f)$   
 ⟨*proof*⟩

**lemma** *image-mset-empty* [*simp*]:  $\text{image-mset } f \{\#\} = \{\#\}$   
 ⟨*proof*⟩

**lemma** *image-mset-single*:  $\text{image-mset } f \{\#x\# \} = \{\#f x\# \}$   
 ⟨*proof*⟩

**lemma** *image-mset-union* [*simp*]:  $\text{image-mset } f (M + N) = \text{image-mset } f M + \text{image-mset } f N$   
 ⟨*proof*⟩

**corollary** *image-mset-add-mset* [*simp*]:  
 $\text{image-mset } f (\text{add-mset } a M) = \text{add-mset } (f a) (\text{image-mset } f M)$   
 ⟨*proof*⟩

**lemma** *set-image-mset* [*simp*]:  $\text{set-mset } (\text{image-mset } f M) = \text{image } f (\text{set-mset } M)$   
 ⟨*proof*⟩

**lemma** *size-image-mset* [*simp*]:  $\text{size } (\text{image-mset } f M) = \text{size } M$   
 ⟨*proof*⟩

**lemma** *image-mset-is-empty-iff* [*simp*]:  $\text{image-mset } f M = \{\#\} \longleftrightarrow M = \{\#\}$   
 ⟨*proof*⟩

**lemma** *image-mset-If*:  
 $\text{image-mset } (\lambda x. \text{if } P x \text{ then } f x \text{ else } g x) A =$   
 $\text{image-mset } f (\text{filter-mset } P A) + \text{image-mset } g (\text{filter-mset } (\lambda x. \neg P x) A)$   
 ⟨*proof*⟩

**lemma** *filter-image-mset*:

*filter-mset*  $P$  (*image-mset*  $f$   $A$ ) = *image-mset*  $f$  (*filter-mset* ( $\lambda x. P (f x)$ )  $A$ )  
 ⟨*proof*⟩

**lemma** *image-mset-Diff*:

**assumes**  $B \subseteq\# A$

**shows** *image-mset*  $f$  ( $A - B$ ) = *image-mset*  $f$   $A -$  *image-mset*  $f$   $B$

⟨*proof*⟩

**lemma** *minus-add-mset-if-not-in-lhs[simp]*:  $x \notin\# A \implies A - \text{add-mset } x B = A - B$

⟨*proof*⟩

**lemma** *image-mset-diff-if-inj*:

**fixes**  $f A B$

**assumes** *inj*  $f$

**shows** *image-mset*  $f$  ( $A - B$ ) = *image-mset*  $f$   $A -$  *image-mset*  $f$   $B$

⟨*proof*⟩

**lemma** *count-image-mset*:

⟨*count* (*image-mset*  $f$   $A$ )  $x$  =  $(\sum y \in f^{-1} \{x\} \cap \text{set-mset } A. \text{count } A y)$ ⟩

⟨*proof*⟩

**lemma** *count-image-mset'*:

⟨*count* (*image-mset*  $f$   $X$ )  $y$  =  $(\sum x \mid x \in\# X \wedge y = f x. \text{count } X x)$ ⟩

⟨*proof*⟩

**lemma** *image-mset-subseteq-mono*:  $A \subseteq\# B \implies \text{image-mset } f A \subseteq\# \text{image-mset } f B$

⟨*proof*⟩

**lemma** *image-mset-subset-mono*:  $M \subset\# N \implies \text{image-mset } f M \subset\# \text{image-mset } f N$

⟨*proof*⟩

**syntax** (ASCII)

-*comprehension-mset* ::  $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow 'a \text{ multiset}$

(⟨⟨*notation*=⟨*mixfix multiset comprehension*⟩⟩{ $\#$ -/. - : $\#$  - $\#$ }⟩)⟩

**syntax**

-*comprehension-mset* ::  $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow 'a \text{ multiset}$

(⟨⟨*notation*=⟨*mixfix multiset comprehension*⟩⟩{ $\#$ -/. -  $\in\#$  - $\#$ }⟩)⟩

**syntax-consts**

-*comprehension-mset*  $\equiv$  *image-mset*

**translations**

{ $\#e. x \in\# M\#$ }  $\equiv$  *CONST image-mset* ( $\lambda x. e$ )  $M$

**syntax** (ASCII)

-*comprehension-mset'* ::  $'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow \text{bool} \Rightarrow 'a \text{ multiset}$

(⟨⟨*notation*=⟨*mixfix multiset comprehension*⟩⟩{ $\#$ -/ | - : $\#$  -/ - $\#$ }⟩)⟩

**syntax**

-comprehension-mset' :: 'a ⇒ 'b ⇒ 'b multiset ⇒ bool ⇒ 'a multiset  
 (⟨⟨notation=⟨mixfix multiset comprehension⟩{#-/ | - ∈# -./ -#}⟩⟩)

**syntax-consts**

-comprehension-mset' ⇒ image-mset

**translations**

{#e | x ∈# M. P#} → {#e. x ∈# {# x ∈# M. P#}#}

This allows to write not just filters like {#x ∈# M. x < c#} but also images like {#x + x. x ∈# M#} and {#x+x|x ∈# M. x < c#}, where the latter is currently displayed as {#x + x. x ∈# {#x ∈# M. x < c#}#}.

**lemma** *in-image-mset*:  $y \in \# \{ \# f x. x \in \# M \# \} \longleftrightarrow y \in f \text{ ' set-mset } M$   
 ⟨proof⟩

**functor** *image-mset*: *image-mset*

⟨proof⟩

**declare**

*image-mset.id* [simp]

*image-mset.identity* [simp]

**lemma** *image-mset-id*[simp]: *image-mset id x = x*

⟨proof⟩

**lemma** *image-mset-cong*:  $(\bigwedge x. x \in \# M \implies f x = g x) \implies \{ \# f x. x \in \# M \# \} = \{ \# g x. x \in \# M \# \}$

⟨proof⟩

**lemma** *image-mset-cong-pair*:

$(\forall x y. (x, y) \in \# M \longrightarrow f x y = g x y) \implies \{ \# f x y. (x, y) \in \# M \# \} = \{ \# g x y. (x, y) \in \# M \# \}$

⟨proof⟩

**lemma** *image-mset-const-eq*:

$\{ \# c. a \in \# M \# \} = \text{replicate-mset (size } M) c$

⟨proof⟩

**lemma** *image-mset-filter-mset-swap*:

*image-mset f (filter-mset (λx. P (f x)) M) = filter-mset P (image-mset f M)*

⟨proof⟩

**lemma** *image-mset-eq-plusD*:

*image-mset f A = B + C* ⇒ ∃ B' C'. *A = B' + C' ∧ B = image-mset f B' ∧ C = image-mset f C'*

⟨proof⟩

**lemma** *image-mset-eq-image-mset-plusD*:

**assumes** *image-mset f A = image-mset f B + C* **and** *inj-f: inj-on f (set-mset A ∪ set-mset B)*

**shows** ∃ C'. *A = B + C' ∧ C = image-mset f C'*

*<proof>*

**lemma** *image-mset-eq-plus-image-msetD*:

$image\text{-}mset\ f\ A = B + image\text{-}mset\ f\ C \implies inj\text{-}on\ f\ (set\text{-}mset\ A \cup set\text{-}mset\ C)$   
 $\implies$   
 $\exists B'. A = B' + C \wedge B = image\text{-}mset\ f\ B'$   
*<proof>*

## 67.9 Further conversions

**primrec** *mset* :: 'a list  $\Rightarrow$  'a multiset **where**

$mset\ [] = \{\#\}$  |  
 $mset\ (a\ \#\ x) = add\text{-}mset\ a\ (mset\ x)$

**lemma** *in-multiset-in-set*:

$x \in\ \#\ mset\ xs \longleftrightarrow x \in\ set\ xs$   
*<proof>*

**lemma** *count-mset*:

$count\ (mset\ xs)\ x = length\ (filter\ (\lambda y. x = y)\ xs)$   
*<proof>*

**lemma** *mset-zero-iff[simp]*:  $(mset\ x = \{\#\}) = (x = [])$   
*<proof>*

**lemma** *mset-zero-iff-right[simp]*:  $(\{\#\} = mset\ x) = (x = [])$   
*<proof>*

**lemma** *mset-replicate [simp]*:  $mset\ (replicate\ n\ x) = replicate\text{-}mset\ n\ x$   
*<proof>*

**lemma** *count-mset-gt-0*:  $x \in\ set\ xs \implies count\ (mset\ xs)\ x > 0$   
*<proof>*

**lemma** *count-mset-0-iff [simp]*:  $count\ (mset\ xs)\ x = 0 \longleftrightarrow x \notin\ set\ xs$   
*<proof>*

**lemma** *mset-single-iff[iff]*:  $mset\ xs = \{\#x\#\} \longleftrightarrow xs = [x]$   
*<proof>*

**lemma** *mset-single-iff-right[iff]*:  $\{\#x\#\} = mset\ xs \longleftrightarrow xs = [x]$   
*<proof>*

**lemma** *set-mset-mset[simp]*:  $set\text{-}mset\ (mset\ xs) = set\ xs$   
*<proof>*

**lemma** *set-mset-comp-mset [simp]*:  $set\text{-}mset\ \circ\ mset = set$   
*<proof>*

**lemma** *size-mset* [simp]:  $\text{size } (\text{mset } xs) = \text{length } xs$   
 ⟨proof⟩

**lemma** *mset-append* [simp]:  $\text{mset } (xs @ ys) = \text{mset } xs + \text{mset } ys$   
 ⟨proof⟩

**lemma** *mset-filter*[simp]:  $\text{mset } (\text{filter } P \ xs) = \{\#x \in\# \text{mset } xs. P \ x \ \#\}$   
 ⟨proof⟩

**lemma** *mset-rev* [simp]:  
 $\text{mset } (\text{rev } xs) = \text{mset } xs$   
 ⟨proof⟩

**lemma** *surj-mset*: *surj mset*  
 ⟨proof⟩

**lemma** *distinct-count-atmost-1*:  
 $\text{distinct } x = (\forall a. \text{count } (\text{mset } x) \ a = (\text{if } a \in \text{set } x \ \text{then } 1 \ \text{else } 0))$   
 ⟨proof⟩

**lemma** *mset-eq-setD*:  
**assumes**  $\text{mset } xs = \text{mset } ys$   
**shows**  $\text{set } xs = \text{set } ys$   
 ⟨proof⟩

**lemma** *set-eq-iff-mset-eq-distinct*:  
 $\langle \text{distinct } x \implies \text{distinct } y \implies \text{set } x = \text{set } y \longleftrightarrow \text{mset } x = \text{mset } y \rangle$   
 ⟨proof⟩

**lemma** *set-eq-iff-mset-remdups-eq*:  
 $\langle \text{set } x = \text{set } y \longleftrightarrow \text{mset } (\text{remdups } x) = \text{mset } (\text{remdups } y) \rangle$   
 ⟨proof⟩

**lemma** *mset-eq-imp-distinct-iff*:  
 $\langle \text{distinct } xs \longleftrightarrow \text{distinct } ys \rangle \ \mathbf{if} \ \langle \text{mset } xs = \text{mset } ys \rangle$   
 ⟨proof⟩

**lemma** *nth-mem-mset*:  $i < \text{length } ls \implies (ls ! i) \in\# \text{mset } ls$   
 ⟨proof⟩

**lemma** *mset-remove1*[simp]:  $\text{mset } (\text{remove1 } a \ xs) = \text{mset } xs - \{\#a\#\}$   
 ⟨proof⟩

**lemma** *mset-eq-length*:  
**assumes**  $\text{mset } xs = \text{mset } ys$   
**shows**  $\text{length } xs = \text{length } ys$   
 ⟨proof⟩

**lemma** *mset-eq-length-filter*:

**assumes**  $mset\ xs = mset\ ys$   
**shows**  $length\ (filter\ (\lambda x. z = x)\ xs) = length\ (filter\ (\lambda y. z = y)\ ys)$   
 $\langle proof \rangle$

**lemma** *fold-multiset-equiv*:  
 $\langle List.fold\ f\ xs = List.fold\ f\ ys \rangle$   
**if**  $f$ :  $\langle \bigwedge x\ y. x \in set\ xs \implies y \in set\ xs \implies f\ x \circ f\ y = f\ y \circ f\ x \rangle$   
**and**  $\langle mset\ xs = mset\ ys \rangle$   
 $\langle proof \rangle$

**lemma** *fold-permuted-eg*:  
 $\langle List.fold\ (\odot)\ xs\ z = List.fold\ (\odot)\ ys\ z \rangle$   
**if**  $\langle mset\ xs = mset\ ys \rangle$   
**and**  $\langle P\ z \rangle$  **and**  $P$ :  $\langle \bigwedge x\ z. x \in set\ xs \implies P\ z \implies P\ (x \odot z) \rangle$   
**and**  $f$ :  $\langle \bigwedge x\ y\ z. x \in set\ xs \implies y \in set\ xs \implies P\ z \implies x \odot (y \odot z) = y \odot (x \odot z) \rangle$   
**for**  $f$  (**infixl**  $\langle \odot \rangle$  70)  
 $\langle proof \rangle$

**lemma** *mset-shuffles*:  $zs \in shuffles\ xs\ ys \implies mset\ zs = mset\ xs + mset\ ys$   
 $\langle proof \rangle$

**lemma** *mset-insort [simp]*:  $mset\ (insort\ x\ xs) = add-mset\ x\ (mset\ xs)$   
 $\langle proof \rangle$

**lemma** *mset-map [simp]*:  $mset\ (map\ f\ xs) = image-mset\ f\ (mset\ xs)$   
 $\langle proof \rangle$

**global-interpretation** *mset-set*: *folding*  $add-mset\ \{\#\}$   
**defines**  $mset-set = folding-on.F\ add-mset\ \{\#\}$   
 $\langle proof \rangle$

**lemma** *sum-multiset-singleton [simp]*:  $sum\ (\lambda n. \{\#n\})\ A = mset-set\ A$   
 $\langle proof \rangle$

**lemma** *count-mset-set [simp]*:  
 $finite\ A \implies x \in A \implies count\ (mset-set\ A)\ x = 1$  (**is** *PROP* ?*P*)  
 $\neg finite\ A \implies count\ (mset-set\ A)\ x = 0$  (**is** *PROP* ?*Q*)  
 $x \notin A \implies count\ (mset-set\ A)\ x = 0$  (**is** *PROP* ?*R*)  
 $\langle proof \rangle$

**lemma** *elem-mset-set [simp, intro]*:  $finite\ A \implies x \in \# mset-set\ A \longleftrightarrow x \in A$   
 $\langle proof \rangle$

**lemma** *mset-set-Union*:  
 $finite\ A \implies finite\ B \implies A \cap B = \{\} \implies mset-set\ (A \cup B) = mset-set\ A + mset-set\ B$   
 $\langle proof \rangle$



**lemma** *filter-mset-mset-set* [simp]:

*finite A*  $\implies$  *filter-mset P* (*mset-set A*) = *mset-set* {*x* ∈ *A*. *P x*}  
 ⟨*proof*⟩

**lemma** *mset-set-Diff*:

**assumes** *finite A B*  $\subseteq$  *A*  
**shows** *mset-set (A - B)* = *mset-set A* - *mset-set B*  
 ⟨*proof*⟩

**lemma** *mset-set-set: distinct xs*  $\implies$  *mset-set (set xs)* = *mset xs*

⟨*proof*⟩

**lemma** *count-mset-set'*: *count (mset-set A) x* = (if *finite A*  $\wedge$  *x* ∈ *A* then 1 else 0)

⟨*proof*⟩

**lemma** *subset-imp-msubset-mset-set*:

**assumes** *A*  $\subseteq$  *B* *finite B*  
**shows** *mset-set A*  $\subseteq\#$  *mset-set B*  
 ⟨*proof*⟩

**lemma** *mset-set-set-mset-msubset*: *mset-set (set-mset A)*  $\subseteq\#$  *A*

⟨*proof*⟩

**lemma** *mset-set-upto-eq-mset-upto*:

⟨*mset-set* {..*n*} = *mset* [0..*n*]⟩  
 ⟨*proof*⟩

**context** *linorder*

**begin**

**definition** *sorted-list-of-multiset* :: 'a multiset  $\Rightarrow$  'a list

**where**

*sorted-list-of-multiset M* = *fold-mset insert* [] *M*

**lemma** *sorted-list-of-multiset-empty* [simp]:

*sorted-list-of-multiset* {#} = []  
 ⟨*proof*⟩

**lemma** *sorted-list-of-multiset-singleton* [simp]:

*sorted-list-of-multiset* {#*x*#} = [*x*]  
 ⟨*proof*⟩

**lemma** *sorted-list-of-multiset-insert* [simp]:

*sorted-list-of-multiset (add-mset x M)* = *List.insert x (sorted-list-of-multiset M)*  
 ⟨*proof*⟩

**end**

**lemma** *mset-sorted-list-of-multiset*[simp]:  $mset (sorted-list-of-multiset M) = M$   
 ⟨proof⟩

**lemma** *sorted-list-of-multiset-mset*[simp]:  $sorted-list-of-multiset (mset xs) = sort\ xs$   
 ⟨proof⟩

**lemma** *finite-set-mset-mset-set*[simp]:  $finite\ A \implies set-mset (mset-set\ A) = A$   
 ⟨proof⟩

**lemma** *mset-set-empty-iff*:  $mset-set\ A = \{\#\} \longleftrightarrow A = \{\} \vee infinite\ A$   
 ⟨proof⟩

**lemma** *infinite-set-mset-mset-set*:  $\neg\ finite\ A \implies set-mset (mset-set\ A) = \{\}$   
 ⟨proof⟩

**lemma** *set-sorted-list-of-multiset* [simp]:  
 $set (sorted-list-of-multiset\ M) = set-mset\ M$   
 ⟨proof⟩

**lemma** *sorted-list-of-mset-set* [simp]:  
 $sorted-list-of-multiset (mset-set\ A) = sorted-list-of-set\ A$   
 ⟨proof⟩

**lemma** *mset-upt* [simp]:  $mset [m..<n] = mset-set\ \{m..<n\}$   
 ⟨proof⟩

**lemma** *image-mset-map-of*:  
 $distinct (map\ fst\ xs) \implies \{\#the (map-of\ xs\ i). i \in \# mset (map\ fst\ xs)\#\} = mset (map\ snd\ xs)$   
 ⟨proof⟩

**lemma** *msubset-mset-set-iff*[simp]:  
**assumes**  $finite\ A\ finite\ B$   
**shows**  $mset-set\ A \subseteq \# mset-set\ B \longleftrightarrow A \subseteq B$   
 ⟨proof⟩

**lemma** *mset-set-eq-iff*[simp]:  
**assumes**  $finite\ A\ finite\ B$   
**shows**  $mset-set\ A = mset-set\ B \longleftrightarrow A = B$   
 ⟨proof⟩

**lemma** *image-mset-mset-set*:  
**assumes**  $inj-on\ f\ A$   
**shows**  $image-mset\ f (mset-set\ A) = mset-set (f\ 'A)$   
 ⟨proof⟩

### 67.10 More properties of the replicate, repeat, and image operations

**lemma** *in-replicate-mset[simp]*:  $x \in \# \text{ replicate-mset } n \ y \longleftrightarrow n > 0 \wedge x = y$   
 ⟨proof⟩

**lemma** *set-mset-replicate-mset-subset[simp]*:  $\text{set-mset } (\text{replicate-mset } n \ x) = (\text{if } n = 0 \text{ then } \{\} \text{ else } \{x\})$   
 ⟨proof⟩

**lemma** *size-replicate-mset[simp]*:  $\text{size } (\text{replicate-mset } n \ M) = n$   
 ⟨proof⟩

**lemma** *size-repeat-mset [simp]*:  $\text{size } (\text{repeat-mset } n \ A) = n * \text{size } A$   
 ⟨proof⟩

**lemma** *size-multiset-sum [simp]*:  $\text{size } (\sum x \in A. f \ x :: 'a \ \text{multiset}) = (\sum x \in A. \text{size } (f \ x))$   
 ⟨proof⟩

**lemma** *size-multiset-sum-list [simp]*:  $\text{size } (\sum X \leftarrow Xs. X :: 'a \ \text{multiset}) = (\sum X \leftarrow Xs. \text{size } X)$   
 ⟨proof⟩

**lemma** *count-le-replicate-mset-subset-eq*:  $n \leq \text{count } M \ x \longleftrightarrow \text{replicate-mset } n \ x \subseteq \# \ M$   
 ⟨proof⟩

**lemma** *replicate-count-mset-eq-filter-eq*:  $\text{replicate } (\text{count } (\text{mset } xs) \ k) \ k = \text{filter } (\text{HOL.eq } k) \ xs$   
 ⟨proof⟩

**lemma** *replicate-mset-eq-empty-iff [simp]*:  $\text{replicate-mset } n \ a = \{\#\} \longleftrightarrow n = 0$   
 ⟨proof⟩

**lemma** *replicate-mset-eq-iff*:  
 $\text{replicate-mset } m \ a = \text{replicate-mset } n \ b \longleftrightarrow m = 0 \wedge n = 0 \vee m = n \wedge a = b$   
 ⟨proof⟩

**lemma** *repeat-mset-cancel1*:  $\text{repeat-mset } a \ A = \text{repeat-mset } a \ B \longleftrightarrow A = B \vee a = 0$   
 ⟨proof⟩

**lemma** *repeat-mset-cancel2*:  $\text{repeat-mset } a \ A = \text{repeat-mset } b \ A \longleftrightarrow a = b \vee A = \{\#\}$   
 ⟨proof⟩

**lemma** *repeat-mset-eq-empty-iff*:  $\text{repeat-mset } n \ A = \{\#\} \longleftrightarrow n = 0 \vee A = \{\#\}$   
 ⟨proof⟩

**lemma** *image-replicate-mset* [*simp*]:

*image-mset f (replicate-mset n a) = replicate-mset n (f a)*

*<proof>*

**lemma** *replicate-mset-msubseteq-iff*:

*replicate-mset m a  $\subseteq\#$  replicate-mset n b  $\longleftrightarrow$   $m = 0 \vee a = b \wedge m \leq n$*

*<proof>*

**lemma** *msubseteq-replicate-msetE*:

**assumes** *A  $\subseteq\#$  replicate-mset n a*

**obtains** *m where  $m \leq n$  and  $A = \text{replicate-mset } m \ a$*

*<proof>*

**lemma** *count-image-mset-lt-imp-lt-raw*:

**assumes**

*finite A and*

*A = set-mset M  $\cup$  set-mset N and*

*count (image-mset f M) b < count (image-mset f N) b*

**shows**  *$\exists x. f x = b \wedge \text{count } M \ x < \text{count } N \ x$*

*<proof>*

**lemma** *count-image-mset-lt-imp-lt*:

**assumes** *cnt-b: count (image-mset f M) b < count (image-mset f N) b*

**shows**  *$\exists x. f x = b \wedge \text{count } M \ x < \text{count } N \ x$*

*<proof>*

**lemma** *count-image-mset-le-imp-lt-raw*:

**assumes**

*finite A and*

*A = set-mset M  $\cup$  set-mset N and*

*count (image-mset f M) (f a) + count N a < count (image-mset f N) (f a) + count M a*

**shows**  *$\exists b. f b = f a \wedge \text{count } M \ b < \text{count } N \ b$*

*<proof>*

**lemma** *count-image-mset-le-imp-lt*:

**assumes**

*count (image-mset f M) (f a)  $\leq$  count (image-mset f N) (f a) and*

*count M a > count N a*

**shows**  *$\exists b. f b = f a \wedge \text{count } M \ b < \text{count } N \ b$*

*<proof>*

**lemma** *size-filter-unsat-elem*:

**assumes**  *$x \in\# M$  and  $\neg P \ x$*

**shows** *size  $\{\#x \in\# M. P \ x\# \} < \text{size } M$*

*<proof>*

**lemma** *size-filter-ne-elem*:  *$x \in\# M \implies \text{size } \{\#y \in\# M. y \neq x\# \} < \text{size } M$*

*<proof>*

**lemma** *size-eq-ex-count-lt*:  
**assumes**  $size\ M = size\ N$  **and**  $M \neq N$   
**shows**  $\exists x. count\ M\ x < count\ N\ x$   
 $\langle proof \rangle$

## 67.11 Big operators

**locale** *comm-monoid-mset* = *comm-monoid*  
**begin**

**interpretation** *comp-fun-commute*  $f$   
 $\langle proof \rangle$

**interpretation** *comp?*: *comp-fun-commute*  $f \circ g$   
 $\langle proof \rangle$

**context**  
**begin**

**definition**  $F :: 'a\ multiset \Rightarrow 'a$   
**where** *eq-fold*:  $F\ M = fold\ mset\ f\ \mathbf{1}\ M$

**lemma** *empty* [*simp*]:  $F\ \{\#\} = \mathbf{1}$   
 $\langle proof \rangle$

**lemma** *singleton* [*simp*]:  $F\ \{\#x\# \} = x$   
 $\langle proof \rangle$

**lemma** *union* [*simp*]:  $F\ (M + N) = F\ M * F\ N$   
 $\langle proof \rangle$

**lemma** *add-mset* [*simp*]:  $F\ (add\ mset\ x\ N) = x * F\ N$   
 $\langle proof \rangle$

**lemma** *insert* [*simp*]:  
**shows**  $F\ (image\ mset\ g\ (add\ mset\ x\ A)) = g\ x * F\ (image\ mset\ g\ A)$   
 $\langle proof \rangle$

**lemma** *remove*:  
**assumes**  $x \in\# A$   
**shows**  $F\ A = x * F\ (A - \{\#x\# \})$   
 $\langle proof \rangle$

**lemma** *neutral*:  
 $\forall x \in\# A. x = \mathbf{1} \implies F\ A = \mathbf{1}$   
 $\langle proof \rangle$

**lemma** *neutral-const* [*simp*]:

$F (\text{image-mset } (\lambda-. \mathbf{1}) A) = \mathbf{1}$   
 ⟨proof⟩ **lemma** *F-image-mset-product*:  
 $F \{\#g \ x \ j * F \{\#g \ i \ j. i \in\# A\#\}, j \in\# B\#\} =$   
 $F (\text{image-mset } (g \ x) B) * F \{\#F \{\#g \ i \ j. i \in\# A\#\}. j \in\# B\#\}$   
 ⟨proof⟩

**lemma** *swap*:  
 $F (\text{image-mset } (\lambda i. F (\text{image-mset } (g \ i) B)) A) =$   
 $F (\text{image-mset } (\lambda j. F (\text{image-mset } (\lambda i. g \ i \ j) A)) B)$   
 ⟨proof⟩

**lemma** *distrib*:  $F (\text{image-mset } (\lambda x. g \ x * h \ x) A) = F (\text{image-mset } g \ A) * F$   
 $(\text{image-mset } h \ A)$   
 ⟨proof⟩

**lemma** *union-disjoint*:  
 $A \cap\# B = \{\#\} \implies F (\text{image-mset } g \ (A \cup\# B)) = F (\text{image-mset } g \ A) * F$   
 $(\text{image-mset } g \ B)$   
 ⟨proof⟩

**end**  
**end**

**lemma** *comp-fun-commute-plus-mset[simp]*: *comp-fun-commute*  $((+) :: 'a \text{ multiset}$   
 $\Rightarrow - \Rightarrow -)$   
 ⟨proof⟩

**declare** *comp-fun-commute.fold-mset-add-mset[OF comp-fun-commute-plus-mset,*  
*simp]*

**lemma** *in-mset-fold-plus-iff[iff]*:  $x \in\# \text{fold-mset } (+) \ M \ NN \longleftrightarrow x \in\# M \vee (\exists N.$   
 $N \in\# NN \wedge x \in\# N)$   
 ⟨proof⟩

**context** *comm-monoid-add*  
**begin**

**sublocale** *sum-mset: comm-monoid-mset plus 0*  
**defines** *sum-mset = sum-mset.F* ⟨proof⟩

**lemma** *sum-unfold-sum-mset*:  
 $\text{sum } f \ A = \text{sum-mset } (\text{image-mset } f \ (\text{mset-set } A))$   
 ⟨proof⟩

**end**

**notation** *sum-mset*  $(\langle \sum \#\rangle)$

**syntax** *(ASCII)*

```

-sum-mset-image :: pttrn ⇒ 'b set ⇒ 'a ⇒ 'a::comm-monoid-add
  (⟨⟨indent=3 notation=binder SUM⟩⟩SUM -:#-. -) [0, 51, 10] 10)
syntax
-sum-mset-image :: pttrn ⇒ 'b set ⇒ 'a ⇒ 'a::comm-monoid-add
  (⟨⟨indent=3 notation=binder ∑⟩⟩∑ -:#-. -) [0, 51, 10] 10)
syntax-consts
-sum-mset-image ⇒ sum-mset
translations
  ∑ i ∈# A. b ⇒ CONST sum-mset (CONST image-mset (λi. b) A)

context comm-monoid-add
begin

lemma sum-mset-sum-list:
  sum-mset (mset xs) = sum-list xs
  ⟨proof⟩

end

context canonically-ordered-monoid-add
begin

lemma sum-mset-0-iff [simp]:
  sum-mset M = 0 ↔ (∀ x ∈ set-mset M. x = 0)
  ⟨proof⟩

end

context ordered-comm-monoid-add
begin

lemma sum-mset-mono:
  sum-mset (image-mset f K) ≤ sum-mset (image-mset g K)
  if ∧i. i ∈# K ⇒ f i ≤ g i
  ⟨proof⟩

end

context cancel-comm-monoid-add
begin

lemma sum-mset-diff:
  sum-mset (M - N) = sum-mset M - sum-mset N if N ⊆# M for M N :: 'a
  multiset
  ⟨proof⟩

end

context semiring-0

```

**begin**

**lemma** *sum-mset-distrib-left*:

$$c * (\sum x \in\# M. f x) = (\sum x \in\# M. c * f(x))$$

*<proof>*

**lemma** *sum-mset-distrib-right*:

$$(\sum x \in\# M. f x) * c = (\sum x \in\# M. f x * c)$$

*<proof>*

**end**

**lemma** *sum-mset-product*:

**fixes**  $f :: 'a::\{comm-monoid-add,times\} \Rightarrow 'b::semiring-0$   
**shows**  $(\sum i \in\# A. f i) * (\sum i \in\# B. g i) = (\sum i \in\# A. \sum j \in\# B. f i * g j)$   
*<proof>*

**context** *semiring-1*

**begin**

**lemma** *sum-mset-replicate-mset* [*simp*]:

$$sum-mset (replicate-mset n a) = of-nat n * a$$

*<proof>*

**lemma** *sum-mset-delta*:

$$sum-mset (image-mset (\lambda x. if x = y then c else 0) A) = c * of-nat (count A y)$$

*<proof>*

**lemma** *sum-mset-delta'*:

$$sum-mset (image-mset (\lambda x. if y = x then c else 0) A) = c * of-nat (count A y)$$

*<proof>*

**end**

**lemma** *of-nat-sum-mset* [*simp*]:

$$of-nat (sum-mset A) = sum-mset (image-mset of-nat A)$$

*<proof>*

**lemma** *size-eq-sum-mset*:

$$size M = (\sum a \in\# M. 1)$$

*<proof>*

**lemma** *size-mset-set* [*simp*]:

$$size (mset-set A) = card A$$

*<proof>*

**lemma** *sum-mset-constant* [*simp*]:

**fixes**  $y :: 'b::semiring-1$   
**shows**  $\langle \sum x \in\# A. y \rangle = of-nat (size A) * y$



*<proof>*

**lemma** *set-mset-Union-mset[simp]*:  $set\text{-}mset (\sum \# MM) = (\bigcup M \in set\text{-}mset MM. set\text{-}mset M)$   
*<proof>*

**lemma** *in-Union-mset-iff[iff]*:  $x \in \# \sum \# MM \longleftrightarrow (\exists M. M \in \# MM \wedge x \in \# M)$   
*<proof>*

**lemma** *count-sum*:  
 $count (sum f A) x = sum (\lambda a. count (f a) x) A$   
*<proof>*

**lemma** *sum-eq-empty-iff*:  
**assumes** *finite A*  
**shows**  $sum f A = \{\#\} \longleftrightarrow (\forall a \in A. f a = \{\#\})$   
*<proof>*

**lemma** *mset-concat*:  $mset (concat xss) = (\sum xs \leftarrow xss. mset xs)$   
*<proof>*

**lemma** *sum-mset-singleton-mset [simp]*:  $(\sum x \in \# A. \{\#f x\# \}) = image\text{-}mset f A$   
*<proof>*

**lemma** *sum-list-singleton-mset [simp]*:  $(\sum x \leftarrow xs. \{\#f x\# \}) = image\text{-}mset f (mset xs)$   
*<proof>*

**lemma** *Union-mset-empty-conv[simp]*:  $\sum \# M = \{\#\} \longleftrightarrow (\forall i \in \# M. i = \{\#\})$   
*<proof>*

**lemma** *Union-image-single-mset[simp]*:  $\sum \# (image\text{-}mset (\lambda x. \{\#x\# \}) m) = m$   
*<proof>*

**lemma** *size-multiset-sum-mset [simp]*:  $size (\sum X \in \# A. X :: 'a multiset) = (\sum X \in \# A. size X)$   
*<proof>*

**context** *comm-monoid-mult*  
**begin**

**sublocale** *prod-mset: comm-monoid-mset times 1*  
**defines**  $prod\text{-}mset = prod\text{-}mset.F$  *<proof>*

**lemma** *prod-mset-empty*:  
 $prod\text{-}mset \{\#\} = 1$   
*<proof>*

**lemma** *prod-mset-singleton*:

*prod-mset* {#x#} = *x*  
 ⟨*proof*⟩

**lemma** *prod-mset-Un*:

*prod-mset* (*A* + *B*) = *prod-mset* *A* \* *prod-mset* *B*  
 ⟨*proof*⟩

**lemma** *prod-mset-prod-list*:

*prod-mset* (*mset xs*) = *prod-list xs*  
 ⟨*proof*⟩

**lemma** *prod-mset-replicate-mset* [*simp*]:

*prod-mset* (*replicate-mset n a*) = *a* <sup>^</sup> *n*  
 ⟨*proof*⟩

**lemma** *prod-unfold-prod-mset*:

*prod f A* = *prod-mset* (*image-mset f* (*mset-set A*))  
 ⟨*proof*⟩

**lemma** *prod-mset-multiplicity*:

*prod-mset M* = *prod* ( $\lambda x. x$  <sup>^</sup> *count M x*) (*set-mset M*)  
 ⟨*proof*⟩

**lemma** *prod-mset-delta*: *prod-mset* (*image-mset* ( $\lambda x. \text{if } x = y \text{ then } c \text{ else } 1$ ) *A*) =  
*c* <sup>^</sup> *count A y*

⟨*proof*⟩

**lemma** *prod-mset-delta'*: *prod-mset* (*image-mset* ( $\lambda x. \text{if } y = x \text{ then } c \text{ else } 1$ ) *A*) =  
*c* <sup>^</sup> *count A y*

⟨*proof*⟩

**lemma** *prod-mset-subset-imp-dvd*:

**assumes** *A*  $\subseteq\#$  *B*

**shows** *prod-mset A dvd prod-mset B*

⟨*proof*⟩

**lemma** *dvd-prod-mset*:

**assumes** *x*  $\in\#$  *A*

**shows** *x dvd prod-mset A*

⟨*proof*⟩

**end**

**notation** *prod-mset* ( $\langle \prod \# \rangle$ )

**syntax** (*ASCII*)

*-prod-mset-image* :: *pttrn*  $\Rightarrow$  '*b set*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a::comm-monoid-mult*

( $\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } PROD \rangle \rangle PROD \text{ } -: \# \text{ } - \text{ } \rangle [0, 51, 10] 10$ )

**syntax**

*-prod-mset-image* :: *pttrn*  $\Rightarrow$  'b set  $\Rightarrow$  'a  $\Rightarrow$  'a::comm-monoid-mult  
 ( $\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \prod \rangle \prod -\in\#-. \rangle [0, 51, 10] 10 \rangle$ )

**syntax-consts**

*-prod-mset-image*  $\equiv$  *prod-mset*

**translations**

$\prod i \in\# A. b \equiv \text{CONST } \text{prod-mset } (\text{CONST } \text{image-mset } (\lambda i. b) A)$

**lemma** *prod-mset-constant* [*simp*]:  $(\prod -\in\# A. c) = c \wedge \text{size } A$   
 $\langle \text{proof} \rangle$

**lemma** (in *semidom*) *prod-mset-zero-iff* [*iff*]:  
 $\text{prod-mset } A = 0 \longleftrightarrow 0 \in\# A$   
 $\langle \text{proof} \rangle$

**lemma** (in *semidom-divide*) *prod-mset-diff*:  
 assumes  $B \subseteq\# A$  and  $0 \notin\# B$   
 shows  $\text{prod-mset } (A - B) = \text{prod-mset } A \text{ div } \text{prod-mset } B$   
 $\langle \text{proof} \rangle$

**lemma** (in *semidom-divide*) *prod-mset-minus*:  
 assumes  $a \in\# A$  and  $a \neq 0$   
 shows  $\text{prod-mset } (A - \{\#a\}) = \text{prod-mset } A \text{ div } a$   
 $\langle \text{proof} \rangle$

**lemma** (in *normalization-semidom*) *normalize-prod-mset-normalize*:  
 $\text{normalize } (\text{prod-mset } (\text{image-mset } \text{normalize } A)) = \text{normalize } (\text{prod-mset } A)$   
 $\langle \text{proof} \rangle$

**lemma** (in *algebraic-semidom*) *is-unit-prod-mset-iff*:  
 $\text{is-unit } (\text{prod-mset } A) \longleftrightarrow (\forall x \in\# A. \text{is-unit } x)$   
 $\langle \text{proof} \rangle$

**lemma** (in *normalization-semidom-multiplicative*) *normalize-prod-mset*:  
 $\text{normalize } (\text{prod-mset } A) = \text{prod-mset } (\text{image-mset } \text{normalize } A)$   
 $\langle \text{proof} \rangle$

**lemma** (in *normalization-semidom-multiplicative*) *normalized-prod-msetI*:  
 assumes  $\bigwedge a. a \in\# A \implies \text{normalize } a = a$   
 shows  $\text{normalize } (\text{prod-mset } A) = \text{prod-mset } A$   
 $\langle \text{proof} \rangle$

**lemma** *image-prod-mset-multiplicity*:  
 $\text{prod-mset } (\text{image-mset } f M) = \text{prod } (\lambda x. f x \wedge \text{count } M x) (\text{set-mset } M)$   
 $\langle \text{proof} \rangle$

## 67.12 Multiset as order-ignorant lists

**context** *linorder*

**begin**

**lemma** *mset-insort* [*simp*]:  
 $mset (insort\text{-}key\ k\ x\ xs) = add\text{-}mset\ x\ (mset\ xs)$   
 ⟨*proof*⟩

**lemma** *mset-sort* [*simp*]:  
 $mset (sort\text{-}key\ k\ xs) = mset\ xs$   
 ⟨*proof*⟩

This lemma shows which properties suffice to show that a function  $f$  with  $f\ xs = ys$  behaves like `sort`.

**lemma** *properties-for-sort-key*:  
**assumes**  $mset\ ys = mset\ xs$   
**and**  $\bigwedge k. k \in set\ ys \implies filter\ (\lambda x. f\ k = f\ x)\ ys = filter\ (\lambda x. f\ k = f\ x)\ xs$   
**and**  $sorted\ (map\ f\ ys)$   
**shows**  $sort\text{-}key\ f\ xs = ys$   
 ⟨*proof*⟩

**lemma** *properties-for-sort*:  
**assumes** *multiset*:  $mset\ ys = mset\ xs$   
**and**  $sorted\ ys$   
**shows**  $sort\ xs = ys$   
 ⟨*proof*⟩

**lemma** *sort-key-inj-key-eq*:  
**assumes** *mset-equal*:  $mset\ xs = mset\ ys$   
**and**  $inj\text{-}on\ f\ (set\ xs)$   
**and**  $sorted\ (map\ f\ ys)$   
**shows**  $sort\text{-}key\ f\ xs = ys$   
 ⟨*proof*⟩

**lemma** *sort-key-eq-sort-key*:  
**assumes**  $mset\ xs = mset\ ys$   
**and**  $inj\text{-}on\ f\ (set\ xs)$   
**shows**  $sort\text{-}key\ f\ xs = sort\text{-}key\ f\ ys$   
 ⟨*proof*⟩

**lemma** *sort-key-by-quicksort*:  
 $sort\text{-}key\ f\ xs = sort\text{-}key\ f\ [x \leftarrow xs. f\ x < f\ (xs\ !\ (length\ xs\ div\ 2))]$   
 @  $[x \leftarrow xs. f\ x = f\ (xs\ !\ (length\ xs\ div\ 2))]$   
 @  $sort\text{-}key\ f\ [x \leftarrow xs. f\ x > f\ (xs\ !\ (length\ xs\ div\ 2))]$  (**is**  $sort\text{-}key\ f\ ?lhs = ?rhs$ )  
 ⟨*proof*⟩

**lemma** *sort-by-quicksort*:  
 $sort\ xs = sort\ [x \leftarrow xs. x < xs\ !\ (length\ xs\ div\ 2)]$   
 @  $[x \leftarrow xs. x = xs\ !\ (length\ xs\ div\ 2)]$   
 @  $sort\ [x \leftarrow xs. x > xs\ !\ (length\ xs\ div\ 2)]$  (**is**  $sort\ ?lhs = ?rhs$ )  
 ⟨*proof*⟩

**lemma** *sort-append*:

**assumes**  $\bigwedge x y. x \in \text{set } xs \implies y \in \text{set } ys \implies x \leq y$   
**shows**  $\text{sort } (xs @ ys) = \text{sort } xs @ \text{sort } ys$   
 $\langle \text{proof} \rangle$

**lemma** *sort-append-replicate-left*:

$(\bigwedge y. y \in \text{set } xs \implies x \leq y) \implies \text{sort } (\text{replicate } n \ x @ xs) = \text{replicate } n \ x @ \text{sort } xs$   
 $\langle \text{proof} \rangle$

**lemma** *sort-append-replicate-right*:

$(\bigwedge y. y \in \text{set } xs \implies x \geq y) \implies \text{sort } (xs @ \text{replicate } n \ x) = \text{sort } xs @ \text{replicate } n \ x$   
 $\langle \text{proof} \rangle$

A stable parameterized quicksort

**definition** *part* ::  $('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b \ \text{list} \Rightarrow 'b \ \text{list} \times 'b \ \text{list} \times 'b \ \text{list}$  **where**

$\text{part } f \ \text{pivot } xs = ([x \leftarrow xs. f \ x < \text{pivot}], [x \leftarrow xs. f \ x = \text{pivot}], [x \leftarrow xs. \text{pivot} < f \ x])$

**lemma** *part-code* [code]:

$\text{part } f \ \text{pivot } [] = ([], [], [])$   
 $\text{part } f \ \text{pivot } (x \# xs) = (\text{let } (lts, eqs, gts) = \text{part } f \ \text{pivot } xs; x' = f \ x \ \text{in}$   
 $\text{if } x' < \text{pivot} \ \text{then } (x \# lts, eqs, gts)$   
 $\text{else if } x' > \text{pivot} \ \text{then } (lts, eqs, x \# gts)$   
 $\text{else } (lts, x \# eqs, gts))$   
 $\langle \text{proof} \rangle$

**lemma** *sort-key-by-quicksort-code* [code]:

$\text{sort-key } f \ xs =$   
 $(\text{case } xs \ \text{of}$   
 $\quad [] \Rightarrow []$   
 $\quad | [x] \Rightarrow xs$   
 $\quad | [x, y] \Rightarrow (\text{if } f \ x \leq f \ y \ \text{then } xs \ \text{else } [y, x])$   
 $\quad | - \Rightarrow$   
 $\quad \text{let } (lts, eqs, gts) = \text{part } f \ (f \ (xs \ ! \ (\text{length } xs \ \text{div } 2))) \ \text{in}$   
 $\quad \text{sort-key } f \ lts @ eqs @ \text{sort-key } f \ gts)$   
 $\langle \text{proof} \rangle$

**end**

**hide-const** (open) *part*

**lemma** *mset-remdups-subset-eq*:  $\text{mset } (\text{remdups } xs) \subseteq\# \text{mset } xs$

$\langle \text{proof} \rangle$

**lemma** *mset-update*:

$i < \text{length } ls \implies \text{mset } (ls[i := v]) = \text{add-mset } v \ (\text{mset } ls - \{\#ls \ ! \ i\#})$   
 $\langle \text{proof} \rangle$

**lemma** *mset-swap*:

$i < \text{length } ls \implies j < \text{length } ls \implies$   
 $\text{mset } (ls[j := ls ! i, i := ls ! j]) = \text{mset } ls$   
 ⟨proof⟩

**lemma** *mset-eq-finite*:

⟨finite {ys. mset ys = mset xs}⟩  
 ⟨proof⟩

### 67.13 The multiset order

**definition** *mult1* :: ('a × 'a) set ⇒ ('a multiset × 'a multiset) set **where**

$\text{mult1 } r = \{(N, M). \exists a MO K. M = \text{add-mset } a MO \wedge N = MO + K \wedge$   
 $(\forall b. b \in\# K \longrightarrow (b, a) \in r)\}$

**definition** *mult* :: ('a × 'a) set ⇒ ('a multiset × 'a multiset) set **where**

$\text{mult } r = (\text{mult1 } r)^+$

**definition** *multp* :: ('a ⇒ 'a ⇒ bool) ⇒ 'a multiset ⇒ 'a multiset ⇒ bool **where**

$\text{multp } r M N \longleftrightarrow (M, N) \in \text{mult } \{(x, y). r x y\}$

**declare** *multp-def*[pred-set-conv]

**lemma** *mult1I*:

**assumes**  $M = \text{add-mset } a MO$  **and**  $N = MO + K$  **and**  $\bigwedge b. b \in\# K \implies (b, a) \in r$   
 ∈  $r$   
**shows**  $(N, M) \in \text{mult1 } r$   
 ⟨proof⟩

**lemma** *mult1E*:

**assumes**  $(N, M) \in \text{mult1 } r$   
**obtains**  $a MO K$  **where**  $M = \text{add-mset } a MO$   $N = MO + K$   $\bigwedge b. b \in\# K \implies$   
 $(b, a) \in r$   
 ⟨proof⟩

**lemma** *mono-mult1*:

**assumes**  $r \subseteq r'$  **shows**  $\text{mult1 } r \subseteq \text{mult1 } r'$   
 ⟨proof⟩

**lemma** *mono-mult*:

**assumes**  $r \subseteq r'$  **shows**  $\text{mult } r \subseteq \text{mult } r'$   
 ⟨proof⟩

**lemma** *mono-multp*[mono]:  $r \leq r' \implies \text{multp } r \leq \text{multp } r'$

⟨proof⟩

**lemma** *not-less-empty* [iff]:  $(M, \{\#\}) \notin \text{mult1 } r$

⟨proof⟩

**67.13.1 Well-foundedness****lemma** *less-add*:**assumes**  $mult1: (N, add-mset\ a\ M0) \in mult1\ r$ **shows**

$$(\exists M. (M, M0) \in mult1\ r \wedge N = add-mset\ a\ M) \vee$$

$$(\exists K. (\forall b. b \in\# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$$
*<proof>***lemma** *all-accessible*:**assumes**  $wf\ r$ **shows**  $\forall M. M \in Wellfounded.acc\ (mult1\ r)$ *<proof>***lemma** *wf-mult1*:  $wf\ r \implies wf\ (mult1\ r)$ *<proof>***lemma** *wf-mult*:  $wf\ r \implies wf\ (mult\ r)$ *<proof>***lemma** *wfp-multp*:  $wfp\ r \implies wfp\ (multp\ r)$ *<proof>***67.13.2 Closure-free presentation**

One direction.

**lemma** *mult-implies-one-step*:**assumes***trans*:  $trans\ r$  **and***MN*:  $(M, N) \in mult\ r$ **shows**  $\exists I\ J\ K. N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k \in set-mset\ K. \exists j \in set-mset\ J. (k, j) \in r)$ *<proof>***lemma** *multp-implies-one-step*:*transp*  $R \implies multp\ R\ M\ N \implies \exists I\ J\ K. N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k \in\# K. \exists x \in\# J. R\ k\ x)$ *<proof>***lemma** *one-step-implies-mult*:**assumes** $J \neq \{\#\}$  **and** $\forall k \in set-mset\ K. \exists j \in set-mset\ J. (k, j) \in r$ **shows**  $(I + K, I + J) \in mult\ r$ *<proof>***lemma** *one-step-implies-multp*: $J \neq \{\#\} \implies \forall k \in\# K. \exists j \in\# J. R\ k\ j \implies multp\ R\ (I + K)\ (I + J)$ *<proof>*

**lemma** *subset-implies-mult*:

**assumes** *sub*:  $A \subset\# B$

**shows**  $(A, B) \in \text{mult } r$

*<proof>*

**lemma** *subset-implies-multp*:  $A \subset\# B \implies \text{multp } r A B$

*<proof>*

**lemma** *multp-repeat-mset-repeat-msetI*:

**assumes** *transp R* **and** *multp R A B* **and**  $n \neq 0$

**shows**  $\text{multp } R (\text{repeat-mset } n A) (\text{repeat-mset } n B)$

*<proof>*

### 67.13.3 Monotonicity

**lemma** *multp-mono-strong*:

**assumes**  $\text{multp } R M1 M2$  **and** *transp R* **and**

*S-if-R*:  $\bigwedge x y. x \in \text{set-mset } M1 \implies y \in \text{set-mset } M2 \implies R x y \implies S x y$

**shows**  $\text{multp } S M1 M2$

*<proof>*

**lemma** *mult-mono-strong*:

**assumes**  $(M1, M2) \in \text{mult } r$  **and** *trans r* **and**

*S-if-R*:  $\bigwedge x y. x \in \text{set-mset } M1 \implies y \in \text{set-mset } M2 \implies (x, y) \in r \implies (x, y)$

$\in s$

**shows**  $(M1, M2) \in \text{mult } s$

*<proof>*

**lemma** *monotone-on-multp-multp-image-mset*:

**assumes** *monotone-on A orda ordb f* **and** *transp orda*

**shows** *monotone-on*  $\{M. \text{set-mset } M \subseteq A\}$   $(\text{multp } orda)$   $(\text{multp } ordb)$   $(\text{image-mset } f)$

*<proof>*

**lemma** *monotone-multp-multp-image-mset*:

**assumes** *monotone orda ordb f* **and** *transp orda*

**shows** *monotone*  $(\text{multp } orda)$   $(\text{multp } ordb)$   $(\text{image-mset } f)$

*<proof>*

**lemma** *multp-image-mset-image-msetI*:

**assumes**  $\text{multp } (\lambda x y. R (f x) (f y)) M1 M2$  **and** *transp R*

**shows**  $\text{multp } R (\text{image-mset } f M1) (\text{image-mset } f M2)$

*<proof>*

**lemma** *multp-image-mset-image-msetD*:

**assumes**

$\text{multp } R (\text{image-mset } f A) (\text{image-mset } f B)$  **and**

*transp R* **and**



*inj-on-f*: *inj-on*  $f$  (*set-mset*  $A \cup \text{set-mset } B$ )  
**shows** *multp*  $(\lambda x y. R (f x) (f y)) A B$   
 ⟨*proof*⟩

#### 67.13.4 The multiset extension is cancellative for multiset union

**lemma** *mult-cancel*:

**assumes** *trans*  $s$  **and** *irrefl-on* (*set-mset*  $Z$ )  $s$   
**shows**  $(X + Z, Y + Z) \in \text{mult } s \longleftrightarrow (X, Y) \in \text{mult } s$  (**is**  $?L \longleftrightarrow ?R$ )  
 ⟨*proof*⟩

**lemma** *multp-cancel*:

*transp*  $R \implies \text{irreflp-on } (\text{set-mset } Z) R \implies \text{multp } R (X + Z) (Y + Z) \longleftrightarrow$   
*multp*  $R X Y$   
 ⟨*proof*⟩

**lemma** *mult-cancel-add-mset*:

*trans*  $r \implies \text{irrefl-on } \{z\} r \implies$   
 $((\text{add-mset } z X, \text{add-mset } z Y) \in \text{mult } r) = ((X, Y) \in \text{mult } r)$   
 ⟨*proof*⟩

**lemma** *multp-cancel-add-mset*:

*transp*  $R \implies \text{irreflp-on } \{z\} R \implies \text{multp } R (\text{add-mset } z X) (\text{add-mset } z Y) =$   
*multp*  $R X Y$   
 ⟨*proof*⟩

**lemma** *mult-cancel-max0*:

**assumes** *trans*  $s$  **and** *irrefl-on* (*set-mset*  $X \cap \text{set-mset } Y$ )  $s$   
**shows**  $(X, Y) \in \text{mult } s \longleftrightarrow (X - X \cap \# Y, Y - X \cap \# Y) \in \text{mult } s$  (**is**  $?L$   
 $\longleftrightarrow ?R$ )  
 ⟨*proof*⟩

**lemma** *mult-cancel-max*:

*trans*  $r \implies \text{irrefl-on } (\text{set-mset } X \cap \text{set-mset } Y) r \implies$   
 $(X, Y) \in \text{mult } r \longleftrightarrow (X - Y, Y - X) \in \text{mult } r$   
 ⟨*proof*⟩

**lemma** *multp-cancel-max*:

*transp*  $R \implies \text{irreflp-on } (\text{set-mset } X \cap \text{set-mset } Y) R \implies \text{multp } R X Y \longleftrightarrow$   
*multp*  $R (X - Y) (Y - X)$   
 ⟨*proof*⟩

#### 67.13.5 Strict partial-order properties

**lemma** *mult1-lessE*:

**assumes**  $(N, M) \in \text{mult1 } \{(a, b). r a b\}$  **and** *asympt*  $r$   
**obtains**  $a M0 K$  **where**  $M = \text{add-mset } a M0 N = M0 + K$   
 $a \notin \# K \wedge b. b \in \# K \implies r b a$   
 ⟨*proof*⟩

**lemma** *trans-on-mult*:

**assumes** *trans-on A r* **and**  $\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$

**shows** *trans-on B (mult r)*

*<proof>*

**lemma** *trans-mult*: *trans r*  $\implies$  *trans (mult r)*

*<proof>*

**lemma** *transp-on-multp*:

**assumes** *transp-on A r* **and**  $\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$

**shows** *transp-on B (multp r)*

*<proof>*

**lemma** *transp-multp*: *transp r*  $\implies$  *transp (multp r)*

*<proof>*

**lemma** *irrefl-mult*:

**assumes** *trans r irrefl r*

**shows** *irrefl (mult r)*

*<proof>*

**lemma** *irreflp-multp*: *transp R*  $\implies$  *irreflp R*  $\implies$  *irreflp (multp R)*

*<proof>*

**instantiation** *multiset* :: (*preorder*) *order begin*

**definition** *less-multiset* :: '*a multiset*  $\Rightarrow$  '*a multiset*  $\Rightarrow$  *bool*

**where**  $M < N \longleftrightarrow \text{multp } (<) M N$

**definition** *less-eq-multiset* :: '*a multiset*  $\Rightarrow$  '*a multiset*  $\Rightarrow$  *bool*

**where**  $\text{less-eq-multiset } M N \longleftrightarrow M < N \vee M = N$

**instance**

*<proof>*

**end**

**lemma** *mset-le-irrefl* [*elim!*]:

**fixes**  $M :: 'a :: \text{preorder multiset}$

**shows**  $M < M \implies \text{False}$

*<proof>*

**lemma** *wfp-less-multiset*[*simp*]:

**assumes**  $wf: wfp ((<) :: ('a :: \text{preorder}) \Rightarrow 'a \Rightarrow \text{bool})$

**shows**  $wfp ((\text{less-eq-multiset}) :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool})$

*<proof>*

**67.13.6 Strict total-order properties****lemma** *total-on-mult*:

**assumes** *total-on A r* and *trans r* and  $\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$   
**shows** *total-on B (mult r)*

*<proof>***lemma** *total-mult*: *total r*  $\implies$  *trans r*  $\implies$  *total (mult r)**<proof>***lemma** *totalp-on-multip*:

*totalp-on A R*  $\implies$  *transp R*  $\implies$   $(\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A) \implies \text{totalp-on } B \text{ (multip R)}$

*<proof>***lemma** *totalp-multip*: *totalp R*  $\implies$  *transp R*  $\implies$  *totalp (multip R)**<proof>***67.14 Quasi-executable version of the multiset extension**

Predicate variants of *mult* and the reflexive closure of *mult*, which are executable whenever the given predicate *P* is. Together with the standard code equations for  $(\cap\#)$  and  $(-)$  this should yield quadratic (with respect to calls to *P*) implementations of *multip-code* and *multeqp-code*.

**definition** *multip-code* ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$   
**where**

*multip-code P N M* =  
 $(\text{let } Z = M \cap\# N; X = M - Z \text{ in}$   
 $X \neq \{\#\} \wedge (\text{let } Y = N - Z \text{ in } (\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. P y x)))$

**definition** *multeqp-code* ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$   
**where**

*multeqp-code P N M* =  
 $(\text{let } Z = M \cap\# N; X = M - Z; Y = N - Z \text{ in}$   
 $(\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. P y x))$

**lemma** *multip-code-iff-mult*:

**assumes** *irrefl-on (set-mset N  $\cap$  set-mset M) R* and *trans R* and  
 $[simp]: \bigwedge x y. P x y \longleftrightarrow (x, y) \in R$   
**shows** *multip-code P N M*  $\longleftrightarrow$   $(N, M) \in \text{mult } R$  (**is**  $?L \longleftrightarrow ?R$ )

*<proof>***lemma** *multip-code-iff-multip*:

*irrefl-on (set-mset M  $\cap$  set-mset N) R*  $\implies$  *transp R*  $\implies$  *multip-code R M N*  
 $\longleftrightarrow$  *multip R M N*

*<proof>***lemma** *multip-code-eq-multip*:

**assumes** *irrefl R* and *transp R*

**shows** *multp-code*  $R = \text{multp } R$   
 ⟨*proof*⟩

**lemma** *multeqp-code-iff-reflcl-mult*:

**assumes** *irrefl-on*  $(\text{set-mset } N \cap \text{set-mset } M) R$  **and** *trans*  $R$  **and**  $\bigwedge x y. P x y$   
 $\longleftrightarrow (x, y) \in R$

**shows** *multeqp-code*  $P N M \longleftrightarrow (N, M) \in (\text{mult } R)^=$   
 ⟨*proof*⟩

**lemma** *multeqp-code-iff-reflclp-multp*:

*irreflp-on*  $(\text{set-mset } M \cap \text{set-mset } N) R \implies \text{transp } R \implies \text{multeqp-code } R M N$   
 $\longleftrightarrow (\text{multp } R)^{==} M N$

⟨*proof*⟩

**lemma** *multeqp-code-eq-reflclp-multp*:

**assumes** *irreflp*  $R$  **and** *transp*  $R$

**shows** *multeqp-code*  $R = (\text{multp } R)^{==}$   
 ⟨*proof*⟩

#### 67.14.1 Monotonicity of multiset union

**lemma** *mult1-union*:  $(B, D) \in \text{mult1 } r \implies (C + B, C + D) \in \text{mult1 } r$   
 ⟨*proof*⟩

**lemma** *union-le-mono2*:  $B < D \implies C + B < C + (D :: 'a :: \text{preorder multiset})$   
 ⟨*proof*⟩

**lemma** *union-le-mono1*:  $B < D \implies B + C < D + (C :: 'a :: \text{preorder multiset})$   
 ⟨*proof*⟩

**lemma** *union-less-mono*:

**fixes**  $A B C D :: 'a :: \text{preorder multiset}$

**shows**  $A < C \implies B < D \implies A + B < C + D$

⟨*proof*⟩

**instantiation** *multiset* ::  $(\text{preorder}) \text{ ordered-ab-semigroup-add}$

**begin**

**instance**

⟨*proof*⟩

**end**

#### 67.14.2 Termination proofs with multiset orders

**lemma** *multi-member-skip*:  $x \in\# XS \implies x \in\# \{\# y \# \} + XS$

**and** *multi-member-this*:  $x \in\# \{\# x \# \} + XS$

**and** *multi-member-last*:  $x \in\# \{\# x \# \}$

⟨*proof*⟩

**definition** *ms-strict* = *mult pair-less*

**definition** *ms-weak* = *ms-strict*  $\cup$  *Id*

**lemma** *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)

*<proof>*

**lemma** *smsI*:

$(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z + B) \in \text{ms-strict}$

*<proof>*

**lemma** *wmsI*:

$(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \vee A = \{\#\} \wedge B = \{\#\}$

$\implies (Z + A, Z + B) \in \text{ms-weak}$

*<proof>*

**inductive** *pw-leq*

**where**

*pw-leq-empty*: *pw-leq*  $\{\#\}$   $\{\#\}$

| *pw-leq-step*:  $\llbracket (x,y) \in \text{pair-leq}; \text{pw-leq } X \ Y \rrbracket \implies \text{pw-leq } (\{\#x\# \} + X) (\{\#y\# \} + Y)$

**lemma** *pw-leq-lstep*:

$(x, y) \in \text{pair-leq} \implies \text{pw-leq } \{\#x\# \} \{\#y\# \}$

*<proof>*

**lemma** *pw-leq-split*:

**assumes** *pw-leq*  $X \ Y$

**shows**  $\exists A \ B \ Z. X = A + Z \wedge Y = B + Z \wedge ((\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \vee (B = \{\#\} \wedge A = \{\#\}))$

*<proof>*

**lemma**

**assumes** *pwleq*: *pw-leq*  $Z \ Z'$

**shows** *ms-strictI*:  $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z' + B) \in \text{ms-strict}$

**and** *ms-weakI1*:  $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z' + B) \in \text{ms-weak}$

**and** *ms-weakI2*:  $(Z + \{\#\}, Z' + \{\#\}) \in \text{ms-weak}$

*<proof>*

**lemma** *empty-neutral*:  $\{\#\} + x = x \ x + \{\#\} = x$

**and** *nonempty-plus*:  $\{\# \ x \ \#\} + rs \neq \{\#\}$

**and** *nonempty-single*:  $\{\# \ x \ \#\} \neq \{\#\}$

*<proof>*

*<ML>*

### 67.15 Legacy theorem bindings

**lemmas** *multi-count-eq = multiset-eq-iff* [*symmetric*]

**lemma** *union-commute*:  $M + N = N + (M::'a \text{ multiset})$   
 ⟨proof⟩

**lemma** *union-assoc*:  $(M + N) + K = M + (N + (K::'a \text{ multiset}))$   
 ⟨proof⟩

**lemma** *union-lcomm*:  $M + (N + K) = N + (M + (K::'a \text{ multiset}))$   
 ⟨proof⟩

**lemmas** *union-ac = union-assoc union-commute union-lcomm add-mset-commute*

**lemma** *union-right-cancel*:  $M + K = N + K \longleftrightarrow M = (N::'a \text{ multiset})$   
 ⟨proof⟩

**lemma** *union-left-cancel*:  $K + M = K + N \longleftrightarrow M = (N::'a \text{ multiset})$   
 ⟨proof⟩

**lemma** *multi-union-self-other-eq*:  $(A::'a \text{ multiset}) + X = A + Y \Longrightarrow X = Y$   
 ⟨proof⟩

**lemma** *mset-subset-trans*:  $(M::'a \text{ multiset}) \subset\# K \Longrightarrow K \subset\# N \Longrightarrow M \subset\# N$   
 ⟨proof⟩

**lemma** *multiset-inter-commute*:  $A \cap\# B = B \cap\# A$   
 ⟨proof⟩

**lemma** *multiset-inter-assoc*:  $A \cap\# (B \cap\# C) = A \cap\# B \cap\# C$   
 ⟨proof⟩

**lemma** *multiset-inter-left-commute*:  $A \cap\# (B \cap\# C) = B \cap\# (A \cap\# C)$   
 ⟨proof⟩

**lemmas** *multiset-inter-ac =*  
*multiset-inter-commute*  
*multiset-inter-assoc*  
*multiset-inter-left-commute*

**lemma** *mset-le-not-refl*:  $\neg M < (M::'a::\text{preorder multiset})$   
 ⟨proof⟩

**lemma** *mset-le-trans*:  $K < M \Longrightarrow M < N \Longrightarrow K < (N::'a::\text{preorder multiset})$   
 ⟨proof⟩

**lemma** *mset-le-not-sym*:  $M < N \Longrightarrow \neg N < (M::'a::\text{preorder multiset})$   
 ⟨proof⟩

**lemma** *mset-le-asym*:  $M < N \Longrightarrow (\neg P \Longrightarrow N < (M::'a::\text{preorder multiset})) \Longrightarrow P$   
 ⟨proof⟩

⟨ML⟩

## 67.16 Naive implementation using lists

**code-datatype** *mset*

**lemma** [*code*]:  $\{\#\} = \text{mset } []$   
 ⟨*proof*⟩

**lemma** [*code*]:  $\text{add-mset } x (\text{mset } xs) = \text{mset } (x \# xs)$   
 ⟨*proof*⟩

**lemma** [*code*]:  $\text{Multiset.is-empty } (\text{mset } xs) \longleftrightarrow \text{List.null } xs$   
 ⟨*proof*⟩

**lemma** *union-code* [*code*]:  $\text{mset } xs + \text{mset } ys = \text{mset } (xs @ ys)$   
 ⟨*proof*⟩

**lemma** [*code*]:  $\text{image-mset } f (\text{mset } xs) = \text{mset } (\text{map } f xs)$   
 ⟨*proof*⟩

**lemma** [*code*]:  $\text{filter-mset } f (\text{mset } xs) = \text{mset } (\text{filter } f xs)$   
 ⟨*proof*⟩

**lemma** [*code*]:  $\text{mset } xs - \text{mset } ys = \text{mset } (\text{fold } \text{remove1 } ys xs)$   
 ⟨*proof*⟩

**lemma** [*code*]:  
 $\text{mset } xs \cap \# \text{mset } ys =$   
 $\text{mset } (\text{snd } (\text{fold } (\lambda x (ys, zs).$   
 $\text{if } x \in \text{set } ys \text{ then } (\text{remove1 } x ys, x \# zs) \text{ else } (ys, zs)) xs (ys, [])))$   
 ⟨*proof*⟩

**lemma** [*code*]:  
 $\text{mset } xs \cup \# \text{mset } ys =$   
 $\text{mset } (\text{case-prod } \text{append } (\text{fold } (\lambda x (ys, zs). (\text{remove1 } x ys, x \# zs)) xs (ys, [])))$   
 ⟨*proof*⟩

**declare** *in-multiset-in-set* [*code-unfold*]

**lemma** [*code*]:  $\text{count } (\text{mset } xs) x = \text{fold } (\lambda y. \text{if } x = y \text{ then } \text{Suc } \text{else } \text{id}) xs 0$   
 ⟨*proof*⟩

**declare** *set-mset-mset* [*code*]

**declare** *sorted-list-of-multiset-mset* [*code*]

**lemma** [*code*]: — not very efficient, but representation-ignorant!

```
mset-set A = mset (sorted-list-of-set A)
⟨proof⟩
```

```
declare size-mset [code]
```

```
fun subset-eq-mset-impl :: 'a list ⇒ 'a list ⇒ bool option where
  subset-eq-mset-impl [] ys = Some (ys ≠ [])
| subset-eq-mset-impl (Cons x xs) ys = (case List.extract ((=) x) ys of
  None ⇒ None
  | Some (ys1,-,ys2) ⇒ subset-eq-mset-impl xs (ys1 @ ys2))
```

```
lemma subset-eq-mset-impl: (subset-eq-mset-impl xs ys = None ⟷ ¬ mset xs
  ⊆# mset ys) ∧
  (subset-eq-mset-impl xs ys = Some True ⟷ mset xs ⊆# mset ys) ∧
  (subset-eq-mset-impl xs ys = Some False ⟷ mset xs = mset ys)
⟨proof⟩
```

```
lemma [code]: mset xs ⊆# mset ys ⟷ subset-eq-mset-impl xs ys ≠ None
⟨proof⟩
```

```
lemma [code]: mset xs ⊆# mset ys ⟷ subset-eq-mset-impl xs ys = Some True
⟨proof⟩
```

```
instantiation multiset :: (equal) equal
begin
```

```
definition
```

```
[code del]: HOL.equal A (B :: 'a multiset) ⟷ A = B
```

```
lemma [code]: HOL.equal (mset xs) (mset ys) ⟷ subset-eq-mset-impl xs ys =
  Some False
⟨proof⟩
```

```
instance
⟨proof⟩
```

```
end
```

```
declare sum-mset-sum-list [code]
```

```
lemma [code]: prod-mset (mset xs) = fold times xs 1
⟨proof⟩
```

Exercise for the casual reader: add implementations for  $(\leq)$  and  $(<)$  (multiset order).

Quickcheck generators

```
context
```

```
  includes term-syntax
```

```
begin
```



**definition**

$msetify :: 'a::typerep\ list \times (unit \Rightarrow Code-Evaluation.term)$   
 $\Rightarrow 'a\ multiset \times (unit \Rightarrow Code-Evaluation.term)$  **where**  
 $[code-unfold]: msetify\ xs = Code-Evaluation.valtermify\ mset\ \{\cdot\}\ xs$

**end**

**instantiation**  $multiset :: (random)\ random$   
**begin**

**context**

**includes**  $state-combinator-syntax$   
**begin**

**definition**

$Quickcheck-Random.random\ i = Quickcheck-Random.random\ i \circ \rightarrow (\lambda xs. Pair\ (msetify\ xs))$

**instance**  $\langle proof \rangle$

**end****end**

**instantiation**  $multiset :: (full-exhaustive)\ full-exhaustive$   
**begin**

**definition**  $full-exhaustive-multiset :: ('a\ multiset \times (unit \Rightarrow term) \Rightarrow (bool \times term\ list)\ option) \Rightarrow natural \Rightarrow (bool \times term\ list)\ option$

**where**

$full-exhaustive-multiset\ f\ i = Quickcheck-Exhaustive.full-exhaustive\ (\lambda xs. f\ (msetify\ xs))\ i$

**instance**  $\langle proof \rangle$

**end**

**hide-const**  $(open)\ msetify$

**67.17 BNF setup****definition**  $rel-mset$  **where**

$rel-mset\ R\ X\ Y \longleftrightarrow (\exists xs\ ys. mset\ xs = X \wedge mset\ ys = Y \wedge list-all2\ R\ xs\ ys)$

**lemma**  $mset-zip-take-Cons-drop-twice$ :

**assumes**  $length\ xs = length\ ys\ j \leq length\ xs$

**shows**  $mset\ (zip\ (take\ j\ xs\ @\ x\ \# drop\ j\ xs)\ (take\ j\ ys\ @\ y\ \# drop\ j\ ys)) =$   
 $add-mset\ (x,y)\ (mset\ (zip\ xs\ ys))$

$\langle proof \rangle$

**lemma** *ex-mset-zip-left*:

**assumes** *length xs = length ys mset xs' = mset xs*

**shows**  $\exists ys'. \text{length } ys' = \text{length } xs' \wedge \text{mset } (\text{zip } xs' \text{ } ys') = \text{mset } (\text{zip } xs \text{ } ys)$

*<proof>*

**lemma** *list-all2-reorder-left-invariance*:

**assumes** *rel: list-all2 R xs ys and ms-x: mset xs' = mset xs*

**shows**  $\exists ys'. \text{list-all2 } R \text{ } xs' \text{ } ys' \wedge \text{mset } ys' = \text{mset } ys$

*<proof>*

**lemma** *ex-mset*:  $\exists xs. \text{mset } xs = X$

*<proof>*

**inductive** *pred-mset* ::  $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$

**where**

*pred-mset P {#}*

|  $\llbracket P \text{ } a; \text{pred-mset } P \text{ } M \rrbracket \Longrightarrow \text{pred-mset } P \text{ } (\text{add-mset } a \text{ } M)$

**lemma** *pred-mset-iff*: — TODO: alias for *Multiset.Ball*

$\langle \text{pred-mset } P \text{ } M \longleftrightarrow \text{Multiset.Ball } M \text{ } P \rangle$  (**is**  $\langle ?P \longleftrightarrow ?Q \rangle$ )

*<proof>*

**bnf** *'a multiset*

*map: image-mset*

*sets: set-mset*

*bd: natLeq*

*wits: {#}*

*rel: rel-mset*

*pred: pred-mset*

*<proof>*

**inductive** *rel-mset'* ::  $\langle ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'b \text{ multiset} \Rightarrow \text{bool} \rangle$

**where**

*Zero[intro]: rel-mset' R {#} {#}*

| *Plus[intro]:  $\llbracket R \text{ } a \text{ } b; \text{rel-mset}' R \text{ } M \text{ } N \rrbracket \Longrightarrow \text{rel-mset}' R \text{ } (\text{add-mset } a \text{ } M) \text{ } (\text{add-mset } b \text{ } N)$*

**lemma** *rel-mset-Zero*: *rel-mset R {#} {#}*

*<proof>*

**declare** *multiset.count[simp]*

**declare** *count-Abs-multiset[simp]*

**declare** *multiset.count-inverse[simp]*

**lemma** *rel-mset-Plus*:

**assumes** *ab: R a b*

**and** *MN: rel-mset R M N*

**shows** *rel-mset R (add-mset a M) (add-mset b N)*

*<proof>*

**lemma** *rel-mset'-imp-rel-mset*:  $rel\text{-}mset' R M N \implies rel\text{-}mset R M N$   
*<proof>*

**lemma** *rel-mset-size*:  $rel\text{-}mset R M N \implies size M = size N$   
*<proof>*

**lemma** *rel-mset-Zero-iff* [*simp*]:  
**shows**  $rel\text{-}mset\ rel\ \{\#\}\ Y \longleftrightarrow Y = \{\#\}$  **and**  $rel\text{-}mset\ rel\ X\ \{\#\}\ \longleftrightarrow X = \{\#\}$   
*<proof>*

**lemma** *multiset-induct2*[*case-names empty addL addR*]:  
**assumes** *empty*:  $P\ \{\#\}\ \{\#\}$   
**and** *addL*:  $\bigwedge a\ M\ N. P\ M\ N \implies P\ (add\text{-}mset\ a\ M)\ N$   
**and** *addR*:  $\bigwedge a\ M\ N. P\ M\ N \implies P\ M\ (add\text{-}mset\ a\ N)$   
**shows**  $P\ M\ N$   
*<proof>*

**lemma** *multiset-induct2-size*[*consumes 1, case-names empty add*]:  
**assumes** *c*:  $size\ M = size\ N$   
**and** *empty*:  $P\ \{\#\}\ \{\#\}$   
**and** *add*:  $\bigwedge a\ b\ M\ N\ a\ b. P\ M\ N \implies P\ (add\text{-}mset\ a\ M)\ (add\text{-}mset\ b\ N)$   
**shows**  $P\ M\ N$   
*<proof>*

**lemma** *msed-map-invL*:  
**assumes** *image-mset f*  $(add\text{-}mset\ a\ M) = N$   
**shows**  $\exists N1. N = add\text{-}mset\ (f\ a)\ N1 \wedge image\text{-}mset\ f\ M = N1$   
*<proof>*

**lemma** *msed-map-invR*:  
**assumes** *image-mset f M*  $= add\text{-}mset\ b\ N$   
**shows**  $\exists M1\ a. M = add\text{-}mset\ a\ M1 \wedge f\ a = b \wedge image\text{-}mset\ f\ M1 = N$   
*<proof>*

**lemma** *msed-rel-invL*:  
**assumes** *rel-mset R*  $(add\text{-}mset\ a\ M)\ N$   
**shows**  $\exists N1\ b. N = add\text{-}mset\ b\ N1 \wedge R\ a\ b \wedge rel\text{-}mset\ R\ M\ N1$   
*<proof>*

**lemma** *msed-rel-invR*:  
**assumes** *rel-mset R M*  $(add\text{-}mset\ b\ N)$   
**shows**  $\exists M1\ a. M = add\text{-}mset\ a\ M1 \wedge R\ a\ b \wedge rel\text{-}mset\ R\ M1\ N$   
*<proof>*

**lemma** *rel-mset-imp-rel-mset'*:  
**assumes** *rel-mset R M N*  
**shows**  $rel\text{-}mset'\ R\ M\ N$

*<proof>*

**lemma** *rel-mset-rel-mset'*:  $rel\text{-}mset\ R\ M\ N = rel\text{-}mset'\ R\ M\ N$

*<proof>*

The main end product for *rel-mset*: inductive characterization:

**lemmas** *rel-mset-induct*[*case-names empty add, induct pred: rel-mset*] =  
*rel-mset'.induct*[*unfolded rel-mset-rel-mset'*[*symmetric*]]

### 67.18 Size setup

**lemma** *size-multiset-o-map*:  $size\text{-}multiset\ g \circ image\text{-}mset\ f = size\text{-}multiset\ (g \circ f)$

*<proof>*

*<ML>*

### 67.19 Lemmas about Size

**lemma** *size-mset-SucE*:  $size\ A = Suc\ n \implies (\bigwedge a\ B.\ A = \{\#a\#\} + B \implies size\ B = n \implies P) \implies P$

*<proof>*

**lemma** *size-Suc-Diff1*:  $x \in\# M \implies Suc\ (size\ (M - \{\#x\#\})) = size\ M$

*<proof>*

**lemma** *size-Diff-singleton*:  $x \in\# M \implies size\ (M - \{\#x\#\}) = size\ M - 1$

*<proof>*

**lemma** *size-Diff-singleton-if*:  $size\ (A - \{\#x\#\}) = (if\ x \in\# A\ then\ size\ A - 1\ else\ size\ A)$

*<proof>*

**lemma** *size-Un-Int*:  $size\ A + size\ B = size\ (A \cup\# B) + size\ (A \cap\# B)$

*<proof>*

**lemma** *size-Un-disjoint*:  $A \cap\# B = \{\#\} \implies size\ (A \cup\# B) = size\ A + size\ B$

*<proof>*

**lemma** *size-Diff-subset-Int*:  $size\ (M - M') = size\ M - size\ (M \cap\# M')$

*<proof>*

**lemma** *diff-size-le-size-Diff*:  $size\ (M :: -\ multiset) - size\ M' \leq size\ (M - M')$

*<proof>*

**lemma** *size-Diff1-less*:  $x \in\# M \implies size\ (M - \{\#x\#\}) < size\ M$

*<proof>*

**lemma** *size-Diff2-less*:  $x \in\# M \implies y \in\# M \implies size\ (M - \{\#x\#\} - \{\#y\#\}) < size\ M$

*<proof>*

**lemma** *size-Diff1-le*:  $size (M - \{x\}) \leq size M$   
 ⟨proof⟩

**lemma** *size-psubset*:  $M \subseteq\# M' \implies size M < size M' \implies M \subset\# M'$   
 ⟨proof⟩

**lifting-update** *multiset.lifting*  
**lifting-forget** *multiset.lifting*

**hide-const** (**open**) *wcount*

**end**

## 68 More Theorems about the Multiset Order

**theory** *Multiset-Order*  
**imports** *Multiset*  
**begin**

### 68.1 Alternative Characterizations

#### 68.1.1 The Dershowitz–Manna Ordering

**definition** *multp<sub>DM</sub>* **where**  
 $multp_{DM} r M N \longleftrightarrow$   
 $(\exists X Y. X \neq \{x\} \wedge X \subseteq\# N \wedge M = (N - X) + Y \wedge (\forall k. k \in\# Y \longrightarrow (\exists a. a \in\# X \wedge r k a)))$

**lemma** *multp<sub>DM</sub>-imp-multp*:  
 $multp_{DM} r M N \implies multp r M N$   
 ⟨proof⟩

#### 68.1.2 The Huet–Oppen Ordering

**definition** *multp<sub>HO</sub>* **where**  
 $multp_{HO} r M N \longleftrightarrow M \neq N \wedge (\forall y. count N y < count M y \longrightarrow (\exists x. r y x \wedge count M x < count N x))$

**lemma** *multp-imp-multp<sub>HO</sub>*:  
**assumes** *asyp* *r* **and** *transp* *r*  
**shows**  $multp r M N \implies multp_{HO} r M N$   
 ⟨proof⟩

**lemma** *multp<sub>HO</sub>-imp-multp<sub>DM</sub>*:  $multp_{HO} r M N \implies multp_{DM} r M N$   
 ⟨proof⟩

**lemma** *multp-eq-multp<sub>DM</sub>*:  $asyp r \implies transp r \implies multp r = multp_{DM} r$   
 ⟨proof⟩

**lemma** *multp-eq-multp<sub>HO</sub>*:  $asympt r \implies transp r \implies multp r = multp_{HO} r$   
 ⟨proof⟩

**lemma** *multp<sub>DM</sub>-plus-plusI[simp]*:  
**assumes**  $multp_{DM} R M1 M2$   
**shows**  $multp_{DM} R (M + M1) (M + M2)$   
 ⟨proof⟩

**lemma** *multp<sub>HO</sub>-plus-plus[simp]*:  $multp_{HO} R (M + M1) (M + M2) \longleftrightarrow multp_{HO} R M1 M2$   
 ⟨proof⟩

**lemma** *strict-subset-implies-multp<sub>DM</sub>*:  $A \subset\# B \implies multp_{DM} r A B$   
 ⟨proof⟩

**lemma** *strict-subset-implies-multp<sub>HO</sub>*:  $A \subset\# B \implies multp_{HO} r A B$   
 ⟨proof⟩

**lemma** *multp<sub>HO</sub>-implies-one-step-strong*:  
**assumes**  $multp_{HO} R A B$   
**defines**  $J \equiv B - A$  **and**  $K \equiv A - B$   
**shows**  $J \neq \{\#\}$  **and**  $\forall k \in\# K. \exists x \in\# J. R k x$   
 ⟨proof⟩

**lemma** *multp<sub>HO</sub>-minus-inter-minus-inter-iff*:  
**fixes**  $M1 M2 :: \text{multiset}$   
**shows**  $multp_{HO} R (M1 - M2) (M2 - M1) \longleftrightarrow multp_{HO} R M1 M2$   
 ⟨proof⟩

**lemma** *multp<sub>HO</sub>-iff-set-mset-less<sub>HO</sub>-set-mset*:  
 $multp_{HO} R M1 M2 \longleftrightarrow (set\text{-}mset (M1 - M2) \neq set\text{-}mset (M2 - M1) \wedge$   
 $(\forall y \in\# M1 - M2. (\exists x \in\# M2 - M1. R y x)))$   
 ⟨proof⟩

### 68.1.3 Monotonicity

**lemma** *multp<sub>DM</sub>-mono-strong*:  
 $multp_{DM} R M1 M2 \implies (\bigwedge x y. x \in\# M1 \implies y \in\# M2 \implies R x y \implies S x y)$   
 $\implies multp_{DM} S M1 M2$   
 ⟨proof⟩

**lemma** *multp<sub>HO</sub>-mono-strong*:  
 $multp_{HO} R M1 M2 \implies (\bigwedge x y. x \in\# M1 \implies y \in\# M2 \implies R x y \implies S x y)$   
 $\implies multp_{HO} S M1 M2$   
 ⟨proof⟩

#### 68.1.4 Properties of Orders

**Asymmetry** The following lemma is a negative result stating that asymmetry of an arbitrary binary relation cannot be simply lifted to  $multp_{HO}$ . It suffices to have four distinct values to build a counterexample.

**lemma** *asympt-not-liftable-to-multp<sub>HO</sub>*:

**fixes**  $a\ b\ c\ d :: 'a$

**assumes** *distinct*  $[a, b, c, d]$

**shows**  $\neg (\forall (R :: 'a \Rightarrow 'a \Rightarrow \text{bool}). \text{asympt } R \longrightarrow \text{asympt } (\text{multp}_{HO} R))$

*<proof>*

However, if the binary relation is both asymmetric and transitive, then  $multp_{HO}$  is also asymmetric.

**lemma** *asympt-on-multp<sub>HO</sub>*:

**assumes** *asympt-on*  $A\ R$  **and** *transp-on*  $A\ R$  **and**

$B\text{-sub-}A: \bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$

**shows** *asympt-on*  $B\ (\text{multp}_{HO} R)$

*<proof>*

**lemma** *asympt-multp<sub>HO</sub>*:

**assumes** *asympt*  $R$  **and** *transp*  $R$

**shows** *asympt*  $(\text{multp}_{HO} R)$

*<proof>*

**Irreflexivity lemma** *irreflp-on-multp<sub>HO</sub>[simp]*: *irreflp-on*  $B\ (\text{multp}_{HO} R)$

*<proof>*

**Transitivity lemma** *transp-on-multp<sub>HO</sub>*:

**assumes** *asympt-on*  $A\ R$  **and** *transp-on*  $A\ R$  **and**  $B\text{-sub-}A: \bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$

**shows** *transp-on*  $B\ (\text{multp}_{HO} R)$

*<proof>*

**lemma** *transp-multp<sub>HO</sub>*:

**assumes** *asympt*  $R$  **and** *transp*  $R$

**shows** *transp*  $(\text{multp}_{HO} R)$

*<proof>*

**Totality lemma** *totalp-on-multp<sub>DM</sub>*:

*totalp-on*  $A\ R \implies (\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A) \implies \text{totalp-on } B\ (\text{multp}_{DM} R)$

*<proof>*

**lemma** *totalp-multp<sub>DM</sub>*: *totalp*  $R \implies \text{totalp } (\text{multp}_{DM} R)$

*<proof>*

**lemma** *totalp-on-multp<sub>HO</sub>*:

*totalp-on*  $A\ R \implies (\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A) \implies \text{totalp-on } B\ (\text{multp}_{HO} R)$

*<proof>*

*<proof>*

**lemma** *totalp-multp<sub>HO</sub>*:  $totalp\ R \implies totalp\ (multp_{HO}\ R)$   
*<proof>*

**Type Classes** **context** *preorder*  
**begin**

**lemma** *order-mult: class.order*  
 $(\lambda M\ N. (M, N) \in mult\ \{(x, y). x < y\} \vee M = N)$   
 $(\lambda M\ N. (M, N) \in mult\ \{(x, y). x < y\})$   
*(is class.order ?le ?less)*  
*<proof>*

The Dershowitz–Manna ordering:

**definition** *less-multiset<sub>DM</sub>* **where**  
 $less-multiset_{DM}\ M\ N \longleftrightarrow$   
 $(\exists X\ Y. X \neq \{\#\} \wedge X \subseteq\# N \wedge M = (N - X) + Y \wedge (\forall k. k \in\# Y \longrightarrow (\exists a. a \in\# X \wedge k < a)))$

The Huet–Oppen ordering:

**definition** *less-multiset<sub>HO</sub>* **where**  
 $less-multiset_{HO}\ M\ N \longleftrightarrow M \neq N \wedge (\forall y. count\ N\ y < count\ M\ y \longrightarrow (\exists x. y < x \wedge count\ M\ x < count\ N\ x))$

**lemma** *mult-imp-less-multiset<sub>HO</sub>*:  
 $(M, N) \in mult\ \{(x, y). x < y\} \implies less-multiset_{HO}\ M\ N$   
*<proof>*

**lemma** *less-multiset<sub>DM</sub>-imp-mult*:  
 $less-multiset_{DM}\ M\ N \implies (M, N) \in mult\ \{(x, y). x < y\}$   
*<proof>*

**lemma** *less-multiset<sub>HO</sub>-imp-less-multiset<sub>DM</sub>*:  $less-multiset_{HO}\ M\ N \implies less-multiset_{DM}\ M\ N$   
*<proof>*

**lemma** *mult-less-multiset<sub>DM</sub>*:  $(M, N) \in mult\ \{(x, y). x < y\} \longleftrightarrow less-multiset_{DM}\ M\ N$   
*<proof>*

**lemma** *mult-less-multiset<sub>HO</sub>*:  $(M, N) \in mult\ \{(x, y). x < y\} \longleftrightarrow less-multiset_{HO}\ M\ N$   
*<proof>*

**lemmas**  $mult_{DM} = mult-less-multiset_{DM}[unfolded\ less-multiset_{DM}-def]$

**lemmas**  $mult_{HO} = mult-less-multiset_{HO}[unfolded\ less-multiset_{HO}-def]$

**end**



**lemma** *less-multiset-less-multiset<sub>HO</sub>*:  $M < N \longleftrightarrow \text{less-multiset}_{HO} M N$   
 ⟨proof⟩

**lemma** *less-multiset<sub>DM</sub>*:  
 $M < N \longleftrightarrow (\exists X Y. X \neq \{\#\} \wedge X \subseteq\# N \wedge M = N - X + Y \wedge (\forall k. k \in\# Y \longrightarrow (\exists a. a \in\# X \wedge k < a)))$   
 ⟨proof⟩

**lemma** *less-multiset<sub>HO</sub>*:  
 $M < N \longleftrightarrow M \neq N \wedge (\forall y. \text{count } N y < \text{count } M y \longrightarrow (\exists x>y. \text{count } M x < \text{count } N x))$   
 ⟨proof⟩

**lemma** *subset-eq-imp-le-multiset*:  
**shows**  $M \subseteq\# N \Longrightarrow M \leq N$   
 ⟨proof⟩

**lemma** *le-multiset-right-total*:  $M < \text{add-mset } x M$   
 ⟨proof⟩

**lemma** *less-eq-multiset-empty-left[simp]*:  $\{\#\} \leq M$   
 ⟨proof⟩

**lemma** *ex-gt-imp-less-multiset*:  $(\exists y. y \in\# N \wedge (\forall x. x \in\# M \longrightarrow x < y)) \Longrightarrow M < N$   
 ⟨proof⟩

**lemma** *less-eq-multiset-empty-right[simp]*:  $M \neq \{\#\} \Longrightarrow \neg M \leq \{\#\}$   
 ⟨proof⟩

**lemma** *le-multiset-empty-left[simp]*:  $M \neq \{\#\} \Longrightarrow \{\#\} < M$   
 ⟨proof⟩

**lemma** *le-multiset-empty-right[simp]*:  $\neg M < \{\#\}$   
 ⟨proof⟩

**lemma** *union-le-diff-plus*:  $P \subseteq\# M \Longrightarrow N < P \Longrightarrow M - P + N < M$   
 ⟨proof⟩

**instantiation** *multiset* :: (preorder) ordered-ab-semigroup-monoid-add-imp-le  
**begin**

**lemma** *less-eq-multiset<sub>HO</sub>*:  
 $M \leq N \longleftrightarrow (\forall y. \text{count } N y < \text{count } M y \longrightarrow (\exists x. y < x \wedge \text{count } M x < \text{count } N x))$

$N x)$   
 ⟨*proof*⟩

**instance** ⟨*proof*⟩

**lemma**

**fixes**  $M N :: 'a \text{ multiset}$   
**shows** *less-eq-multiset-plus-left*:  $N \leq (M + N)$   
**and** *less-eq-multiset-plus-right*:  $M \leq (M + N)$   
 ⟨*proof*⟩

**lemma**

**fixes**  $M N :: 'a \text{ multiset}$   
**shows** *le-multiset-plus-left-nonempty*:  $M \neq \{\#\} \implies N < M + N$   
**and** *le-multiset-plus-right-nonempty*:  $N \neq \{\#\} \implies M < M + N$   
 ⟨*proof*⟩

**end**

**lemma** *all-lt-Max-imp-lt-mset*:  $N \neq \{\#\} \implies (\forall a \in\# M. a < \text{Max} (\text{set-mset } N)) \implies M < N$   
 ⟨*proof*⟩

**lemma** *lt-imp-ex-count-lt*:  $M < N \implies \exists y. \text{count } M y < \text{count } N y$   
 ⟨*proof*⟩

**lemma** *subset-imp-less-mset*:  $A \subset\# B \implies A < B$   
 ⟨*proof*⟩

**lemma** *image-mset-strict-mono*:

**assumes** *mono-f*:  $\forall x \in \text{set-mset } M. \forall y \in \text{set-mset } N. x < y \longrightarrow f x < f y$   
**and** *less*:  $M < N$   
**shows** *image-mset f M < image-mset f N*  
 ⟨*proof*⟩

**lemma** *image-mset-mono*:

**assumes** *mono-f*:  $\forall x \in \text{set-mset } M. \forall y \in \text{set-mset } N. x < y \longrightarrow f x < f y$   
**and** *less*:  $M \leq N$   
**shows** *image-mset f M \leq image-mset f N*  
 ⟨*proof*⟩

**lemma** *mset-lt-single-right-iff[simp]*:  $M < \{\#y\# \} \longleftrightarrow (\forall x \in\# M. x < y)$  **for**  $y :: 'a::\text{linorder}$   
 ⟨*proof*⟩

**lemma** *mset-le-single-right-iff[simp]*:

$M \leq \{\#y\# \} \longleftrightarrow M = \{\#y\# \} \vee (\forall x \in\# M. x < y)$  **for**  $y :: 'a::\text{linorder}$   
 ⟨*proof*⟩

### 68.1.5 Simplifications

**lemma** *multp<sub>HO</sub>-repeat-mset-repeat-mset[simp]*:

**assumes**  $n \neq 0$

**shows**  $\text{multp}_{HO} R (\text{repeat-mset } n A) (\text{repeat-mset } n B) \longleftrightarrow \text{multp}_{HO} R A B$

*<proof>*

**lemma** *multp<sub>HO</sub>-double-double[simp]*:  $\text{multp}_{HO} R (A + A) (B + B) \longleftrightarrow \text{multp}_{HO} R A B$

*<proof>*

### 68.2 Simprocs

**lemma** *mset-le-add-iff1*:

$j \leq (i::\text{nat}) \implies (\text{repeat-mset } i u + m \leq \text{repeat-mset } j u + n) = (\text{repeat-mset } (i-j) u + m \leq n)$

*<proof>*

**lemma** *mset-le-add-iff2*:

$i \leq (j::\text{nat}) \implies (\text{repeat-mset } i u + m \leq \text{repeat-mset } j u + n) = (m \leq \text{repeat-mset } (j-i) u + n)$

*<proof>*

*<ML>*

### 68.3 Additional facts and instantiations

**lemma** *ex-gt-count-imp-le-multiset*:

$(\forall y :: 'a :: \text{order}. y \in\# M + N \longrightarrow y \leq x) \implies \text{count } M x < \text{count } N x \implies M < N$

*<proof>*

**lemma** *mset-lt-single-iff[iff]*:  $\{\#x\# \} < \{\#y\# \} \longleftrightarrow x < y$

*<proof>*

**lemma** *mset-le-single-iff[iff]*:  $\{\#x\# \} \leq \{\#y\# \} \longleftrightarrow x \leq y$  **for**  $x y :: 'a :: \text{order}$

*<proof>*

**instance** *multiset* :: *(linorder) linordered-cancel-ab-semigroup-add*

*<proof>*

**lemma** *less-eq-multiset-total*:  $\neg M \leq N \implies N \leq M$  **for**  $M N :: 'a :: \text{linorder multiset}$

*<proof>*

**instantiation** *multiset* :: *(wellorder) wellorder*

**begin**

**lemma** *wf-less-multiset*:  $\text{wf } \{(M :: 'a \text{ multiset}, N). M < N\}$

*<proof>*

**instance**

*<proof>*

**end**

**instantiation** *multiset* :: (*preorder*) *order-bot*

**begin**

**definition** *bot-multiset* :: 'a *multiset* **where** *bot-multiset* = {#}

**instance** *<proof>*

**end**

**instance** *multiset* :: (*preorder*) *no-top*

*<proof>*

**instance** *multiset* :: (*preorder*) *ordered-cancel-comm-monoid-add*

*<proof>*

**instantiation** *multiset* :: (*linorder*) *distrib-lattice*

**begin**

**definition** *inf-multiset* :: 'a *multiset*  $\Rightarrow$  'a *multiset*  $\Rightarrow$  'a *multiset* **where**

*inf-multiset* *A B* = (*if* *A* < *B* *then* *A* *else* *B*)

**definition** *sup-multiset* :: 'a *multiset*  $\Rightarrow$  'a *multiset*  $\Rightarrow$  'a *multiset* **where**

*sup-multiset* *A B* = (*if* *B* > *A* *then* *B* *else* *A*)

**instance**

*<proof>*

**end**

**lemma** *add-mset-lt-left-lt*:  $a < b \Longrightarrow \text{add-mset } a \ A < \text{add-mset } b \ A$

*<proof>*

**lemma** *add-mset-le-left-le*:  $a \leq b \Longrightarrow \text{add-mset } a \ A \leq \text{add-mset } b \ A$  **for**  $a :: 'a ::$

*linorder*

*<proof>*

**lemma** *add-mset-lt-right-lt*:  $A < B \Longrightarrow \text{add-mset } a \ A < \text{add-mset } a \ B$

*<proof>*

**lemma** *add-mset-le-right-le*:  $A \leq B \Longrightarrow \text{add-mset } a \ A \leq \text{add-mset } a \ B$

*<proof>*

**lemma** *add-mset-lt-lt-lt*:

**assumes**  $a\text{-lt-}b: a < b$  **and**  $A\text{-le-}B: A < B$   
**shows**  $\text{add-mset } a \ A < \text{add-mset } b \ B$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{add-mset-lt-lt-le}: a < b \implies A \leq B \implies \text{add-mset } a \ A < \text{add-mset } b \ B$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{add-mset-lt-le-lt}: a \leq b \implies A < B \implies \text{add-mset } a \ A < \text{add-mset } b \ B$  **for**  
 $a :: 'a :: \text{linorder}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{add-mset-le-le-le}:$   
**fixes**  $a :: 'a :: \text{linorder}$   
**assumes**  $a\text{-le-}b: a \leq b$  **and**  $A\text{-le-}B: A \leq B$   
**shows**  $\text{add-mset } a \ A \leq \text{add-mset } b \ B$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{Max-lt-imp-lt-mset}:$   
**assumes**  $n\text{-nemp}: N \neq \{\#\}$  **and**  $\text{max}: \text{Max-mset } M < \text{Max-mset } N$  (**is**  $?max\text{-}M < ?max\text{-}N$ )  
**shows**  $M < N$   
 $\langle \text{proof} \rangle$

**end**

## 69 Fixed Length Lists

**theory**  $NList$   
**imports**  $Main$   
**begin**

**definition**  $nlists :: \text{nat} \Rightarrow 'a \ \text{set} \Rightarrow 'a \ \text{list set}$   
**where**  $nlists \ n \ A = \{xs. \ \text{size } xs = n \wedge \ \text{set } xs \subseteq A\}$

**lemma**  $nlistsI: \llbracket \text{size } xs = n; \ \text{set } xs \subseteq A \rrbracket \implies xs \in nlists \ n \ A$   
 $\langle \text{proof} \rangle$

These `[simp]` attributes are double-edged. Many proofs in Jinja rely on it but they can degrade performance.

**lemma**  $nlistsE\text{-length} \ [simp]: xs \in nlists \ n \ A \implies \text{size } xs = n$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{in-nlists-UNIV}: xs \in nlists \ k \ UNIV \longleftrightarrow \text{length } xs = k$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{less-lengthI}: \llbracket xs \in nlists \ n \ A; \ p < n \rrbracket \implies p < \text{size } xs$   
 $\langle \text{proof} \rangle$

**lemma**  $nlistsE\text{-set}[simp]: xs \in nlists \ n \ A \implies \text{set } xs \subseteq A$

*<proof>*

**lemma** *nlists-mono*:

**assumes**  $A \subseteq B$  **shows**  $nlists\ n\ A \subseteq nlists\ n\ B$

*<proof>*

**lemma** *nlists-singleton*:  $nlists\ n\ \{a\} = \{\text{replicate}\ n\ a\}$

*<proof>*

**lemma** *nlists-n-0* [*simp*]:  $nlists\ 0\ A = \{\}\}$

*<proof>*

**lemma** *in-nlists-Suc-iff*:  $(xs \in nlists\ (Suc\ n)\ A) = (\exists y \in A. \exists ys \in nlists\ n\ A. xs = y\#\!ys)$

*<proof>*

**lemma** *Cons-in-nlists-Suc* [*iff*]:  $(x\#\!xs \in nlists\ (Suc\ n)\ A) \longleftrightarrow (x \in A \wedge xs \in nlists\ n\ A)$

*<proof>*

**lemma** *nlists-Suc*:  $nlists\ (Suc\ n)\ A = (\bigcup a \in A. (\#\!)\ a\ '\ nlists\ n\ A)$

*<proof>*

**lemma** *nlists-not-empty*:  $A \neq \{\} \implies \exists xs. xs \in nlists\ n\ A$

*<proof>*

**lemma** *nlistsE-nth-in*:  $\llbracket xs \in nlists\ n\ A; i < n \rrbracket \implies xs!\!i \in A$

*<proof>*

**lemma** *nlists-Cons-Suc* [*elim!*]:

$l\#\!xs \in nlists\ n\ A \implies (\bigwedge n'. n = Suc\ n' \implies l \in A \implies xs \in nlists\ n'\ A \implies P) \implies P$

*<proof>*

**lemma** *nlists-appendE* [*elim!*]:

$a\@\!b \in nlists\ n\ A \implies (\bigwedge n1\ n2. n = n1 + n2 \implies a \in nlists\ n1\ A \implies b \in nlists\ n2\ A \implies P) \implies P$

*<proof>*

**lemma** *nlists-update-in-list* [*simp, intro!*]:

$\llbracket xs \in nlists\ n\ A; x \in A \rrbracket \implies xs[\!i := x] \in nlists\ n\ A$

*<proof>*

**lemma** *nlists-appendI* [*intro?*]:

$\llbracket a \in nlists\ n\ A; b \in nlists\ m\ A \rrbracket \implies a\@\!b \in nlists\ (n+m)\ A$

*<proof>*

**lemma** *nlists-append*:

$xs @ ys \in nlists\ k\ A \iff$   
 $k = length(xs @ ys) \wedge xs \in nlists\ (length\ xs)\ A \wedge ys \in nlists\ (length\ ys)\ A$   
 ⟨proof⟩

**lemma** *nlists-map* [*simp*]:  $(map\ f\ xs \in nlists\ (size\ xs)\ A) = (f\ ` set\ xs \subseteq A)$   
 ⟨proof⟩

**lemma** *nlists-replicateI* [*intro*]:  $x \in A \implies replicate\ n\ x \in nlists\ n\ A$   
 ⟨proof⟩

Link to an executable version on lists in List.

**lemma** *nlists-set*[*code*]:  $nlists\ n\ (set\ xs) = set(List.n-lists\ n\ xs)$   
 ⟨proof⟩

end

## 70 Non-negative, non-positive integers and reals

**theory** *Nonpos-Ints*  
**imports** *Complex-Main*  
**begin**

### 70.1 Non-positive integers

The set of non-positive integers on a ring. (in analogy to the set of non-negative integers  $\mathbf{N}$ ) This is useful e.g. for the Gamma function.

**definition** *nonpos-Ints* ( $\langle \mathbf{Z}_{\leq 0} \rangle$ ) **where**  $\mathbf{Z}_{\leq 0} = \{of-int\ n \mid n. n \leq 0\}$

**lemma** *zero-in-nonpos-Ints* [*simp,intro*]:  $0 \in \mathbf{Z}_{\leq 0}$   
 ⟨proof⟩

**lemma** *neg-one-in-nonpos-Ints* [*simp,intro*]:  $-1 \in \mathbf{Z}_{\leq 0}$   
 ⟨proof⟩

**lemma** *neg-numeral-in-nonpos-Ints* [*simp,intro*]:  $-numeral\ n \in \mathbf{Z}_{\leq 0}$   
 ⟨proof⟩

**lemma** *one-notin-nonpos-Ints* [*simp*]:  $(1 :: 'a :: ring-char-0) \notin \mathbf{Z}_{\leq 0}$   
 ⟨proof⟩

**lemma** *numeral-notin-nonpos-Ints* [*simp*]:  $(numeral\ n :: 'a :: ring-char-0) \notin \mathbf{Z}_{\leq 0}$   
 ⟨proof⟩

**lemma** *minus-of-nat-in-nonpos-Ints* [*simp, intro*]:  $- of-nat\ n \in \mathbf{Z}_{\leq 0}$   
 ⟨proof⟩

**lemma** *of-nat-in-nonpos-Ints-iff*:  $(of-nat\ n :: 'a :: \{ring-1,ring-char-0\}) \in \mathbf{Z}_{\leq 0} \iff n = 0$

*<proof>*

**lemma** *nonpos-Ints-of-int*:  $n \leq 0 \implies \text{of-int } n \in \mathbf{Z}_{\leq 0}$   
*<proof>*

**lemma** *nonpos-IntsI*:  
 $x \in \mathbf{Z} \implies x \leq 0 \implies (x :: 'a :: \text{linordered-idom}) \in \mathbf{Z}_{\leq 0}$   
*<proof>*

**lemma** *nonpos-Ints-subset-Ints*:  $\mathbf{Z}_{\leq 0} \subseteq \mathbf{Z}$   
*<proof>*

**lemma** *nonpos-Ints-nonpos* [dest]:  $x \in \mathbf{Z}_{\leq 0} \implies x \leq (0 :: 'a :: \text{linordered-idom})$   
*<proof>*

**lemma** *nonpos-Ints-Int* [dest]:  $x \in \mathbf{Z}_{\leq 0} \implies x \in \mathbf{Z}$   
*<proof>*

**lemma** *nonpos-Ints-cases*:  
**assumes**  $x \in \mathbf{Z}_{\leq 0}$   
**obtains**  $n$  **where**  $x = \text{of-int } n$   $n \leq 0$   
*<proof>*

**lemma** *nonpos-Ints-cases'*:  
**assumes**  $x \in \mathbf{Z}_{\leq 0}$   
**obtains**  $n$  **where**  $x = -\text{of-nat } n$   
*<proof>*

**lemma** *of-real-in-nonpos-Ints-iff*:  $(\text{of-real } x :: 'a :: \text{real-algebra-1}) \in \mathbf{Z}_{\leq 0} \iff x \in \mathbf{Z}_{\leq 0}$   
*<proof>*

**lemma** *nonpos-Ints-altdef*:  $\mathbf{Z}_{\leq 0} = \{n \in \mathbf{Z}. (n :: 'a :: \text{linordered-idom}) \leq 0\}$   
*<proof>*

**lemma** *uminus-in-Nats-iff*:  $-x \in \mathbf{N} \iff x \in \mathbf{Z}_{\leq 0}$   
*<proof>*

**lemma** *uminus-in-nonpos-Ints-iff*:  $-x \in \mathbf{Z}_{\leq 0} \iff x \in \mathbf{N}$   
*<proof>*

**lemma** *nonpos-Ints-mult*:  $x \in \mathbf{Z}_{\leq 0} \implies y \in \mathbf{Z}_{\leq 0} \implies x * y \in \mathbf{N}$   
*<proof>*

**lemma** *Nats-mult-nonpos-Ints*:  $x \in \mathbf{N} \implies y \in \mathbf{Z}_{\leq 0} \implies x * y \in \mathbf{Z}_{\leq 0}$   
*<proof>*

**lemma** *nonpos-Ints-mult-Nats*:  
 $x \in \mathbf{Z}_{\leq 0} \implies y \in \mathbf{N} \implies x * y \in \mathbf{Z}_{\leq 0}$



*<proof>*

**lemma** *nonpos-Ints-add*:

$x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{Z}_{\leq 0} \implies x + y \in \mathbb{Z}_{\leq 0}$   
*<proof>*

**lemma** *nonpos-Ints-diff-Nats*:

$x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{N} \implies x - y \in \mathbb{Z}_{\leq 0}$   
*<proof>*

**lemma** *Nats-diff-nonpos-Ints*:

$x \in \mathbb{N} \implies y \in \mathbb{Z}_{\leq 0} \implies x - y \in \mathbb{N}$   
*<proof>*

**lemma** *plus-of-nat-eq-0-imp*:  $z + \text{of-nat } n = 0 \implies z \in \mathbb{Z}_{\leq 0}$

*<proof>*

## 70.2 Non-negative reals

**definition** *nonneg-Reals* :: 'a::real-algebra-1 set ( $\langle \mathbb{R}_{\geq 0} \rangle$ )

where  $\mathbb{R}_{\geq 0} = \{\text{of-real } r \mid r. r \geq 0\}$

**lemma** *nonneg-Reals-of-real-iff* [simp]:  $\text{of-real } r \in \mathbb{R}_{\geq 0} \iff r \geq 0$

*<proof>*

**lemma** *nonneg-Reals-subset-Reals*:  $\mathbb{R}_{\geq 0} \subseteq \mathbb{R}$

*<proof>*

**lemma** *nonneg-Reals-Real* [dest]:  $x \in \mathbb{R}_{\geq 0} \implies x \in \mathbb{R}$

*<proof>*

**lemma** *nonneg-Reals-of-nat-I* [simp]:  $\text{of-nat } n \in \mathbb{R}_{\geq 0}$

*<proof>*

**lemma** *nonneg-Reals-cases*:

**assumes**  $x \in \mathbb{R}_{\geq 0}$

**obtains**  $r$  **where**  $x = \text{of-real } r$   $r \geq 0$

*<proof>*

**lemma** *nonneg-Reals-zero-I* [simp]:  $0 \in \mathbb{R}_{\geq 0}$

*<proof>*

**lemma** *nonneg-Reals-one-I* [simp]:  $1 \in \mathbb{R}_{\geq 0}$

*<proof>*

**lemma** *nonneg-Reals-minus-one-I* [simp]:  $-1 \notin \mathbb{R}_{\geq 0}$

*<proof>*

**lemma** *nonneg-Reals-numeral-I* [simp]:  $\text{numeral } w \in \mathbb{R}_{\geq 0}$

*<proof>*

**lemma** *nonneg-Reals-minus-numeral-I* [simp]:  $- \text{numeral } w \notin \mathbb{R}_{\geq 0}$   
*<proof>*

**lemma** *nonneg-Reals-add-I* [simp]:  $[[a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0}] \implies a + b \in \mathbb{R}_{\geq 0}$   
*<proof>*

**lemma** *nonneg-Reals-mult-I* [simp]:  $[[a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0}] \implies a * b \in \mathbb{R}_{\geq 0}$   
*<proof>*

**lemma** *nonneg-Reals-inverse-I* [simp]:  
**fixes**  $a :: 'a::\text{real-div-algebra}$   
**shows**  $a \in \mathbb{R}_{\geq 0} \implies \text{inverse } a \in \mathbb{R}_{\geq 0}$   
*<proof>*

**lemma** *nonneg-Reals-divide-I* [simp]:  
**fixes**  $a :: 'a::\text{real-div-algebra}$   
**shows**  $[[a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0}] \implies a / b \in \mathbb{R}_{\geq 0}$   
*<proof>*

**lemma** *nonneg-Reals-pow-I* [simp]:  $a \in \mathbb{R}_{\geq 0} \implies a^{\hat{n}} \in \mathbb{R}_{\geq 0}$   
*<proof>*

**lemma** *complex-nonneg-Reals-iff*:  $z \in \mathbb{R}_{\geq 0} \longleftrightarrow \text{Re } z \geq 0 \wedge \text{Im } z = 0$   
*<proof>*

**lemma** *ii-not-nonneg-Reals* [iff]:  $i \notin \mathbb{R}_{\geq 0}$   
*<proof>*

### 70.3 Non-positive reals

**definition** *nonpos-Reals* ::  $'a::\text{real-algebra-1}$  set ( $\langle \mathbb{R}_{\leq 0} \rangle$ )  
**where**  $\mathbb{R}_{\leq 0} = \{\text{of-real } r \mid r. r \leq 0\}$

**lemma** *nonpos-Reals-of-real-iff* [simp]:  $\text{of-real } r \in \mathbb{R}_{\leq 0} \longleftrightarrow r \leq 0$   
*<proof>*

**lemma** *nonpos-Reals-subset-Reals*:  $\mathbb{R}_{\leq 0} \subseteq \mathbb{R}$   
*<proof>*

**lemma** *nonpos-Ints-subset-nonpos-Reals*:  $\mathbb{Z}_{\leq 0} \subseteq \mathbb{R}_{\leq 0}$   
*<proof>*

**lemma** *nonpos-Reals-of-nat-iff* [simp]:  $\text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow n=0$   
*<proof>*

**lemma** *nonpos-Reals-Real* [dest]:  $x \in \mathbb{R}_{\leq 0} \implies x \in \mathbb{R}$   
*<proof>*

**lemma** *nonpos-Reals-cases*:

**assumes**  $x \in \mathbb{R}_{\leq 0}$

**obtains**  $r$  **where**  $x = \text{of-real } r$   $r \leq 0$

*<proof>*

**lemma** *uminus-nonneg-Reals-iff* [simp]:  $-x \in \mathbb{R}_{\geq 0} \longleftrightarrow x \in \mathbb{R}_{\leq 0}$

*<proof>*

**lemma** *uminus-nonpos-Reals-iff* [simp]:  $-x \in \mathbb{R}_{\leq 0} \longleftrightarrow x \in \mathbb{R}_{\geq 0}$

*<proof>*

**lemma** *nonpos-Reals-zero-I* [simp]:  $0 \in \mathbb{R}_{\leq 0}$

*<proof>*

**lemma** *nonpos-Reals-one-I* [simp]:  $1 \notin \mathbb{R}_{\leq 0}$

*<proof>*

**lemma** *nonpos-Reals-numeral-I* [simp]: *numeral*  $w \notin \mathbb{R}_{\leq 0}$

*<proof>*

**lemma** *nonpos-Reals-add-I* [simp]:  $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a + b \in \mathbb{R}_{\leq 0}$

*<proof>*

**lemma** *nonpos-Reals-mult-I1*:  $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a * b \in \mathbb{R}_{\leq 0}$

*<proof>*

**lemma** *nonpos-Reals-mult-I2*:  $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a * b \in \mathbb{R}_{\leq 0}$

*<proof>*

**lemma** *nonpos-Reals-mult-of-nat-iff*:

**fixes**  $a :: 'a :: \text{real-div-algebra}$  **shows**  $a * \text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n=0$

*<proof>*

**lemma** *nonpos-Reals-inverse-I*:

**fixes**  $a :: 'a :: \text{real-div-algebra}$

**shows**  $a \in \mathbb{R}_{\leq 0} \implies \text{inverse } a \in \mathbb{R}_{\leq 0}$

*<proof>*

**lemma** *nonpos-Reals-divide-I1*:

**fixes**  $a :: 'a :: \text{real-div-algebra}$

**shows**  $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a / b \in \mathbb{R}_{\leq 0}$

*<proof>*

**lemma** *nonpos-Reals-divide-I2*:

**fixes**  $a :: 'a :: \text{real-div-algebra}$

**shows**  $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a / b \in \mathbb{R}_{\leq 0}$

*<proof>*

**lemma** *nonpos-Reals-divide-of-nat-iff*:

**fixes**  $a :: 'a :: \text{real-div-algebra}$  **shows**  $a / \text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n = 0$   
 $\langle \text{proof} \rangle$

**lemma** *nonpos-Reals-inverse-iff* [simp]:

**fixes**  $a :: 'a :: \text{real-div-algebra}$   
**shows**  $\text{inverse } a \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0}$   
 $\langle \text{proof} \rangle$

**lemma** *nonpos-Reals-pow-I*:  $\llbracket a \in \mathbb{R}_{\leq 0}; \text{odd } n \rrbracket \implies a^{\widehat{n}} \in \mathbb{R}_{\leq 0}$

$\langle \text{proof} \rangle$

**lemma** *complex-nonpos-Reals-iff*:  $z \in \mathbb{R}_{\leq 0} \longleftrightarrow \text{Re } z \leq 0 \wedge \text{Im } z = 0$

$\langle \text{proof} \rangle$

**lemma** *ii-not-nonpos-Reals* [iff]:  $i \notin \mathbb{R}_{\leq 0}$

$\langle \text{proof} \rangle$

**end**

## 71 Numeral Syntax for Types

**theory** *Numeral-Type*

**imports** *Cardinality*

**begin**

### 71.1 Numeral Types

**typedef** *num0* = *UNIV* :: *nat set*  $\langle \text{proof} \rangle$

**typedef** *num1* = *UNIV* :: *unit set*  $\langle \text{proof} \rangle$

**typedef**  $'a \text{ bit0}$  =  $\{0 ..< 2 * \text{int } \text{CARD}('a::\text{finite})\}$

$\langle \text{proof} \rangle$

**typedef**  $'a \text{ bit1}$  =  $\{0 ..< 1 + 2 * \text{int } \text{CARD}('a::\text{finite})\}$

$\langle \text{proof} \rangle$

**lemma** *card-num0* [simp]:  $\text{CARD}(\text{num0}) = 0$

$\langle \text{proof} \rangle$

**lemma** *infinite-num0*:  $\neg \text{finite}(\text{UNIV} :: \text{num0 set})$

$\langle \text{proof} \rangle$

**lemma** *card-num1* [simp]:  $\text{CARD}(\text{num1}) = 1$

$\langle \text{proof} \rangle$

**lemma** *card-bit0* [simp]:  $\text{CARD}('a \text{ bit0}) = 2 * \text{CARD}('a::\text{finite})$

$\langle \text{proof} \rangle$

**lemma** *card-bit1* [*simp*]:  $CARD('a \text{ bit1}) = Suc (2 * CARD('a::finite))$   
 ⟨*proof*⟩

## 71.2 *num1*

**instance** *num1* :: *finite*  
 ⟨*proof*⟩

**instantiation** *num1* :: *CARD-1*  
**begin**

**instance**  
 ⟨*proof*⟩

**end**

**lemma** *num1-eq-iff*:  $(x::num1) = (y::num1) \longleftrightarrow True$   
 ⟨*proof*⟩

**instantiation** *num1* :: {*comm-ring, comm-monoid-mult, numeral*}  
**begin**

**instance**  
 ⟨*proof*⟩

**end**

**lemma** *num1-eqI*:  
**fixes** *a::num1* **shows**  $a = b$   
 ⟨*proof*⟩

**lemma** *num1-eq1* [*simp*]:  
**fixes** *a::num1* **shows**  $a = 1$   
 ⟨*proof*⟩

**lemma** *forall-1* [*simp*]:  $(\forall i::num1. P i) \longleftrightarrow P 1$   
 ⟨*proof*⟩

**lemma** *ex-1* [*simp*]:  $(\exists x::num1. P x) \longleftrightarrow P 1$   
 ⟨*proof*⟩

**instantiation** *num1* :: *linorder* **begin**

**definition**  $a < b \longleftrightarrow Rep\text{-}num1\ a < Rep\text{-}num1\ b$

**definition**  $a \leq b \longleftrightarrow Rep\text{-}num1\ a \leq Rep\text{-}num1\ b$

**instance**  
 ⟨*proof*⟩

**end**

**instance** *num1* :: *wellorder*

*<proof>*

**instance** *bit0* :: (*finite*) *card2*  
*<proof>*

**instance** *bit1* :: (*finite*) *card2*  
*<proof>*

### 71.3 Locales for modular arithmetic subtypes

**locale** *mod-type* =  
**fixes** *n* :: *int*  
**and** *Rep* :: 'a::{*zero,one,plus,times,uminus,minus*} ⇒ *int*  
**and** *Abs* :: *int* ⇒ 'a::{*zero,one,plus,times,uminus,minus*}  
**assumes** *type*: *type-definition Rep Abs {0..*n*}*  
**and** *size1*:  $1 < n$   
**and** *zero-def*:  $0 = \text{Abs } 0$   
**and** *one-def*:  $1 = \text{Abs } 1$   
**and** *add-def*:  $x + y = \text{Abs } ((\text{Rep } x + \text{Rep } y) \bmod n)$   
**and** *mult-def*:  $x * y = \text{Abs } ((\text{Rep } x * \text{Rep } y) \bmod n)$   
**and** *diff-def*:  $x - y = \text{Abs } ((\text{Rep } x - \text{Rep } y) \bmod n)$   
**and** *minus-def*:  $-x = \text{Abs } ((-\text{Rep } x) \bmod n)$   
**begin**

**lemma** *size0*:  $0 < n$   
*<proof>*

**lemmas** *definitions* =  
*zero-def one-def add-def mult-def minus-def diff-def*

**lemma** *Rep-less-n*:  $\text{Rep } x < n$   
*<proof>*

**lemma** *Rep-le-n*:  $\text{Rep } x \leq n$   
*<proof>*

**lemma** *Rep-inject-sym*:  $x = y \longleftrightarrow \text{Rep } x = \text{Rep } y$   
*<proof>*

**lemma** *Rep-inverse*:  $\text{Abs } (\text{Rep } x) = x$   
*<proof>*

**lemma** *Abs-inverse*:  $m \in \{0..*n\} \implies \text{Rep } (\text{Abs } m) = m*$   
*<proof>*

**lemma** *Rep-Abs-mod*:  $\text{Rep } (\text{Abs } (m \bmod n)) = m \bmod n$   
*<proof>*

**lemma** *Rep-Abs-0*:  $\text{Rep } (\text{Abs } 0) = 0$   
 ⟨proof⟩

**lemma** *Rep-0*:  $\text{Rep } 0 = 0$   
 ⟨proof⟩

**lemma** *Rep-Abs-1*:  $\text{Rep } (\text{Abs } 1) = 1$   
 ⟨proof⟩

**lemma** *Rep-1*:  $\text{Rep } 1 = 1$   
 ⟨proof⟩

**lemma** *Rep-mod*:  $\text{Rep } x \text{ mod } n = \text{Rep } x$   
 ⟨proof⟩

**lemmas** *Rep-simps* =  
*Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1*

**lemma** *comm-ring-1*:  $\text{OFCLASS}('a, \text{comm-ring-1-class})$   
 ⟨proof⟩

**end**

**locale** *mod-ring* = *mod-type*  $n$  *Rep* *Abs*  
**for**  $n :: \text{int}$   
**and**  $\text{Rep} :: 'a :: \{\text{comm-ring-1}\} \Rightarrow \text{int}$   
**and**  $\text{Abs} :: \text{int} \Rightarrow 'a :: \{\text{comm-ring-1}\}$   
**begin**

**lemma** *of-nat-eq*:  $\text{of-nat } k = \text{Abs } (\text{int } k \text{ mod } n)$   
 ⟨proof⟩

**lemma** *of-int-eq*:  $\text{of-int } z = \text{Abs } (z \text{ mod } n)$   
 ⟨proof⟩

**lemma** *Rep-numeral*:  $\text{Rep } (\text{numeral } w) = \text{numeral } w \text{ mod } n$   
 ⟨proof⟩

**lemma** *iszero-numeral*:  
 $\text{iszero } (\text{numeral } w :: 'a) \longleftrightarrow \text{numeral } w \text{ mod } n = 0$   
 ⟨proof⟩

**lemma** *cases*:  
**assumes**  $1: \bigwedge z. \llbracket (x :: 'a) = \text{of-int } z; 0 \leq z; z < n \rrbracket \Longrightarrow P$   
**shows**  $P$   
 ⟨proof⟩

**lemma** *induct*:  
 $(\bigwedge z. \llbracket 0 \leq z; z < n \rrbracket \Longrightarrow P (\text{of-int } z)) \Longrightarrow P (x :: 'a)$

*<proof>*

**lemma** *UNIV-eq*: (*UNIV* :: 'a set) = Abs ' {0..*n*}  
*<proof>*

**lemma** *CARD-eq*: *CARD*('a) = nat *n*  
*<proof>*

**lemma** *CHAR-eq* [*simp*]: *CHAR*('a) = *CARD*('a)  
*<proof>*

**end**

## 71.4 Ring class instances

Unfortunately *ring-1* instance is not possible for *num1*, since 0 and 1 are not distinct.

**instantiation**

*bit0* and *bit1* :: (*finite*) {*zero,one,plus,times,uminus,minus*}  
**begin**

**definition** *Abs-bit0'* :: int  $\Rightarrow$  'a *bit0* **where**  
*Abs-bit0'* *x* = *Abs-bit0* (*x mod int CARD*('a *bit0*))

**definition** *Abs-bit1'* :: int  $\Rightarrow$  'a *bit1* **where**  
*Abs-bit1'* *x* = *Abs-bit1* (*x mod int CARD*('a *bit1*))

**definition** 0 = *Abs-bit0* 0

**definition** 1 = *Abs-bit0* 1

**definition** *x + y* = *Abs-bit0'* (*Rep-bit0* *x* + *Rep-bit0* *y*)

**definition** *x \* y* = *Abs-bit0'* (*Rep-bit0* *x* \* *Rep-bit0* *y*)

**definition** *x - y* = *Abs-bit0'* (*Rep-bit0* *x* - *Rep-bit0* *y*)

**definition** - *x* = *Abs-bit0'* (- *Rep-bit0* *x*)

**definition** 0 = *Abs-bit1* 0

**definition** 1 = *Abs-bit1* 1

**definition** *x + y* = *Abs-bit1'* (*Rep-bit1* *x* + *Rep-bit1* *y*)

**definition** *x \* y* = *Abs-bit1'* (*Rep-bit1* *x* \* *Rep-bit1* *y*)

**definition** *x - y* = *Abs-bit1'* (*Rep-bit1* *x* - *Rep-bit1* *y*)

**definition** - *x* = *Abs-bit1'* (- *Rep-bit1* *x*)

**instance** *<proof>*

**end**

**interpretation** *bit0*:

*mod-type int CARD*('a::*finite bit0*)

*Rep-bit0* :: 'a::*finite bit0*  $\Rightarrow$  int

*Abs-bit0* :: int  $\Rightarrow$  'a::*finite bit0*



*<proof>*

**interpretation** *bit1*:

*mod-type int CARD('a::finite bit1)*  
*Rep-bit1 :: 'a::finite bit1  $\Rightarrow$  int*  
*Abs-bit1 :: int  $\Rightarrow$  'a::finite bit1*  
*<proof>*

**instance** *bit0* :: (*finite*) *comm-ring-1*

*<proof>*

**instance** *bit1* :: (*finite*) *comm-ring-1*

*<proof>*

**interpretation** *bit0*:

*mod-ring int CARD('a::finite bit0)*  
*Rep-bit0 :: 'a::finite bit0  $\Rightarrow$  int*  
*Abs-bit0 :: int  $\Rightarrow$  'a::finite bit0*  
*<proof>*

**interpretation** *bit1*:

*mod-ring int CARD('a::finite bit1)*  
*Rep-bit1 :: 'a::finite bit1  $\Rightarrow$  int*  
*Abs-bit1 :: int  $\Rightarrow$  'a::finite bit1*  
*<proof>*

Set up cases, induction, and arithmetic

**lemmas** *bit0-cases* [*case-names of-int*, *cases type: bit0*] = *bit0.cases*

**lemmas** *bit1-cases* [*case-names of-int*, *cases type: bit1*] = *bit1.cases*

**lemmas** *bit0-induct* [*case-names of-int*, *induct type: bit0*] = *bit0.induct*

**lemmas** *bit1-induct* [*case-names of-int*, *induct type: bit1*] = *bit1.induct*

**lemmas** *bit0-iszero-numeral* [*simp*] = *bit0.iszero-numeral*

**lemmas** *bit1-iszero-numeral* [*simp*] = *bit1.iszero-numeral*

**lemmas** [*simp*] = *eq-numeral-iff-iszero* [**where** *'a='a bit0*] **for** *dummy :: 'a::finite*

**lemmas** [*simp*] = *eq-numeral-iff-iszero* [**where** *'a='a bit1*] **for** *dummy :: 'a::finite*

## 71.5 Order instances

**instantiation** *bit0* and *bit1* :: (*finite*) *linorder* **begin**

**definition**  $a < b \longleftrightarrow \text{Rep-bit0 } a < \text{Rep-bit0 } b$

**definition**  $a \leq b \longleftrightarrow \text{Rep-bit0 } a \leq \text{Rep-bit0 } b$

**definition**  $a < b \longleftrightarrow \text{Rep-bit1 } a < \text{Rep-bit1 } b$

**definition**  $a \leq b \longleftrightarrow \text{Rep-bit1 } a \leq \text{Rep-bit1 } b$

**instance**

*<proof>*

**end**

```
instance bit0 and bit1 :: (finite) wellorder
⟨proof⟩
```

## 71.6 Code setup and type classes for code generation

Code setup for *num0* and *num1*

```
definition Num0 :: num0 where Num0 = Abs-num0 0
code-datatype Num0
```

```
instantiation num0 :: equal begin
definition equal-num0 :: num0 ⇒ num0 ⇒ bool
  where equal-num0 = (=)
instance ⟨proof⟩
end
```

```
lemma equal-num0-code [code]:
  equal-class.equal Num0 Num0 = True
⟨proof⟩
```

```
code-datatype 1 :: num1
```

```
instantiation num1 :: equal begin
definition equal-num1 :: num1 ⇒ num1 ⇒ bool
  where equal-num1 = (=)
instance ⟨proof⟩
end
```

```
lemma equal-num1-code [code]:
  equal-class.equal (1 :: num1) 1 = True
⟨proof⟩
```

```
instantiation num1 :: enum begin
definition enum-class.enum = [1 :: num1]
definition enum-class.enum-all P = P (1 :: num1)
definition enum-class.enum-ex P = P (1 :: num1)
instance
  ⟨proof⟩
end
```

```
instantiation num0 and num1 :: card-UNIV begin
definition finite-UNIV = Phantom(num0) False
definition card-UNIV = Phantom(num0) 0
definition finite-UNIV = Phantom(num1) True
definition card-UNIV = Phantom(num1) 1
instance
  ⟨proof⟩
end
```

Code setup for 'a bit0 and 'a bit1

**declare**

*bit0.Rep-inverse*[code abstype]  
*bit0.Rep-0*[code abstract]  
*bit0.Rep-1*[code abstract]

**lemma** *Abs-bit0'-code* [code abstract]:

*Rep-bit0* (*Abs-bit0'*  $x :: 'a :: \text{finite } bit0$ ) =  $x \text{ mod int } (CARD('a \text{ bit0}))$   
 ⟨proof⟩

**lemma** *inj-on-Abs-bit0*:

*inj-on* (*Abs-bit0* ::  $\text{int} \Rightarrow 'a \text{ bit0}$ )  $\{0..<2 * \text{int } CARD('a :: \text{finite})\}$   
 ⟨proof⟩

**declare**

*bit1.Rep-inverse*[code abstype]  
*bit1.Rep-0*[code abstract]  
*bit1.Rep-1*[code abstract]

**lemma** *Abs-bit1'-code* [code abstract]:

*Rep-bit1* (*Abs-bit1'*  $x :: 'a :: \text{finite } bit1$ ) =  $x \text{ mod int } (CARD('a \text{ bit1}))$   
 ⟨proof⟩

**lemma** *inj-on-Abs-bit1*:

*inj-on* (*Abs-bit1* ::  $\text{int} \Rightarrow 'a \text{ bit1}$ )  $\{0..<1 + 2 * \text{int } CARD('a :: \text{finite})\}$   
 ⟨proof⟩

**instantiation** *bit0 and bit1* :: (*finite*) *equal begin*

**definition** *equal-class.equal*  $x \ y \longleftrightarrow \text{Rep-bit0 } x = \text{Rep-bit0 } y$

**definition** *equal-class.equal*  $x \ y \longleftrightarrow \text{Rep-bit1 } x = \text{Rep-bit1 } y$

**instance**

⟨proof⟩

**end**

**instantiation** *bit0* :: (*finite*) *enum begin*

**definition** (*enum-class.enum* ::  $'a \text{ bit0 list}$ ) =  $\text{map } (Abs-bit0' \circ \text{int}) (\text{upt } 0 \ (CARD('a \text{ bit0})))$

**definition** *enum-class.enum-all*  $P = (\forall b :: 'a \text{ bit0} \in \text{set } \text{enum-class.enum}. P \ b)$

**definition** *enum-class.enum-ex*  $P = (\exists b :: 'a \text{ bit0} \in \text{set } \text{enum-class.enum}. P \ b)$

**instance** ⟨proof⟩

**end**

**instantiation** *bit1* :: (*finite*) *enum begin*

**definition** (*enum-class.enum* ::  $'a \text{ bit1 list}$ ) =  $\text{map } (Abs-bit1' \circ \text{int}) (\text{upt } 0 \ (CARD('a \text{ bit1})))$

**definition** *enum-class.enum-all*  $P = (\forall b :: 'a \text{ bit1} \in \text{set enum-class.enum. } P \ b)$

**definition** *enum-class.enum-ex*  $P = (\exists b :: 'a \text{ bit1} \in \text{set enum-class.enum. } P \ b)$

**instance**

*<proof>*

**end**

**instantiation** *bit0* and *bit1* :: (*finite*) *finite-UNIV* **begin**

**definition** *finite-UNIV* = *Phantom('a bit0) True*

**definition** *finite-UNIV* = *Phantom('a bit1) True*

**instance** *<proof>*

**end**

**instantiation** *bit0* and *bit1* :: (*{finite,card-UNIV}*) *card-UNIV* **begin**

**definition** *card-UNIV* = *Phantom('a bit0) (2 \* of-phantom (card-UNIV :: 'a card-UNIV))*

**definition** *card-UNIV* = *Phantom('a bit1) (1 + 2 \* of-phantom (card-UNIV :: 'a card-UNIV))*

**instance** *<proof>*

**end**

## 71.7 Syntax

**syntax**

*-NumeralType* :: *num-token => type (<(<open-block notation=<type-literal number>>-)>)*

*-NumeralType0* :: *type (<(<open-block notation=<type-literal number>>0)>)*

*-NumeralType1* :: *type (<(<open-block notation=<type-literal number>>1)>)*

**translations**

*(type) 1 == (type) num1*

*(type) 0 == (type) num0*

*<ML>*

## 71.8 Examples

**lemma** *CARD(0) = 0* *<proof>*

**lemma** *CARD(17) = 17* *<proof>*

**lemma** *CHAR(23) = 23* *<proof>*

**lemma**  $8 * 11^3 - 6 = (2::5)$  *<proof>*

**end**

## 72 $\omega$ -words

**theory** *Omega-Words-Fun*

**imports** *Infinite-Set*  
**begin**

Note: This theory is based on Stefan Merz’s work.

Automata recognize languages, which are sets of words. For the theory of  $\omega$ -automata, we are mostly interested in  $\omega$ -words, but it is sometimes useful to reason about finite words, too. We are modeling finite words as lists; this lets us benefit from the existing library. Other formalizations could be investigated, such as representing words as functions whose domains are initial intervals of the natural numbers.

## 72.1 Type declaration and elementary operations

We represent  $\omega$ -words as functions from the natural numbers to the alphabet type. Other possible formalizations include a coinductive definition or a uniform encoding of finite and infinite words, as studied by Müller et al.

**type-synonym**

*'a word* = nat  $\Rightarrow$  *'a*

We can prefix a finite word to an  $\omega$ -word, and a way to obtain an  $\omega$ -word from a finite, non-empty word is by  $\omega$ -iteration.

**definition**

*conc* :: [*'a list*, *'a word*]  $\Rightarrow$  *'a word* (**infixr**  $\frown$  65)  
**where**  $w \frown x == \lambda n. \text{if } n < \text{length } w \text{ then } w!n \text{ else } x (n - \text{length } w)$

**definition**

*iter* :: *'a list*  $\Rightarrow$  *'a word* ( $\langle \langle \text{notation} = \langle \text{postfix } \omega \rangle \rangle \rangle^{-\omega}$ ) [1000]  
**where**  $\text{iter } w == \text{if } w = [] \text{ then undefined else } (\lambda n. w!(n \bmod (\text{length } w)))$

**lemma** *conc-empty[simp]*:  $[] \frown w = w$

*<proof>*

**lemma** *conc-fst[simp]*:  $n < \text{length } w \implies (w \frown x) n = w!n$

*<proof>*

**lemma** *conc-snd[simp]*:  $\neg(n < \text{length } w) \implies (w \frown x) n = x (n - \text{length } w)$

*<proof>*

**lemma** *iter-nth [simp]*:  $0 < \text{length } w \implies w^\omega n = w!(n \bmod (\text{length } w))$

*<proof>*

**lemma** *conc-conc[simp]*:  $u \frown v \frown w = (u @ v) \frown w$  (**is** ?lhs = ?rhs)

*<proof>*

**lemma** *range-conc[simp]*:  $\text{range } (w_1 \frown w_2) = \text{set } w_1 \cup \text{range } w_2$

*<proof>*

**lemma** *iter-unroll*:  $0 < \text{length } w \implies w^\omega = w \frown w^\omega$   
 ⟨proof⟩

## 72.2 Subsequence, Prefix, and Suffix

**definition** *suffix* ::  $[\text{nat}, 'a \text{ word}] \Rightarrow 'a \text{ word}$   
 where  $\text{suffix } k \ x \equiv \lambda n. x \ (k+n)$

**definition** *subsequence* ::  $'a \text{ word} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ list}$   
 (⟨⟨open-block notation=⟨mixfix subsequence⟩⟩- [- → -]⟩ 900)  
 where  $\text{subsequence } w \ i \ j \equiv \text{map } w \ [i..<j]$

**abbreviation** *prefix* ::  $\text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ list}$   
 where  $\text{prefix } n \ w \equiv \text{subsequence } w \ 0 \ n$

**lemma** *suffix-nth* [*simp*]:  $(\text{suffix } k \ x) \ n = x \ (k+n)$   
 ⟨proof⟩

**lemma** *suffix-0* [*simp*]:  $\text{suffix } 0 \ x = x$   
 ⟨proof⟩

**lemma** *suffix-suffix* [*simp*]:  $\text{suffix } m \ (\text{suffix } k \ x) = \text{suffix } (k+m) \ x$   
 ⟨proof⟩

**lemma** *subsequence-append*:  $\text{prefix } (i + j) \ w = \text{prefix } i \ w \ @ \ (w \ [i \rightarrow i + j])$   
 ⟨proof⟩

**lemma** *subsequence-drop*[*simp*]:  $\text{drop } i \ (w \ [j \rightarrow k]) = w \ [j + i \rightarrow k]$   
 ⟨proof⟩

**lemma** *subsequence-empty*[*simp*]:  $w \ [i \rightarrow j] = [] \iff j \leq i$   
 ⟨proof⟩

**lemma** *subsequence-length*[*simp*]:  $\text{length } (\text{subsequence } w \ i \ j) = j - i$   
 ⟨proof⟩

**lemma** *subsequence-nth*[*simp*]:  $k < j - i \implies (w \ [i \rightarrow j]) \ ! \ k = w \ (i + k)$   
 ⟨proof⟩

**lemma** *subseq-to-zero*[*simp*]:  $w \ [i \rightarrow 0] = []$   
 ⟨proof⟩

**lemma** *subseq-to-smaller*[*simp*]:  $i \geq j \implies w \ [i \rightarrow j] = []$   
 ⟨proof⟩

**lemma** *subseq-to-Suc*[*simp*]:  $i \leq j \implies w \ [i \rightarrow \text{Suc } j] = w \ [i \rightarrow j] \ @ \ [w \ j]$   
 ⟨proof⟩

**lemma** *subsequence-singleton*[simp]:  $w [i \rightarrow \text{Suc } i] = [w i]$   
 ⟨proof⟩

**lemma** *subsequence-prefix-suffix*:  $\text{prefix } (j - i) (\text{suffix } i w) = w [i \rightarrow j]$   
 ⟨proof⟩

**lemma** *prefix-suffix*:  $x = \text{prefix } n x \frown (\text{suffix } n x)$   
 ⟨proof⟩

**declare** *prefix-suffix*[*symmetric, simp*]

**lemma** *word-split*: **obtains**  $v_1 v_2$  **where**  $v = v_1 \frown v_2$  *length*  $v_1 = k$   
 ⟨proof⟩

**lemma** *set-subsequence*[simp]:  $\text{set } (w[i \rightarrow j]) = w\{i..<j\}$   
 ⟨proof⟩

**lemma** *subsequence-take*[simp]:  $\text{take } i (w [j \rightarrow k]) = w [j \rightarrow \min (j + i) k]$   
 ⟨proof⟩

**lemma** *subsequence-shift*[simp]:  $(\text{suffix } i w) [j \rightarrow k] = w [i + j \rightarrow i + k]$   
 ⟨proof⟩

**lemma** *suffix-subseq-join*[simp]:  $i \leq j \implies v [i \rightarrow j] \frown \text{suffix } j v = \text{suffix } i v$   
 ⟨proof⟩

**lemma** *prefix-conc-fst*[simp]:  
**assumes**  $j \leq \text{length } w$   
**shows**  $\text{prefix } j (w \frown w') = \text{take } j w$   
 ⟨proof⟩

**lemma** *prefix-conc-snd*[simp]:  
**assumes**  $n \geq \text{length } u$   
**shows**  $\text{prefix } n (u \frown v) = u @ \text{prefix } (n - \text{length } u) v$   
 ⟨proof⟩

**lemma** *prefix-conc-length*[simp]:  $\text{prefix } (\text{length } w) (w \frown w') = w$   
 ⟨proof⟩

**lemma** *suffix-conc-fst*[simp]:  
**assumes**  $n \leq \text{length } u$   
**shows**  $\text{suffix } n (u \frown v) = \text{drop } n u \frown v$   
 ⟨proof⟩

**lemma** *suffix-conc-snd*[simp]:  
**assumes**  $n \geq \text{length } u$

**shows**  $\text{suffix } n (u \frown v) = \text{suffix } (n - \text{length } u) v$   
 ⟨proof⟩

**lemma**  $\text{suffix-conc-length}[simp]$ :  $\text{suffix } (\text{length } w) (w \frown w') = w'$   
 ⟨proof⟩

**lemma**  $\text{concat-eq}[iff]$ :  
**assumes**  $\text{length } v_1 = \text{length } v_2$   
**shows**  $v_1 \frown u_1 = v_2 \frown u_2 \longleftrightarrow v_1 = v_2 \wedge u_1 = u_2$   
 (is ?lhs  $\longleftrightarrow$  ?rhs)  
 ⟨proof⟩

**lemma**  $\text{same-concat-eq}[iff]$ :  $u \frown v = u \frown w \longleftrightarrow v = w$   
 ⟨proof⟩

**lemma**  $\text{comp-concat}[simp]$ :  $f \circ u \frown v = \text{map } f u \frown (f \circ v)$   
 ⟨proof⟩

### 72.3 Prepending

**primrec**  $\text{build} :: 'a \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \text{ (infixr } \langle \#\#\rangle \text{ 65)}$   
**where**  $(a \#\# w) 0 = a \mid (a \#\# w) (\text{Suc } i) = w i$

**lemma**  $\text{build-eq}[iff]$ :  $a_1 \#\# w_1 = a_2 \#\# w_2 \longleftrightarrow a_1 = a_2 \wedge w_1 = w_2$   
 ⟨proof⟩

**lemma**  $\text{build-cons}[simp]$ :  $(a \# u) \frown v = a \#\# u \frown v$   
 ⟨proof⟩

**lemma**  $\text{build-append}[simp]$ :  $(w @ a \# u) \frown v = w \frown a \#\# u \frown v$   
 ⟨proof⟩

**lemma**  $\text{build-first}[simp]$ :  $w 0 \#\# \text{suffix } (\text{Suc } 0) w = w$   
 ⟨proof⟩

**lemma**  $\text{build-split}[intro]$ :  $w = w 0 \#\# \text{suffix } 1 w$   
 ⟨proof⟩

**lemma**  $\text{build-range}[simp]$ :  $\text{range } (a \#\# w) = \text{insert } a (\text{range } w)$   
 ⟨proof⟩

**lemma**  $\text{suffix-singleton-suffix}[simp]$ :  $w i \#\# \text{suffix } (\text{Suc } i) w = \text{suffix } i w$   
 ⟨proof⟩

Find the first occurrence of a letter from a given set

**lemma**  $\text{word-first-split-set}$ :  
**assumes**  $A \cap \text{range } w \neq \{\}$   
**obtains**  $u a v$  **where**  $w = u \frown [a] \frown v \wedge A \cap \text{set } u = \{\} \wedge a \in A$   
 ⟨proof⟩



## 72.4 The limit set of an $\omega$ -word

The limit set (also called infinity set) of an  $\omega$ -word is the set of letters that appear infinitely often in the word. This set plays an important role in defining acceptance conditions of  $\omega$ -automata.

**definition** *limit* :: 'a word  $\Rightarrow$  'a set  
**where** *limit*  $x \equiv \{a . \exists_{\infty} n . x n = a\}$

**lemma** *limit-iff-frequent*:  $a \in \text{limit } x \longleftrightarrow (\exists_{\infty} n . x n = a)$   
 ⟨proof⟩

The following is a different way to define the limit, using the reverse image, making the laws about reverse image applicable to the limit set. (Might want to change the definition above?)

**lemma** *limit-vimage*:  $(a \in \text{limit } x) = \text{infinite } (x - \{a\})$   
 ⟨proof⟩

**lemma** *two-in-limit-iff*:  
 $(\{a, b\} \subseteq \text{limit } x) =$   
 $((\exists n . x n = a) \wedge (\forall n . x n = a \longrightarrow (\exists m > n . x m = b)) \wedge (\forall m . x m = b \longrightarrow$   
 $(\exists n > m . x n = a)))$   
 (is ?lhs = (?r1  $\wedge$  ?r2  $\wedge$  ?r3))  
 ⟨proof⟩

For  $\omega$ -words over a finite alphabet, the limit set is non-empty. Moreover, from some position onward, any such word contains only letters from its limit set.

**lemma** *limit-nonempty*:  
**assumes** *fin*: *finite* (range  $x$ )  
**shows**  $\exists a . a \in \text{limit } x$   
 ⟨proof⟩

**lemmas** *limit-nonemptyE* = *limit-nonempty*[THEN *exE*]

**lemma** *limit-inter-INF*:  
**assumes** *hyp*:  $\text{limit } w \cap S \neq \{\}$   
**shows**  $\exists_{\infty} n . w n \in S$   
 ⟨proof⟩

The reverse implication is true only if  $S$  is finite.

**lemma** *INF-limit-inter*:  
**assumes** *hyp*:  $\exists_{\infty} n . w n \in S$   
**and** *fin*: *finite* ( $S \cap \text{range } w$ )  
**shows**  $\exists a . a \in \text{limit } w \cap S$   
 ⟨proof⟩

**lemma** *fin-ex-inf-eq-limit*: *finite*  $A \implies (\exists_{\infty} i . w i \in A) \longleftrightarrow \text{limit } w \cap A \neq \{\}$   
 ⟨proof⟩

**lemma** *limit-in-range-suffix*:  $\text{limit } x \subseteq \text{range } (\text{suffix } k \ x)$   
 ⟨proof⟩

**lemma** *limit-in-range*:  $\text{limit } r \subseteq \text{range } r$   
 ⟨proof⟩

**lemmas** *limit-in-range-suffixD* = *limit-in-range-suffix*[THEN subsetD]

**lemma** *limit-subset*:  $\text{limit } f \subseteq f \text{ ‘ } \{n..\}$   
 ⟨proof⟩

**theorem** *limit-is-suffix*:  
 assumes *fin*: *finite* (range *x*)  
 shows  $\exists k. \text{limit } x = \text{range } (\text{suffix } k \ x)$   
 ⟨proof⟩

**lemmas** *limit-is-suffixE* = *limit-is-suffix*[THEN exE]

The limit set enjoys some simple algebraic laws with respect to concatenation, suffixes, iteration, and renaming.

**theorem** *limit-conc* [*simp*]:  $\text{limit } (w \frown x) = \text{limit } x$   
 ⟨proof⟩

**theorem** *limit-suffix* [*simp*]:  $\text{limit } (\text{suffix } n \ x) = \text{limit } x$   
 ⟨proof⟩

**theorem** *limit-iter* [*simp*]:  
 assumes *nempty*:  $0 < \text{length } w$   
 shows  $\text{limit } w^\omega = \text{set } w$   
 ⟨proof⟩

**lemma** *limit-o* [*simp*]:  
 assumes *a*:  $a \in \text{limit } w$   
 shows  $f \ a \in \text{limit } (f \circ w)$   
 ⟨proof⟩

The converse relation is not true in general:  $f(a)$  can be in the limit of  $f \circ w$  even though  $a$  is not in the limit of  $w$ . However, *limit* commutes with renaming if the function is injective. More generally, if  $f(a)$  is the image of only finitely many elements, some of these must be in the limit of  $w$ .

**lemma** *limit-o-inv*:  
 assumes *fin*: *finite* ( $f \text{ ‘ } \{x\}$ )  
 and *x*:  $x \in \text{limit } (f \circ w)$   
 shows  $\exists a \in (f \text{ ‘ } \{x\}). a \in \text{limit } w$   
 ⟨proof⟩

**theorem** *limit-inj* [*simp*]:  
 assumes *inj*: *inj* *f*

**shows**  $\text{limit } (f \circ w) = f \cdot (\text{limit } w)$   
 ⟨proof⟩

**lemma** *limit-inter-empty*:  
**assumes** *fin*: *finite* (*range w*)  
**assumes** *hyp*:  $\text{limit } w \cap S = \{\}$   
**shows**  $\forall \infty n. w \ n \notin S$   
 ⟨proof⟩

If the limit is the suffix of the sequence’s range, we may increase the suffix index arbitrarily

**lemma** *limit-range-suffix-incr*:  
**assumes**  $\text{limit } r = \text{range } (\text{suffix } i \ r)$   
**assumes**  $j \geq i$   
**shows**  $\text{limit } r = \text{range } (\text{suffix } j \ r)$   
 (is ?lhs = ?rhs)  
 ⟨proof⟩

For two finite sequences, we can find a common suffix index such that the limits can be represented as these suffixes’ ranges.

**lemma** *common-range-limit*:  
**assumes** *finite* (*range x*)  
**and** *finite* (*range y*)  
**obtains** *i* **where**  $\text{limit } x = \text{range } (\text{suffix } i \ x)$   
**and**  $\text{limit } y = \text{range } (\text{suffix } i \ y)$   
 ⟨proof⟩

## 72.5 Index sequences and piecewise definitions

A word can be defined piecewise: given a sequence of words  $w_0, w_1, \dots$  and a strictly increasing sequence of integers  $i_0, i_1, \dots$  where  $i_0 = 0$ , a single word is obtained by concatenating subwords of the  $w_n$  as given by the integers: the resulting word is

$$(w_0)_{i_0} \dots (w_0)_{i_1-1} (w_1)_{i_1} \dots (w_1)_{i_2-1} \dots$$

We prepare the field by proving some trivial facts about such sequences of indexes.

**definition** *idx-sequence* :: *nat word*  $\Rightarrow$  *bool*  
**where** *idx-sequence* *idx*  $\equiv (\text{idx } 0 = 0) \wedge (\forall n. \text{idx } n < \text{idx } (\text{Suc } n))$

**lemma** *idx-sequence-less*:  
**assumes** *iseq*: *idx-sequence* *idx*  
**shows**  $\text{idx } n < \text{idx } (\text{Suc}(n+k))$   
 ⟨proof⟩

**lemma** *idx-sequence-inj*:  
**assumes** *iseq*: *idx-sequence* *idx*

**and**  $eq: idx\ m = idx\ n$   
**shows**  $m = n$   
 $\langle proof \rangle$

**lemma** *idx-sequence-mono*:  
**assumes**  $iseq: idx\text{-}sequence\ idx$   
**and**  $m: m \leq n$   
**shows**  $idx\ m \leq idx\ n$   
 $\langle proof \rangle$

Given an index sequence, every natural number is contained in the interval defined by two adjacent indexes, and in fact this interval is determined uniquely.

**lemma** *idx-sequence-idx*:  
**assumes**  $idx\text{-}sequence\ idx$   
**shows**  $idx\ k \in \{idx\ k ..< idx\ (Suc\ k)\}$   
 $\langle proof \rangle$

**lemma** *idx-sequence-interval*:  
**assumes**  $iseq: idx\text{-}sequence\ idx$   
**shows**  $\exists k. n \in \{idx\ k ..< idx\ (Suc\ k)\}$   
**(is**  $?P\ n$  **is**  $\exists k. ?in\ n\ k$ **)**  
 $\langle proof \rangle$

**lemma** *idx-sequence-interval-unique*:  
**assumes**  $iseq: idx\text{-}sequence\ idx$   
**and**  $k: n \in \{idx\ k ..< idx\ (Suc\ k)\}$   
**and**  $m: n \in \{idx\ m ..< idx\ (Suc\ m)\}$   
**shows**  $k = m$   
 $\langle proof \rangle$

**lemma** *idx-sequence-unique-interval*:  
**assumes**  $iseq: idx\text{-}sequence\ idx$   
**shows**  $\exists! k. n \in \{idx\ k ..< idx\ (Suc\ k)\}$   
 $\langle proof \rangle$

Now we can define the piecewise construction of a word using an index sequence.

**definition** *merge* ::  $'a\ word\ word \Rightarrow nat\ word \Rightarrow 'a\ word$   
**where**  $merge\ ws\ idx \equiv \lambda n. let\ i = THE\ i. n \in \{idx\ i ..< idx\ (Suc\ i)\}\ in\ ws\ i\ n$

**lemma** *merge*:  
**assumes**  $idx: idx\text{-}sequence\ idx$   
**and**  $n: n \in \{idx\ i ..< idx\ (Suc\ i)\}$   
**shows**  $merge\ ws\ idx\ n = ws\ i\ n$   
 $\langle proof \rangle$

**lemma** *merge0*:  
**assumes**  $idx: idx\text{-}sequence\ idx$

**shows** *merge ws idx 0 = ws 0 0*  
 ⟨*proof*⟩

**lemma** *merge-Suc*:

**assumes** *idx: idx-sequence idx*  
**and** *n: n ∈ {idx i ..< idx (Suc i)}*  
**shows** *merge ws idx (Suc n) = (if Suc n = idx (Suc i) then ws (Suc i) else ws i) (Suc n)*  
 ⟨*proof*⟩

**end**

## 73 Combinator syntax for generic, open state monads (single-threaded monads)

**theory** *Open-State-Syntax*

**imports** *Main*

**begin**

**context**

**includes** *state-combinator-syntax*

**begin**

### 73.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is transformed single-threadedly).

To enter from the Haskell world, [https://www.engr.mun.ca/~theo/Misc/haskell\\_and\\_monads.htm](https://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm) makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

### 73.2 State transformations and combinators

We classify functions operating on states into two categories:

**transformations** with type signature  $\sigma \Rightarrow \sigma'$ , transforming a state.

**“yielding” transformations** with type signature  $\sigma \Rightarrow \alpha \times \sigma'$ , “yielding” a side result while transforming a state.

**queries** with type signature  $\sigma \Rightarrow \alpha$ , computing a result dependent on a state.

By convention we write  $\sigma$  for types representing states and  $\alpha, \beta, \gamma, \dots$  for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type  $\sigma$  are used in a single-threaded way: after application of a transformation on a value of type  $\sigma$ , the former value should not be used again. To achieve this, we use a set of monad combinators:

Given two transformations  $f$  and  $g$ , they may be directly composed using the  $(\circ>)$  combinator, forming a forward composition:  $(f \circ> g) s = f (g s)$ .

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the  $(\circ\rightarrow)$  combinator:  $(f \circ\rightarrow (\lambda x. g)) s = (\text{let } (x, s') = f s \text{ in } g s')$ .

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *return* for this purpose.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.
- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units (“open monad”).
- The type of states may change due to a transformation.

### 73.3 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

**lemmas** *monad-simp = Pair-scomp scomp-Pair id-fcomp fcomp-id*  
*scomp-scomp scomp-fcomp fcomp-scomp fcomp-assoc*

Evaluation of monadic expressions by force:

**lemmas** *monad-collapse = monad-simp fcomp-apply scomp-apply split-beta*

**end**

### 73.4 Do-syntax

nonterminal *sdo-binds* and *sdo-bind*

**syntax**

```
-sdo-block :: sdo-binds ⇒ 'a
  (⟨⟨open-block notation=⟨mixfix exec block⟩⟩exec {//(2 -)//}) [12] 62)
-sdo-bind :: [pttrn, 'a] ⇒ sdo-bind
  (⟨⟨indent=2 notation=⟨infix exec bind⟩⟩- <- / -⟩ 13)
-sdo-let :: [pttrn, 'a] ⇒ sdo-bind
  (⟨⟨indent=2 notation=⟨infix exec let⟩⟩let - = / -⟩ [1000, 13] 13)
-sdo-then :: 'a ⇒ sdo-bind (⟨-⟩ [14] 13)
-sdo-final :: 'a ⇒ sdo-binds (⟨-⟩)
-sdo-cons :: [sdo-bind, sdo-binds] ⇒ sdo-binds
  (⟨⟨open-block notation=⟨infix exec next⟩⟩-; / -⟩ [13, 12] 12)
```

**syntax (ASCII)**

```
-sdo-bind :: [pttrn, 'a] ⇒ sdo-bind
  (⟨⟨indent=2 notation=⟨infix exec bind⟩⟩- <- / -⟩ 13)
```

**syntax-consts**

```
-sdo-let == Let
```

**translations**

```
-sdo-block (-sdo-cons (-sdo-bind p t) (-sdo-final e))
  == CONST scomp t (λp. e)
-sdo-block (-sdo-cons (-sdo-then t) (-sdo-final e))
  => CONST fcomp t e
-sdo-final (-sdo-block (-sdo-cons (-sdo-then t) (-sdo-final e)))
  <= -sdo-final (CONST fcomp t e)
-sdo-block (-sdo-cons (-sdo-then t) e)
  <= CONST fcomp t (-sdo-block e)
-sdo-block (-sdo-cons (-sdo-let p t) bs)
  == let p = t in -sdo-block bs
-sdo-block (-sdo-cons b (-sdo-cons c cs))
  == -sdo-block (-sdo-cons b (-sdo-final (-sdo-block (-sdo-cons c cs))))
-sdo-cons (-sdo-let p t) (-sdo-final s)
  == -sdo-final (let p = t in s)
-sdo-block (-sdo-final e) => e
```

For an example, see `~/src/HOL/Proofs/Extraction/Higman_Extraction.thy`.

end

## 74 Canonical order on option type

```
theory Option-ord
imports Main
begin
```

**unbundle** *lattice-syntax*

**instantiation** *option* :: (*preorder*) *preorder*  
**begin**

**definition** *less-eq-option* **where**

$x \leq y \longleftrightarrow (\text{case } x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } x \Rightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{False} \mid \text{Some } y \Rightarrow x \leq y))$

**definition** *less-option* **where**

$x < y \longleftrightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{False} \mid \text{Some } y \Rightarrow (\text{case } x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } x \Rightarrow x < y))$

**lemma** *less-eq-option-None* [*simp*]:  $\text{None} \leq x$   
 ⟨*proof*⟩

**lemma** *less-eq-option-None-code* [*code*]:  $\text{None} \leq x \longleftrightarrow \text{True}$   
 ⟨*proof*⟩

**lemma** *less-eq-option-None-is-None*:  $x \leq \text{None} \implies x = \text{None}$   
 ⟨*proof*⟩

**lemma** *less-eq-option-Some-None* [*simp*, *code*]:  $\text{Some } x \leq \text{None} \longleftrightarrow \text{False}$   
 ⟨*proof*⟩

**lemma** *less-eq-option-Some* [*simp*, *code*]:  $\text{Some } x \leq \text{Some } y \longleftrightarrow x \leq y$   
 ⟨*proof*⟩

**lemma** *less-option-None* [*simp*, *code*]:  $x < \text{None} \longleftrightarrow \text{False}$   
 ⟨*proof*⟩

**lemma** *less-option-None-is-Some*:  $\text{None} < x \implies \exists z. x = \text{Some } z$   
 ⟨*proof*⟩

**lemma** *less-option-None-Some* [*simp*]:  $\text{None} < \text{Some } x$   
 ⟨*proof*⟩

**lemma** *less-option-None-Some-code* [*code*]:  $\text{None} < \text{Some } x \longleftrightarrow \text{True}$   
 ⟨*proof*⟩

**lemma** *less-option-Some* [*simp*, *code*]:  $\text{Some } x < \text{Some } y \longleftrightarrow x < y$   
 ⟨*proof*⟩

**instance**  
 ⟨*proof*⟩

**end**

**instance** *option* :: (*order*) *order*



*<proof>*

**instance** *option* :: (*linorder*) *linorder*  
*<proof>*

**instantiation** *option* :: (*order*) *order-bot*  
**begin**

**definition** *bot-option* **where**  $\perp = \text{None}$

**instance**  
*<proof>*

**end**

**instantiation** *option* :: (*order-top*) *order-top*  
**begin**

**definition** *top-option* **where**  $\top = \text{Some } \top$

**instance**  
*<proof>*

**end**

**instance** *option* :: (*wellorder*) *wellorder*  
*<proof>*

**instantiation** *option* :: (*inf*) *inf*  
**begin**

**definition** *inf-option* **where**

$x \sqcap y = (\text{case } x \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } y \Rightarrow \text{Some } (x \sqcap y)))$

**lemma** *inf-None-1* [*simp*, *code*]:  $\text{None} \sqcap y = \text{None}$   
*<proof>*

**lemma** *inf-None-2* [*simp*, *code*]:  $x \sqcap \text{None} = \text{None}$   
*<proof>*

**lemma** *inf-Some* [*simp*, *code*]:  $\text{Some } x \sqcap \text{Some } y = \text{Some } (x \sqcap y)$   
*<proof>*

**instance** *<proof>*

**end**

**instantiation** *option* :: (*sup*) *sup*

**begin**

**definition** *sup-option* **where**

$x \sqcup y = (\text{case } x \text{ of } \text{None} \Rightarrow y \mid \text{Some } x' \Rightarrow (\text{case } y \text{ of } \text{None} \Rightarrow x \mid \text{Some } y \Rightarrow \text{Some } (x' \sqcup y)))$

**lemma** *sup-None-1* [*simp*, *code*]:  $\text{None} \sqcup y = y$   
 ⟨*proof*⟩

**lemma** *sup-None-2* [*simp*, *code*]:  $x \sqcup \text{None} = x$   
 ⟨*proof*⟩

**lemma** *sup-Some* [*simp*, *code*]:  $\text{Some } x \sqcup \text{Some } y = \text{Some } (x \sqcup y)$   
 ⟨*proof*⟩

**instance** ⟨*proof*⟩

**end**

**instance** *option* :: (*semilattice-inf*) *semilattice-inf*  
 ⟨*proof*⟩

**instance** *option* :: (*semilattice-sup*) *semilattice-sup*  
 ⟨*proof*⟩

**instance** *option* :: (*lattice*) *lattice* ⟨*proof*⟩

**instance** *option* :: (*lattice*) *bounded-lattice-bot* ⟨*proof*⟩

**instance** *option* :: (*bounded-lattice-top*) *bounded-lattice-top* ⟨*proof*⟩

**instance** *option* :: (*bounded-lattice-top*) *bounded-lattice* ⟨*proof*⟩

**instance** *option* :: (*distrib-lattice*) *distrib-lattice*  
 ⟨*proof*⟩

**instantiation** *option* :: (*complete-lattice*) *complete-lattice*  
**begin**

**definition** *Inf-option* :: 'a *option set*  $\Rightarrow$  'a *option* **where**

$\sqcap A = (\text{if } \text{None} \in A \text{ then } \text{None} \text{ else } \text{Some } (\sqcap \text{Option.the } A))$

**lemma** *None-in-Inf* [*simp*]:  $\text{None} \in A \Longrightarrow \sqcap A = \text{None}$   
 ⟨*proof*⟩

**definition** *Sup-option* :: 'a *option set*  $\Rightarrow$  'a *option* **where**

$\sqcup A = (\text{if } A = \{\} \vee A = \{\text{None}\} \text{ then } \text{None} \text{ else } \text{Some } (\sqcup \text{Option.the } A))$

**lemma** *empty-Sup* [*simp*]:  $\sqcup \{\} = \text{None}$

*<proof>*

**lemma** *singleton-None-Sup* [*simp*]:  $\sqcup \{None\} = None$   
*<proof>*

**instance**  
*<proof>*

**end**

**lemma** *Some-Inf*:  
 $Some (\prod A) = \prod (Some \text{ ` } A)$   
*<proof>*

**lemma** *Some-Sup*:  
 $A \neq \{\} \implies Some (\sqcup A) = \sqcup (Some \text{ ` } A)$   
*<proof>*

**lemma** *Some-INF*:  
 $Some (\prod_{x \in A}. f x) = (\prod_{x \in A}. Some (f x))$   
*<proof>*

**lemma** *Some-SUP*:  
 $A \neq \{\} \implies Some (\sqcup_{x \in A}. f x) = \sqcup_{x \in A}. Some (f x)$   
*<proof>*

**lemma** *option-Inf-Sup*:  $\prod (Sup \text{ ` } A) \leq \sqcup (Inf \text{ ` } \{f \text{ ` } A \mid f. \forall Y \in A. f Y \in Y\})$   
**for**  $A :: ('a :: complete-distrib-lattice \text{ option}) \text{ set set}$   
*<proof>*

**instance** *option* :: (*complete-distrib-lattice*) *complete-distrib-lattice*  
*<proof>*

**instance** *option* :: (*complete-linorder*) *complete-linorder* *<proof>*

**unbundle** *no lattice-syntax*

**end**

## 75 Futures and parallel lists for code generated towards Isabelle/ML

**theory** *Parallel*  
**imports** *Main*  
**begin**

### 75.1 Futures

**datatype** *'a future* = fork unit  $\Rightarrow$  *'a*

**primrec** *join* :: *'a future*  $\Rightarrow$  *'a* **where**  
*join* (fork *f*) = *f* ()

**lemma** *future-eqI* [intro!]:  
**assumes** *join f* = *join g*  
**shows** *f* = *g*  
 ⟨*proof*⟩

**code-printing**

**type-constructor** *future*  $\rightarrow$  (Eval) - *future*  
 | **constant** *fork*  $\rightarrow$  (Eval) *Future.fork*  
 | **constant** *join*  $\rightarrow$  (Eval) *Future.join*

**code-reserved** (Eval) *Future future*

### 75.2 Parallel lists

**definition** *map* :: (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  *'a list*  $\Rightarrow$  *'b list* **where**  
 [*simp*]: *map* = *List.map*

**definition** *forall* :: (*'a*  $\Rightarrow$  *bool*)  $\Rightarrow$  *'a list*  $\Rightarrow$  *bool* **where**  
*forall* = *list-all*

**lemma** *forall-all* [*simp*]:  
*forall P xs*  $\longleftrightarrow$  ( $\forall x \in \text{set } xs. P x$ )  
 ⟨*proof*⟩

**definition** *exists* :: (*'a*  $\Rightarrow$  *bool*)  $\Rightarrow$  *'a list*  $\Rightarrow$  *bool* **where**  
*exists* = *list-ex*

**lemma** *exists-ex* [*simp*]:  
*exists P xs*  $\longleftrightarrow$  ( $\exists x \in \text{set } xs. P x$ )  
 ⟨*proof*⟩

**code-printing**

**constant** *map*  $\rightarrow$  (Eval) *Par'-List.map*  
 | **constant** *forall*  $\rightarrow$  (Eval) *Par'-List.forall*  
 | **constant** *exists*  $\rightarrow$  (Eval) *Par'-List.exists*

**code-reserved** (Eval) *Par-List*

**hide-const** (open) *fork join map exists forall*

**end**

## 76 Input syntax for pattern aliases (or “as-patterns” in Haskell)

```
theory Pattern-Aliases
imports Main
begin
```

Most functional languages (Haskell, ML, Scala) support aliases in patterns. This allows to refer to a subpattern with a variable name. This theory implements this using a check phase. It works well for function definitions (see usage below). All features are packed into a **bundle**.

The following caveats should be kept in mind:

- The translation expects a term of the form  $f x y = rhs$ , where  $x$  and  $y$  are patterns that may contain aliases. The result of the translation is a nested *Let*-expression on the right hand side. The code generator *does not* print Isabelle pattern aliases to target language pattern aliases.
- The translation does not process nested equalities; only the top-level equality is translated.
- Terms that do not adhere to the above shape may either stay untranslated or produce an error message. The **fun** command will complain if pattern aliases are left untranslated. In particular, there are no checks whether the patterns are wellformed or linear.
- The corresponding uncheck phase attempts to reverse the translation (no guarantee). The additionally introduced variables are bound using a “fake quantifier” that does not appear in the output.
- To obtain reasonable induction principles in function definitions, the bundle also declares a custom congruence rule for *Let* that only affects **fun**. This congruence rule might lead to an explosion in term size (although that is rare)! In some circumstances (using *let* to destructure tuples), the internal construction of functions stumbles over this rule and prints an error. To mitigate this, either
  - activate the bundle locally (**context includes ... begin**) or
  - rewrite the *let*-expression to use *case*:  $let (a, b) = x in (b, a)$  becomes  $case x of (a, b) \Rightarrow (b, a)$ .
- The bundle also adds the  $Let ?s ?f \equiv ?f ?s$  rule to the simpset.

**76.1 Definition****consts**

*as* :: 'a ⇒ 'a ⇒ 'a  
*fake-quant* :: ('a ⇒ prop) ⇒ prop

**lemma** *let-cong-unfolding*:  $M = N \implies f N = g N \implies \text{Let } M f = \text{Let } N g$   
 ⟨*proof*⟩

**translations**  $P \leq \text{CONST } \text{fake-quant } (\lambda x. P)$

⟨*ML*⟩**bundle** *pattern-aliases* **begin**

**notation** *as* (**infixr** <=> 1)

⟨*ML*⟩

**declare** *let-cong-unfolding* [*fundef-cong*]  
**declare** *Let-def* [*simp*]

**end****hide-const** *as***hide-const** *fake-quant***76.2 Usage****context** **includes** *pattern-aliases* **begin**

Not very useful for plain definitions, but works anyway.

**private definition** *test-1*  $x (y =: z) = y + z$

**lemma** *test-1*  $x y = y + y$   
 ⟨*proof*⟩

Very useful for function definitions.

**private fun** *test-2* **where**

*test-2*  $(y \# (y' \# ys =: x') =: x) = x @ x' @ x' |$   
*test-2* - = []

**lemma** *test-2*  $(y \# y' \# ys) = (y \# y' \# ys) @ (y' \# ys) @ (y' \# ys)$   
 ⟨*proof*⟩

⟨*ML*⟩**end****end**

## 77 Periodic Functions

```
theory Periodic-Fun
imports Complex-Main
begin
```

A locale for periodic functions. The idea is that one proves  $f(x + p) = f(x)$  for some period  $p$  and gets derived results like  $f(x - p) = f(x)$  and  $f(x + 2p) = f(x)$  for free.

$g$  and  $gm$  are “plus/minus  $k$  periods” functions.  $g1$  and  $gn1$  are “plus/minus one period” functions. This is useful e.g. if the period is one; the lemmas one gets are then  $f(x + 1) = f x$  instead of  $f(x + 1 * 1) = f x$  etc.

```
locale periodic-fun =
  fixes f :: ('a :: {ring-1}) => 'b and g gm :: 'a => 'a => 'a and g1 gn1 :: 'a => 'a
  assumes plus-1: f (g1 x) = f x
  assumes periodic-arg-plus-0: g x 0 = x
  assumes periodic-arg-plus-distrib: g x (of-int (m + n)) = g (g x (of-int n)) (of-int m)
  assumes plus-1-eq: g x 1 = g1 x and minus-1-eq: g x (-1) = gn1 x
    and minus-eq: g x (-y) = gm x y
begin
```

```
lemma plus-of-nat: f (g x (of-nat n)) = f x
  <proof>
```

```
lemma minus-of-nat: f (gm x (of-nat n)) = f x
  <proof>
```

```
lemma plus-of-int: f (g x (of-int n)) = f x
  <proof>
```

```
lemma minus-of-int: f (gm x (of-int n)) = f x
  <proof>
```

```
lemma plus-numeral: f (g x (numeral n)) = f x
  <proof>
```

```
lemma minus-numeral: f (gm x (numeral n)) = f x
  <proof>
```

```
lemma minus-1: f (gn1 x) = f x
  <proof>
```

```
lemmas periodic-simps = plus-of-nat minus-of-nat plus-of-int minus-of-int
  plus-numeral minus-numeral plus-1 minus-1
```

```
end
```

Specialised case of the *periodic-fun* locale for periods that are not 1.

Gives lemmas  $f (x - period) = f x$  etc.

```

locale periodic-fun-simple =
  fixes f :: ('a :: {ring-1})  $\Rightarrow$  'b and period :: 'a
  assumes plus-period: f (x + period) = f x
begin
sublocale periodic-fun f  $\lambda z x. z + x * period$   $\lambda z x. z - x * period$ 
   $\lambda z. z + period$   $\lambda z. z - period$ 
  <proof>
end

```

Specialised case of the *periodic-fun* locale for period 1. Gives lemmas  $f (x - 1) = f x$  etc.

```

locale periodic-fun-simple' =
  fixes f :: ('a :: {ring-1})  $\Rightarrow$  'b
  assumes plus-period: f (x + 1) = f x
begin
sublocale periodic-fun f  $\lambda z x. z + x$   $\lambda z x. z - x$   $\lambda z. z + 1$   $\lambda z. z - 1$ 
  <proof>

```

**lemma** of-nat: f (of-nat n) = f 0 <proof>

**lemma** uminus-of-nat: f (-of-nat n) = f 0 <proof>

**lemma** of-int: f (of-int n) = f 0 <proof>

**lemma** uminus-of-int: f (-of-int n) = f 0 <proof>

**lemma** of-numeral: f (numeral n) = f 0 <proof>

**lemma** of-neg-numeral: f (-numeral n) = f 0 <proof>

**lemma** of-1: f 1 = f 0 <proof>

**lemma** of-neg-1: f (-1) = f 0 <proof>

**lemmas** periodic-simps' =

of-nat uminus-of-nat of-int uminus-of-int of-numeral of-neg-numeral of-1 of-neg-1

**end**

**lemma** sin-plus-pi: sin ((z :: 'a :: {real-normed-field,banach}) + of-real pi) = -  
sin z  
<proof>

**lemma** cos-plus-pi: cos ((z :: 'a :: {real-normed-field,banach}) + of-real pi) = -  
cos z  
<proof>

**interpretation** sin: periodic-fun-simple sin 2 \* of-real pi :: 'a :: {real-normed-field,banach}  
<proof>

**interpretation** cos: periodic-fun-simple cos 2 \* of-real pi :: 'a :: {real-normed-field,banach}  
<proof>

**interpretation** tan: periodic-fun-simple tan 2 \* of-real pi :: 'a :: {real-normed-field,banach}  
<proof>



**interpretation** *cot: periodic-fun-simple cot 2 \* of-real pi :: 'a :: {real-normed-field,banach}*  
 ⟨proof⟩

**lemma** *cos-eq-neg-periodic-intro:*

**assumes**  $x - y = 2*(of-int\ k)*pi + pi \vee x + y = 2*(of-int\ k)*pi + pi$   
**shows**  $\cos x = -\cos y$  ⟨proof⟩

**lemma** *cos-eq-periodic-intro:*

**assumes**  $x - y = 2*(of-int\ k)*pi \vee x + y = 2*(of-int\ k)*pi$   
**shows**  $\cos x = \cos y$   
 ⟨proof⟩

**lemma** *cos-eq-arccos-Ex:*

$\cos x = y \iff -1 \leq y \wedge y \leq 1 \wedge (\exists k::int. x = \arccos y + 2*k*pi \vee x = -\arccos y + 2*k*pi)$  (is ?L=?R)  
 ⟨proof⟩

end

## 78 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view

**theory** *Poly-Mapping*

**imports** *Groups-Big-Fun Fun-Lexorder More-List*

**begin**

### 78.1 Preliminary: auxiliary operations for *almost everywhere zero*

A central notion for polynomials are functions being *almost everywhere zero*. For these we provide some auxiliary definitions and lemmas.

**lemma** *finite-mult-not-eq-zero-leftI:*

**fixes**  $f :: 'b \Rightarrow 'a :: mult-zero$   
**assumes**  $finite\ \{a. f\ a \neq 0\}$   
**shows**  $finite\ \{a. g\ a * f\ a \neq 0\}$   
 ⟨proof⟩

**lemma** *finite-mult-not-eq-zero-rightI:*

**fixes**  $f :: 'b \Rightarrow 'a :: mult-zero$   
**assumes**  $finite\ \{a. f\ a \neq 0\}$   
**shows**  $finite\ \{a. f\ a * g\ a \neq 0\}$   
 ⟨proof⟩

**lemma** *finite-mult-not-eq-zero-prodI:*

**fixes**  $f\ g :: 'a \Rightarrow 'b::semiring-0$   
**assumes**  $finite\ \{a. f\ a \neq 0\}$  (is finite ?F)

**assumes** *finite* {*b*. *g b* ≠ 0} (**is** *finite* ?*G*)  
**shows** *finite* {(*a*, *b*). *f a* \* *g b* ≠ 0}  
 ⟨*proof*⟩

**lemma** *finite-not-eq-zero-sumI*:  
**fixes** *f g* :: '*a*::*monoid-add* ⇒ '*b*::*semiring-0*  
**assumes** *finite* {*a*. *f a* ≠ 0} (**is** *finite* ?*F*)  
**assumes** *finite* {*b*. *g b* ≠ 0} (**is** *finite* ?*G*)  
**shows** *finite* {*a* + *b* | *a b*. *f a* ≠ 0 ∧ *g b* ≠ 0} (**is** *finite* ?*FG*)  
 ⟨*proof*⟩

**lemma** *finite-mult-not-eq-zero-sumI*:  
**fixes** *f g* :: '*a*::*monoid-add* ⇒ '*b*::*semiring-0*  
**assumes** *finite* {*a*. *f a* ≠ 0}  
**assumes** *finite* {*b*. *g b* ≠ 0}  
**shows** *finite* {*a* + *b* | *a b*. *f a* \* *g b* ≠ 0}  
 ⟨*proof*⟩

**lemma** *finite-Sum-any-not-eq-zero-weakenI*:  
**assumes** *finite* {*a*. ∃ *b*. *f a b* ≠ 0}  
**shows** *finite* {*a*. *Sum-any* (*f a*) ≠ 0}  
 ⟨*proof*⟩

**context** *zero*  
**begin**

**definition** *when* :: '*a* ⇒ *bool* ⇒ '*a* (**infixl** ⟨*when*⟩ 20)  
**where**  
 (*a when P*) = (*if P then a else 0*)

Case distinctions always complicate matters, particularly when nested. The (*when*) operation allows to minimise these if *0* is the false-case value and makes proof obligations much more readable.

**lemma** *when [simp]*:  
 $P \implies (a \text{ when } P) = a$   
 $\neg P \implies (a \text{ when } P) = 0$   
 ⟨*proof*⟩

**lemma** *when-simps [simp]*:  
 (*a when True*) = *a*  
 (*a when False*) = 0  
 ⟨*proof*⟩

**lemma** *when-cong*:  
**assumes**  $P \longleftrightarrow Q$   
**and**  $Q \implies a = b$   
**shows** (*a when P*) = (*b when Q*)  
 ⟨*proof*⟩

**lemma** *zero-when* [*simp*]:

$$(0 \text{ when } P) = 0$$

*<proof>*

**lemma** *when-when*:

$$(a \text{ when } P \text{ when } Q) = (a \text{ when } P \wedge Q)$$

*<proof>*

**lemma** *when-commute*:

$$(a \text{ when } Q \text{ when } P) = (a \text{ when } P \text{ when } Q)$$

*<proof>*

**lemma** *when-neq-zero* [*simp*]:

$$(a \text{ when } P) \neq 0 \iff P \wedge a \neq 0$$

*<proof>*

**end**

**context** *monoid-add*

**begin**

**lemma** *when-add-distrib*:

$$(a + b \text{ when } P) = (a \text{ when } P) + (b \text{ when } P)$$

*<proof>*

**end**

**context** *semiring-1*

**begin**

**lemma** *zero-power-eq*:

$$0 \wedge n = (1 \text{ when } n = 0)$$

*<proof>*

**end**

**context** *comm-monoid-add*

**begin**

**lemma** *Sum-any-when-equal* [*simp*]:

$$(\sum a. (f a \text{ when } a = b)) = f b$$

*<proof>*

**lemma** *Sum-any-when-equal'* [*simp*]:

$$(\sum a. (f a \text{ when } b = a)) = f b$$

*<proof>*

**lemma** *Sum-any-when-independent*:

$$(\sum a. g a \text{ when } P) = ((\sum a. g a) \text{ when } P)$$

*<proof>*

**lemma** *Sum-any-when-dependent-prod-right:*

$(\sum (a, b). g \ a \ \text{when } b = h \ a) = (\sum a. g \ a)$   
*<proof>*

**lemma** *Sum-any-when-dependent-prod-left:*

$(\sum (a, b). g \ b \ \text{when } a = h \ b) = (\sum b. g \ b)$   
*<proof>*

**end**

**context** *cancel-comm-monoid-add*

**begin**

**lemma** *when-diff-distrib:*

$(a - b \ \text{when } P) = (a \ \text{when } P) - (b \ \text{when } P)$   
*<proof>*

**end**

**context** *group-add*

**begin**

**lemma** *when-uminus-distrib:*

$(- \ a \ \text{when } P) = - \ (a \ \text{when } P)$   
*<proof>*

**end**

**context** *mult-zero*

**begin**

**lemma** *mult-when:*

$a * (b \ \text{when } P) = (a * b \ \text{when } P)$   
*<proof>*

**lemma** *when-mult:*

$(a \ \text{when } P) * b = (a * b \ \text{when } P)$   
*<proof>*

**end**

## 78.2 Type definition

The following type is of central importance:

```
typedef (overloaded) ('a, 'b) poly-mapping (<(- =>0 /-)> [1, 0] 0) =
  {f :: 'a => 'b::zero. finite {x. f x ≠ 0}}
morphisms lookup Abs-poly-mapping
```

*<proof>*

**declare** *lookup-inverse* [*simp*]

**declare** *lookup-inject* [*simp*]

**lemma** *lookup-Abs-poly-mapping* [*simp*]:

*finite* {*x. f x ≠ 0*}  $\implies$  *lookup* (*Abs-poly-mapping* *f*) = *f*

*<proof>*

**lemma** *finite-lookup* [*simp*]:

*finite* {*k. lookup f k ≠ 0*}

*<proof>*

**lemma** *finite-lookup-nat* [*simp*]:

**fixes** *f* :: '*a*  $\Rightarrow_0$  *nat*

**shows** *finite* {*k. 0 < lookup f k*}

*<proof>*

**lemma** *poly-mapping-eqI*:

**assumes**  $\bigwedge k. \text{lookup } f \ k = \text{lookup } g \ k$

**shows** *f* = *g*

*<proof>*

**lemma** *poly-mapping-eq-iff*: *a* = *b*  $\longleftrightarrow$  *lookup a* = *lookup b*

*<proof>*

We model the universe of functions being *almost everywhere zero* by means of a separate type '*a*  $\Rightarrow_0$  '*b*. For convenience we provide a suggestive infix syntax which is a variant of the usual function space syntax. Conversion between both types happens through the morphisms

*lookup*

*Abs-poly-mapping*

satisfying

*Abs-poly-mapping* (*lookup* ?*x*) = ?*x*

*finite* {*x. ?f x ≠ 0*}  $\implies$  *lookup* (*Abs-poly-mapping* ?*f*) = ?*f*

Luckily, we have rarely to deal with those low-level morphisms explicitly but rely on Isabelle’s *lifting* package with its method *transfer* and its specification tool *lift-definition*.

**setup-lifting** *type-definition-poly-mapping*

**code-datatype** *Abs-poly-mapping*—FIXME? workaround for preventing *code-astype* setup

'*a*  $\Rightarrow_0$  '*b* serves distinctive purposes:

1. A clever nesting as  $(nat \Rightarrow_0 nat) \Rightarrow_0 'a$  later in theory *MPoly* gives a suitable representation type for polynomials *almost for free*: Interpreting  $nat \Rightarrow_0 nat$  as a mapping from variable identifiers to exponents yields monomials, and the whole type maps monomials to coefficients. Lets call this the *ultimate interpretation*.
2. A further more specialised type isomorphic to  $nat \Rightarrow_0 'a$  is apt to direct implementation using code generation [1].

Note that despite the names *mapping* and *lookup* suggest something implementation-near, it is best to keep  $'a \Rightarrow_0 'b$  as an abstract *algebraic* type providing operations like *addition*, *multiplication* without any notion of key-order, data structures etc. This implementations-specific notions are easily introduced later for particular implementations but do not provide any gain for specifying logical properties of polynomials.

### 78.3 Additive structure

The additive structure covers the usual operations  $0$ ,  $+$  and (unary and binary)  $-$ . Recalling the ultimate interpretation, it is obvious that these have just lift the corresponding operations on values to mappings.

Isabelle has a rich hierarchy of algebraic type classes, and in such situations of pointwise lifting a typical pattern is to have instantiations for a considerable number of type classes.

The operations themselves are specified using *lift-definition*, where the proofs of the *almost everywhere zero* property can be significantly involved.

The *lookup* operation is supposed to be usable explicitly (unless in other situations where the morphisms between types are somehow internal to the *lifting* package). Hence it is good style to provide explicit rewrite rules how *lookup* acts on operations immediately.

**instantiation** *poly-mapping* :: (type, zero) zero  
**begin**

**lift-definition** *zero-poly-mapping* ::  $'a \Rightarrow_0 'b$   
**is**  $\lambda k. 0$   
*<proof>*

**instance** *<proof>*

**end**

**lemma** *Abs-poly-mapping [simp]*: *Abs-poly-mapping* ( $\lambda k. 0$ ) = 0  
*<proof>*

**lemma** *lookup-zero [simp]*: *lookup* 0  $k$  = 0  
*<proof>*

**instantiation** *poly-mapping* :: (*type*, *monoid-add*) *monoid-add*  
**begin**

**lift-definition** *plus-poly-mapping* ::  
 $( 'a \Rightarrow_0 'b ) \Rightarrow ( 'a \Rightarrow_0 'b ) \Rightarrow 'a \Rightarrow_0 'b$   
**is**  $\lambda f1 f2 k. f1 k + f2 k$   
 $\langle proof \rangle$

**instance**  
 $\langle proof \rangle$

**end**

**lemma** *lookup-add*:  $lookup (f + g) k = lookup f k + lookup g k$   
 $\langle proof \rangle$

**instance** *poly-mapping* :: (*type*, *comm-monoid-add*) *comm-monoid-add*  
 $\langle proof \rangle$

**lemma** *lookup-sum*:  $lookup (sum pp X) i = sum (\lambda x. lookup (pp x) i) X$   
 $\langle proof \rangle$

**instantiation** *poly-mapping* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*  
**begin**

**lift-definition** *minus-poly-mapping* ::  $( 'a \Rightarrow_0 'b ) \Rightarrow ( 'a \Rightarrow_0 'b ) \Rightarrow 'a \Rightarrow_0 'b$   
**is**  $\lambda f1 f2 k. f1 k - f2 k$   
 $\langle proof \rangle$

**instance**  
 $\langle proof \rangle$

**end**

**instantiation** *poly-mapping* :: (*type*, *ab-group-add*) *ab-group-add*  
**begin**

**lift-definition** *uminus-poly-mapping* ::  $( 'a \Rightarrow_0 'b ) \Rightarrow 'a \Rightarrow_0 'b$   
**is** *uminus*  
 $\langle proof \rangle$

**instance**  
 $\langle proof \rangle$

**end**

**lemma** *lookup-uminus* [*simp*]:  
 $lookup (- f) k = - lookup f k$   
 ⟨*proof*⟩

**lemma** *lookup-minus*:  
 $lookup (f - g) k = lookup f k - lookup g k$   
 ⟨*proof*⟩

## 78.4 Multiplicative structure

**instantiation** *poly-mapping* :: (*zero*, *zero-neq-one*) *zero-neq-one*  
**begin**

**lift-definition** *one-poly-mapping* :: 'a  $\Rightarrow_0$  'b  
**is**  $\lambda k. 1$  when  $k = 0$   
 ⟨*proof*⟩

**instance**  
 ⟨*proof*⟩

**end**

**lemma** *lookup-one*:  $lookup 1 k = (1 \text{ when } k = 0)$   
 ⟨*proof*⟩

**lemma** *lookup-one-zero* [*simp*]:  
 $lookup 1 0 = 1$   
 ⟨*proof*⟩

**definition** *prod-fun* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a::*monoid-add*  $\Rightarrow$  'b::*semiring-0*  
**where**  
 $prod-fun f1 f2 k = (\sum l. f1 l * (\sum q. (f2 q \text{ when } k = l + q)))$

**lemma** *prod-fun-unfold-prod*:  
**fixes**  $f g :: 'a :: monoid-add \Rightarrow 'b :: semiring-0$   
**assumes** *fin-f*: *finite* { $a. f a \neq 0$ }  
**assumes** *fin-g*: *finite* { $b. g b \neq 0$ }  
**shows**  $prod-fun f g k = (\sum (a, b). f a * g b \text{ when } k = a + b)$   
 ⟨*proof*⟩

**lemma** *finite-prod-fun*:  
**fixes**  $f1 f2 :: 'a :: monoid-add \Rightarrow 'b :: semiring-0$   
**assumes** *fin1*: *finite* { $l. f1 l \neq 0$ }  
**and** *fin2*: *finite* { $q. f2 q \neq 0$ }  
**shows** *finite* { $k. prod-fun f1 f2 k \neq 0$ }  
 ⟨*proof*⟩

**instantiation** *poly-mapping* :: (*monoid-add*, *semiring-0*) *semiring-0*



**begin**

**lift-definition** *times-poly-mapping* :: ( $'a \Rightarrow_0 'b$ )  $\Rightarrow$  ( $'a \Rightarrow_0 'b$ )  $\Rightarrow$   $'a \Rightarrow_0 'b$   
 is *prod-fun*  
 $\langle$ *proof* $\rangle$

**instance**  
 $\langle$ *proof* $\rangle$

**end**

**lemma** *lookup-mult*:  
 $lookup (f * g) k = (\sum l. lookup f l * (\sum q. lookup g q \text{ when } k = l + q))$   
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0*) *comm-semiring-0*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*monoid-add*, *semiring-0-cancel*) *semiring-0-cancel*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*monoid-add*, *semiring-1*) *semiring-1*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-1*) *comm-semiring-1*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*monoid-add*, *semiring-1-cancel*) *semiring-1-cancel*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*monoid-add*, *ring*) *ring*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*comm-monoid-add*, *comm-ring*) *comm-ring*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*monoid-add*, *ring-1*) *ring-1*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*comm-monoid-add*, *comm-ring-1*) *comm-ring-1*  
 $\langle$ *proof* $\rangle$

## 78.5 Single-point mappings

**lift-definition** *single* ::  $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow_0 'b::zero$   
 is  $\lambda k v k'. (v \text{ when } k = k')$

$\langle \text{proof} \rangle$

**lemma** *inj-single* [*iff*]:

$\text{inj } (\text{single } k)$

$\langle \text{proof} \rangle$

**lemma** *lookup-single*:

$\text{lookup } (\text{single } k \ v) \ k' = (v \ \text{when } k = k')$

$\langle \text{proof} \rangle$

**lemma** *lookup-single-eq* [*simp*]:

$\text{lookup } (\text{single } k \ v) \ k = v$

$\langle \text{proof} \rangle$

**lemma** *lookup-single-not-eq*:

$k \neq k' \implies \text{lookup } (\text{single } k \ v) \ k' = 0$

$\langle \text{proof} \rangle$

**lemma** *single-zero* [*simp*]:

$\text{single } k \ 0 = 0$

$\langle \text{proof} \rangle$

**lemma** *single-one* [*simp*]:

$\text{single } 0 \ 1 = 1$

$\langle \text{proof} \rangle$

**lemma** *single-add*:

$\text{single } k \ (a + b) = \text{single } k \ a + \text{single } k \ b$

$\langle \text{proof} \rangle$

**lemma** *single-uminus*:

$\text{single } k \ (- a) = - \text{single } k \ a$

$\langle \text{proof} \rangle$

**lemma** *single-diff*:

$\text{single } k \ (a - b) = \text{single } k \ a - \text{single } k \ b$

$\langle \text{proof} \rangle$

**lemma** *single-numeral* [*simp*]:

$\text{single } 0 \ (\text{numeral } n) = \text{numeral } n$

$\langle \text{proof} \rangle$

**lemma** *lookup-numeral*:

$\text{lookup } (\text{numeral } n) \ k = (\text{numeral } n \ \text{when } k = 0)$

$\langle \text{proof} \rangle$

**lemma** *single-of-nat* [*simp*]:

$\text{single } 0 \ (\text{of-nat } n) = \text{of-nat } n$

$\langle \text{proof} \rangle$

**lemma** *lookup-of-nat*:

*lookup (of-nat n) k = (of-nat n when k = 0)*  
 ⟨proof⟩

**lemma** *of-nat-single*:

*of-nat = single 0 ∘ of-nat*  
 ⟨proof⟩

**lemma** *mult-single*:

*single k a \* single l b = single (k + l) (a \* b)*  
 ⟨proof⟩

**instance** *poly-mapping* :: (*monoid-add*, *semiring-char-0*) *semiring-char-0*

⟨proof⟩

**instance** *poly-mapping* :: (*monoid-add*, *ring-char-0*) *ring-char-0*

⟨proof⟩

**lemma** *single-of-int* [*simp*]:

*single 0 (of-int k) = of-int k*  
 ⟨proof⟩

**lemma** *lookup-of-int*:

*lookup (of-int l) k = (of-int l when k = 0)*  
 ⟨proof⟩

## 78.6 Integral domains

**instance** *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *semiring-no-zero-divisors*)  
*semiring-no-zero-divisors*

The *linorder* constraint is a pragmatic device for the proof — maybe it can be dropped

⟨proof⟩

**instance** *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *ring-no-zero-divisors*)

*ring-no-zero-divisors*

⟨proof⟩

**instance** *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *ring-1-no-zero-divisors*)

*ring-1-no-zero-divisors*

⟨proof⟩

**instance** *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *idom*) *idom*

⟨proof⟩

## 78.7 Mapping order

**instantiation** *poly-mapping* :: (*linorder*, {*zero*, *linorder*}) *linorder*

**begin**

**lift-definition** *less-poly-mapping* :: ( $'a \Rightarrow_0 'b$ )  $\Rightarrow$  ( $'a \Rightarrow_0 'b$ )  $\Rightarrow$  *bool*  
**is** *less-fun*  
 $\langle$ *proof* $\rangle$

**lift-definition** *less-eq-poly-mapping* :: ( $'a \Rightarrow_0 'b$ )  $\Rightarrow$  ( $'a \Rightarrow_0 'b$ )  $\Rightarrow$  *bool*  
**is**  $\lambda f g. \text{less-fun } f g \vee f = g$   
 $\langle$ *proof* $\rangle$

**instance**  $\langle$ *proof* $\rangle$

**end**

**instance** *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imple*,  
*linorder*}) *ordered-ab-semigroup-add*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imple*,  
*cancel-comm-monoid-add*, *linorder*}) *linordered-cancel-ab-semigroup-add*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imple*,  
*cancel-comm-monoid-add*, *linorder*}) *ordered-comm-monoid-add*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imple*,  
*cancel-comm-monoid-add*, *linorder*}) *ordered-cancel-comm-monoid-add*  
 $\langle$ *proof* $\rangle$

**instance** *poly-mapping* :: (*linorder*, *linordered-ab-group-add*) *linordered-ab-group-add*  
 $\langle$ *proof* $\rangle$

For pragmatism we leave out the final elements in the hierarchy: *linordered-ring*, *linordered-ring-strict*, *linordered-idom*; remember that the order instance is a mere technical device, not a deeper algebraic property.

## 78.8 Fundamental mapping notions

**lift-definition** *keys* :: ( $'a \Rightarrow_0 'b::\text{zero}$ )  $\Rightarrow$   $'a$  *set*  
**is**  $\lambda f. \{k. f k \neq 0\}$   $\langle$ *proof* $\rangle$

**lift-definition** *range* :: ( $'a \Rightarrow_0 'b::\text{zero}$ )  $\Rightarrow$   $'b$  *set*  
**is**  $\lambda f :: 'a \Rightarrow 'b. \text{Set.range } f - \{0\}$   $\langle$ *proof* $\rangle$

**lemma** *finite-keys* [*simp*]:  
*finite* (*keys* *f*)  
 $\langle$ *proof* $\rangle$

**lemma** *not-in-keys-iff-lookup-eq-zero*:

$$k \notin \text{keys } f \iff \text{lookup } f \ k = 0$$

*<proof>*

**lemma** *lookup-not-eq-zero-eq-in-keys*:

$$\text{lookup } f \ k \neq 0 \iff k \in \text{keys } f$$

*<proof>*

**lemma** *lookup-eq-zero-in-keys-contradict* [*dest*]:

$$\text{lookup } f \ k = 0 \implies \neg k \in \text{keys } f$$

*<proof>*

**lemma** *finite-range* [*simp*]: *finite* (*Poly-Mapping.range* *p*)

*<proof>*

**lemma** *in-keys-lookup-in-range* [*simp*]:

$$k \in \text{keys } f \implies \text{lookup } f \ k \in \text{range } f$$

*<proof>*

**lemma** *in-keys-iff*:  $x \in (\text{keys } s) = (\text{lookup } s \ x \neq 0)$

*<proof>*

**lemma** *keys-zero* [*simp*]:

$$\text{keys } 0 = \{\}$$

*<proof>*

**lemma** *range-zero* [*simp*]:

$$\text{range } 0 = \{\}$$

*<proof>*

**lemma** *keys-add*:

$$\text{keys } (f + g) \subseteq \text{keys } f \cup \text{keys } g$$

*<proof>*

**lemma** *keys-one* [*simp*]:

$$\text{keys } 1 = \{0\}$$

*<proof>*

**lemma** *range-one* [*simp*]:

$$\text{range } 1 = \{1\}$$

*<proof>*

**lemma** *keys-single* [*simp*]:

$$\text{keys } (\text{single } k \ v) = (\text{if } v = 0 \text{ then } \{\} \text{ else } \{k\})$$

*<proof>*

**lemma** *range-single* [*simp*]:

$$\text{range } (\text{single } k \ v) = (\text{if } v = 0 \text{ then } \{\} \text{ else } \{v\})$$

*<proof>*

**lemma** *keys-mult*:

$keys (f * g) \subseteq \{a + b \mid a \in keys f \wedge b \in keys g\}$   
 ⟨proof⟩

**lemma** *setsum-keys-plus-distrib*:

**assumes** *hom-0*:  $\bigwedge k. f k 0 = 0$

**and** *hom-plus*:  $\bigwedge k. k \in Poly-Mapping.keys p \cup Poly-Mapping.keys q \implies f k (Poly-Mapping.lookup p k + Poly-Mapping.lookup q k) = f k (Poly-Mapping.lookup p k) + f k (Poly-Mapping.lookup q k)$

**shows**

$(\sum_{k \in Poly-Mapping.keys (p + q)}. f k (Poly-Mapping.lookup (p + q) k)) =$   
 $(\sum_{k \in Poly-Mapping.keys p}. f k (Poly-Mapping.lookup p k)) +$   
 $(\sum_{k \in Poly-Mapping.keys q}. f k (Poly-Mapping.lookup q k))$   
 (is ?lhs = ?p + ?q)

⟨proof⟩

## 78.9 Degree

**definition** *degree* ::  $(nat \Rightarrow_0 'a::zero) \Rightarrow nat$

**where**

$degree f = Max (insert 0 (Suc ` keys f))$

**lemma** *degree-zero* [*simp*]:

$degree 0 = 0$   
 ⟨proof⟩

**lemma** *degree-one* [*simp*]:

$degree 1 = 1$   
 ⟨proof⟩

**lemma** *degree-single-zero* [*simp*]:

$degree (single k 0) = 0$   
 ⟨proof⟩

**lemma** *degree-single-not-zero* [*simp*]:

$v \neq 0 \implies degree (single k v) = Suc k$   
 ⟨proof⟩

**lemma** *degree-zero-iff* [*simp*]:

$degree f = 0 \iff f = 0$   
 ⟨proof⟩

**lemma** *degree-greater-zero-in-keys*:

**assumes**  $0 < degree f$

**shows**  $degree f - 1 \in keys f$

⟨proof⟩

**lemma** *in-keys-less-degree*:

$n \in \text{keys } f \implies n < \text{degree } f$   
 ⟨proof⟩

**lemma** *beyond-degree-lookup-zero*:  
 $\text{degree } f \leq n \implies \text{lookup } f \ n = 0$   
 ⟨proof⟩

**lemma** *degree-add*:  
 $\text{degree } (f + g) \leq \max (\text{degree } f) (\text{Poly-Mapping.degree } g)$   
 ⟨proof⟩

**lemma** *sorted-list-of-set-keys*:  
 $\text{sorted-list-of-set } (\text{keys } f) = \text{filter } (\lambda k. k \in \text{keys } f) [0..<\text{degree } f]$  (**is** - = ?r)  
 ⟨proof⟩

## 78.10 Inductive structure

**lift-definition** *update* :: 'a  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\Rightarrow_0$  'b::zero)  $\Rightarrow$  'a  $\Rightarrow_0$  'b  
**is**  $\lambda k \ v \ f. f(k := v)$   
 ⟨proof⟩

**lemma** *update-induct* [case-names const update]:  
**assumes** *const'*: P 0  
**assumes** *update'*:  $\bigwedge f \ a \ b. a \notin \text{keys } f \implies b \neq 0 \implies P \ f \implies P (\text{update } a \ b \ f)$   
**shows** P f  
 ⟨proof⟩

**lemma** *lookup-update*:  
 $\text{lookup } (\text{update } k \ v \ f) \ k' = (\text{if } k = k' \ \text{then } v \ \text{else } \text{lookup } f \ k')$   
 ⟨proof⟩

**lemma** *keys-update*:  
 $\text{keys } (\text{update } k \ v \ f) = (\text{if } v = 0 \ \text{then } \text{keys } f - \{k\} \ \text{else } \text{insert } k \ (\text{keys } f))$   
 ⟨proof⟩

## 78.11 Quasi-functorial structure

**lift-definition** *map* :: ('b::zero  $\Rightarrow$  'c::zero)  
 $\Rightarrow$  ('a  $\Rightarrow_0$  'b)  $\Rightarrow$  ('a  $\Rightarrow_0$  'c::zero)  
**is**  $\lambda g \ f \ k. g (f \ k)$  when  $f \ k \neq 0$   
 ⟨proof⟩

**context**  
**fixes** *f* :: 'b  $\Rightarrow$  'a  
**assumes** *inj-f*: inj f  
**begin**

**lift-definition** *map-key* :: ('a  $\Rightarrow_0$  'c::zero)  $\Rightarrow$  'b  $\Rightarrow_0$  'c  
**is**  $\lambda p. p \circ f$   
 ⟨proof⟩

**end**

**lemma** *map-key-compose*:

**assumes** [*transfer-rule*]: *inj f inj g*

**shows**  $\text{map-key } f \ (\text{map-key } g \ p) = \text{map-key } (g \circ f) \ p$

*<proof>*

**lemma** *map-key-id*:

$\text{map-key } (\lambda x. x) \ p = p$

*<proof>*

**context**

**fixes**  $f :: 'a \Rightarrow 'b$

**assumes** *inj-f* [*transfer-rule*]: *inj f*

**begin**

**lemma** *map-key-map*:

$\text{map-key } f \ (\text{map } g \ p) = \text{map } g \ (\text{map-key } f \ p)$

*<proof>*

**lemma** *map-key-plus*:

$\text{map-key } f \ (p + q) = \text{map-key } f \ p + \text{map-key } f \ q$

*<proof>*

**lemma** *keys-map-key*:

$\text{keys } (\text{map-key } f \ p) = f \ ` \ \text{keys } p$

*<proof>*

**lemma** *map-key-zero* [*simp*]:

$\text{map-key } f \ 0 = 0$

*<proof>*

**lemma** *map-key-single* [*simp*]:

$\text{map-key } f \ (\text{single } (f \ k) \ v) = \text{single } k \ v$

*<proof>*

**end**

**lemma** *mult-map-scale-conv-mult*:  $\text{map } ((* \ s) \ p) = \text{single } 0 \ s * p$

*<proof>*

**lemma** *map-single* [*simp*]:

$(c = 0 \Longrightarrow f \ 0 = 0) \Longrightarrow \text{map } f \ (\text{single } x \ c) = \text{single } x \ (f \ c)$

*<proof>*

**lemma** *map-eq-zero-iff*:  $\text{map } f \ p = 0 \longleftrightarrow (\forall k \in \text{keys } p. f \ (\text{lookup } p \ k) = 0)$

*<proof>*



**78.12 Canonical dense representation of  $\text{nat} \Rightarrow_0 'a$** **abbreviation** *no-trailing-zeros* ::  $'a :: \text{zero list} \Rightarrow \text{bool}$ **where***no-trailing-zeros*  $\equiv$  *no-trailing* ((=) 0)**lift-definition** *nth* ::  $'a \text{ list} \Rightarrow (\text{nat} \Rightarrow_0 'a::\text{zero})$ **is** *nth-default* 0 $\langle \text{proof} \rangle$ The opposite direction is directly specified on (later) type *nat-mapping*.**lemma** *nth-Nil* [*simp*]:*nth* [] = 0 $\langle \text{proof} \rangle$ **lemma** *nth-singleton* [*simp*]:*nth* [v] = *single* 0 v $\langle \text{proof} \rangle$ **lemma** *nth-replicate* [*simp*]:*nth* (*replicate* n 0 @ [v]) = *single* n v $\langle \text{proof} \rangle$ **lemma** *nth-strip-while* [*simp*]:*nth* (*strip-while* ((=) 0) xs) = *nth* xs $\langle \text{proof} \rangle$ **lemma** *nth-strip-while'* [*simp*]:*nth* (*strip-while* ( $\lambda k. k = 0$ ) xs) = *nth* xs $\langle \text{proof} \rangle$ **lemma** *nth-eq-iff*:*nth* xs = *nth* ys  $\longleftrightarrow$  *strip-while* (*HOL.eq* 0) xs = *strip-while* (*HOL.eq* 0) ys $\langle \text{proof} \rangle$ **lemma** *lookup-nth* [*simp*]:*lookup* (*nth* xs) = *nth-default* 0 xs $\langle \text{proof} \rangle$ **lemma** *keys-nth* [*simp*]:*keys* (*nth* xs) = *fst* ' { (n, v)  $\in$  *set* (*enumerate* 0 xs). v  $\neq$  0 } $\langle \text{proof} \rangle$ **lemma** *range-nth* [*simp*]:*range* (*nth* xs) = *set* xs - {0} $\langle \text{proof} \rangle$ **lemma** *degree-nth*:*no-trailing-zeros* xs  $\implies$  *degree* (*nth* xs) = *length* xs $\langle \text{proof} \rangle$

**lemma** *nth-trailing-zeros* [simp]:  
 $nth\ (xs\ @\ replicate\ n\ 0) = nth\ xs$   
 ⟨proof⟩

**lemma** *nth-idem*:  
 $nth\ (List.map\ (lookup\ f)\ [0..<degree\ f]) = f$   
 ⟨proof⟩

**lemma** *nth-idem-bound*:  
**assumes**  $degree\ f \leq n$   
**shows**  $nth\ (List.map\ (lookup\ f)\ [0..<n]) = f$   
 ⟨proof⟩

### 78.13 Canonical sparse representation of $'a \Rightarrow_0 'b$

**lift-definition** *the-value* ::  $('a \times 'b)\ list \Rightarrow 'a \Rightarrow_0 'b::zero$   
**is**  $\lambda xs\ k.$  case map-of xs k of None  $\Rightarrow 0$  | Some v  $\Rightarrow v$   
 ⟨proof⟩

**definition** *items* ::  $('a::linorder \Rightarrow_0 'b::zero) \Rightarrow ('a \times 'b)\ list$   
**where**

$items\ f = List.map\ (\lambda k.\ (k,\ lookup\ f\ k))\ (sorted-list-of-set\ (keys\ f))$

For the canonical sparse representation we provide both directions of morphisms since the specification of ordered association lists in theory *OAL-list* will support arbitrary linear orders *linorder* as keys, not just natural numbers *nat*.

**lemma** *the-value-items* [simp]:  
 $the-value\ (items\ f) = f$   
 ⟨proof⟩

**lemma** *lookup-the-value*:  
 $lookup\ (the-value\ xs)\ k = (case\ map-of\ xs\ k\ of\ None \Rightarrow 0\ |\ Some\ v \Rightarrow v)$   
 ⟨proof⟩

**lemma** *items-the-value*:  
**assumes** *sorted*  $(List.map\ fst\ xs)$  **and** *distinct*  $(List.map\ fst\ xs)$  **and**  $0 \notin snd\ 'set\ xs$   
**shows**  $items\ (the-value\ xs) = xs$   
 ⟨proof⟩

**lemma** *the-value-Nil* [simp]:  
 $the-value\ [] = 0$   
 ⟨proof⟩

**lemma** *the-value-Cons* [simp]:  
 $the-value\ (x\ \# \ xs) = update\ (fst\ x)\ (snd\ x)\ (the-value\ xs)$   
 ⟨proof⟩

**lemma** *items-zero* [*simp*]:

*items 0* = []  
 ⟨*proof*⟩

**lemma** *items-one* [*simp*]:

*items 1* = [(0, 1)]  
 ⟨*proof*⟩

**lemma** *items-single* [*simp*]:

*items (single k v)* = (if *v* = 0 then [] else [(*k*, *v*)])  
 ⟨*proof*⟩

**lemma** *in-set-items-iff* [*simp*]:

$(k, v) \in \text{set } (\text{items } f) \longleftrightarrow k \in \text{keys } f \wedge \text{lookup } f \ k = v$   
 ⟨*proof*⟩

## 78.14 Size estimation

**context**

**fixes** *f* :: 'a ⇒ nat  
**and** *g* :: 'b :: zero ⇒ nat

**begin**

**definition** *poly-mapping-size* :: ('a ⇒<sub>0</sub> 'b) ⇒ nat

**where**

*poly-mapping-size m* = *g 0* + (∑ *k* ∈ *keys m*. *Suc (f k + g (lookup m k))*)

**lemma** *poly-mapping-size-0* [*simp*]:

*poly-mapping-size 0* = *g 0*  
 ⟨*proof*⟩

**lemma** *poly-mapping-size-single* [*simp*]:

*poly-mapping-size (single k v)* = (if *v* = 0 then *g 0* else *g 0 + f k + g v + 1*)  
 ⟨*proof*⟩

**lemma** *keys-less-poly-mapping-size*:

$k \in \text{keys } m \implies f \ k + g \ (\text{lookup } m \ k) < \text{poly-mapping-size } m$   
 ⟨*proof*⟩

**lemma** *lookup-le-poly-mapping-size*:

$g \ (\text{lookup } m \ k) \leq \text{poly-mapping-size } m$   
 ⟨*proof*⟩

**lemma** *poly-mapping-size-estimation*:

$k \in \text{keys } m \implies y \leq f \ k + g \ (\text{lookup } m \ k) \implies y < \text{poly-mapping-size } m$   
 ⟨*proof*⟩

**lemma** *poly-mapping-size-estimation2*:

**assumes**  $v \in \text{range } m$  **and**  $y \leq g \ v$   
**shows**  $y < \text{poly-mapping-size } m$   
 ⟨proof⟩

**end**

**lemma** *poly-mapping-size-one* [simp]:  
 $\text{poly-mapping-size } f \ g \ 1 = g \ 0 + f \ 0 + g \ 1 + 1$   
 ⟨proof⟩

**lemma** *poly-mapping-size-cong* [fundef-cong]:  
 $m = m' \implies g \ 0 = g' \ 0 \implies (\bigwedge k. k \in \text{keys } m' \implies f \ k = f' \ k)$   
 $\implies (\bigwedge v. v \in \text{range } m' \implies g \ v = g' \ v)$   
 $\implies \text{poly-mapping-size } f \ g \ m = \text{poly-mapping-size } f' \ g' \ m'$   
 ⟨proof⟩

**instantiation** *poly-mapping* :: (type, zero) size  
**begin**

**definition** *size = poly-mapping-size* ( $\lambda\cdot. 0$ ) ( $\lambda\cdot. 0$ )

**instance** ⟨proof⟩

**end**

### 78.15 Further mapping operations and properties

It is like in algebra: there are many definitions, some are also used

**lift-definition** *mapp* ::  
 $('a \Rightarrow 'b :: \text{zero} \Rightarrow 'c :: \text{zero}) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'c)$   
**is**  $\lambda f \ p \ k. (\text{if } k \in \text{keys } p \text{ then } f \ k \ (\text{lookup } p \ k) \ \text{else } 0)$   
 ⟨proof⟩

**lemma** *mapp-cong* [fundef-cong]:  
 $\llbracket m = m'; \bigwedge k. k \in \text{keys } m' \implies f \ k \ (\text{lookup } m' \ k) = f' \ k \ (\text{lookup } m' \ k) \rrbracket$   
 $\implies \text{mapp } f \ m = \text{mapp } f' \ m'$   
 ⟨proof⟩

**lemma** *lookup-mapp*:  
 $\text{lookup } (\text{mapp } f \ p) \ k = (f \ k \ (\text{lookup } p \ k) \ \text{when } k \in \text{keys } p)$   
 ⟨proof⟩

**lemma** *keys-mapp-subset*:  $\text{keys } (\text{mapp } f \ p) \subseteq \text{keys } p$   
 ⟨proof⟩

### 78.16 Free Abelian Groups Over a Type

**abbreviation** *frag-of* ::  $'a \Rightarrow 'a \Rightarrow_0 \text{int}$   
**where**  $\text{frag-of } c \equiv \text{Poly-Mapping.single } c \ (1::\text{int})$

**lemma** *lookup-frag-of* [simp]:

$$\text{Poly-Mapping.lookup}(\text{frag-of } c) = (\lambda x. \text{if } x = c \text{ then } 1 \text{ else } 0)$$

⟨proof⟩

**lemma** *frag-of-nonzero* [simp]: *frag-of*  $a \neq 0$

⟨proof⟩

**definition** *frag-cmul* ::  $\text{int} \Rightarrow ('a \Rightarrow_0 \text{int}) \Rightarrow ('a \Rightarrow_0 \text{int})$

**where** *frag-cmul*  $c a = \text{Abs-poly-mapping } (\lambda x. c * \text{Poly-Mapping.lookup } a x)$

**lemma** *frag-cmul-zero* [simp]: *frag-cmul*  $0 x = 0$

⟨proof⟩

**lemma** *frag-cmul-zero2* [simp]: *frag-cmul*  $c 0 = 0$

⟨proof⟩

**lemma** *frag-cmul-one* [simp]: *frag-cmul*  $1 x = x$

⟨proof⟩

**lemma** *frag-cmul-minus-one* [simp]: *frag-cmul*  $(-1) x = -x$

⟨proof⟩

**lemma** *frag-cmul-cmul* [simp]: *frag-cmul*  $c (\text{frag-cmul } d x) = \text{frag-cmul } (c*d) x$

⟨proof⟩

**lemma** *lookup-frag-cmul* [simp]: *poly-mapping.lookup* (*frag-cmul*  $c x$ )  $i = c * \text{poly-mapping.lookup } x i$

⟨proof⟩

**lemma** *minus-frag-cmul* [simp]:  $-\text{frag-cmul } k x = \text{frag-cmul } (-k) x$

⟨proof⟩

**lemma** *keys-frag-of*: *Poly-Mapping.keys*(*frag-of*  $a$ ) =  $\{a\}$

⟨proof⟩

**lemma** *finite-cmul-nonzero*: *finite*  $\{x. c * \text{Poly-Mapping.lookup } a x \neq (0::\text{int})\}$

⟨proof⟩

**lemma** *keys-cmul*: *Poly-Mapping.keys*(*frag-cmul*  $c a$ )  $\subseteq \text{Poly-Mapping.keys } a$

⟨proof⟩

**lemma** *keys-cmul-iff* [iff]:  $i \in \text{Poly-Mapping.keys } (\text{frag-cmul } c x) \iff i \in \text{Poly-Mapping.keys } x \wedge c \neq 0$

⟨proof⟩

**lemma** *keys-minus* [simp]: *Poly-Mapping.keys*( $-a$ ) = *Poly-Mapping.keys*  $a$

⟨proof⟩

**lemma** *keys-diff*:

$$\text{Poly-Mapping.keys}(a - b) \subseteq \text{Poly-Mapping.keys } a \cup \text{Poly-Mapping.keys } b$$

*<proof>*

**lemma** *keys-eq-empty* [simp]:  $\text{Poly-Mapping.keys } c = \{\} \longleftrightarrow c = 0$

*<proof>*

**lemma** *frag-cmul-eq-0-iff* [simp]:  $\text{frag-cmul } k \ c = 0 \longleftrightarrow k=0 \vee c=0$

*<proof>*

**lemma** *frag-of-eq*:  $\text{frag-of } x = \text{frag-of } y \longleftrightarrow x = y$

*<proof>*

**lemma** *frag-cmul-distrib*:  $\text{frag-cmul } (c+d) \ a = \text{frag-cmul } c \ a + \text{frag-cmul } d \ a$

*<proof>*

**lemma** *frag-cmul-distrib2*:  $\text{frag-cmul } c \ (a+b) = \text{frag-cmul } c \ a + \text{frag-cmul } c \ b$

*<proof>*

**lemma** *frag-cmul-diff-distrib*:  $\text{frag-cmul } (a - b) \ c = \text{frag-cmul } a \ c - \text{frag-cmul } b \ c$

*<proof>*

**lemma** *frag-cmul-sum*:

$$\text{frag-cmul } a \ (\text{sum } b \ I) = (\sum i \in I. \text{frag-cmul } a \ (b \ i))$$

*<proof>*

**lemma** *keys-sum*:  $\text{Poly-Mapping.keys}(\text{sum } b \ I) \subseteq (\bigcup i \in I. \text{Poly-Mapping.keys}(b \ i))$

*<proof>*

**definition** *frag-extend* ::  $('b \Rightarrow 'a \Rightarrow_0 \text{int}) \Rightarrow ('b \Rightarrow_0 \text{int}) \Rightarrow 'a \Rightarrow_0 \text{int}$

**where**  $\text{frag-extend } b \ x \equiv (\sum i \in \text{Poly-Mapping.keys } x. \text{frag-cmul } (\text{Poly-Mapping.lookup } x \ i) \ (b \ i))$

**lemma** *frag-extend-0* [simp]:  $\text{frag-extend } b \ 0 = 0$

*<proof>*

**lemma** *frag-extend-of* [simp]:  $\text{frag-extend } f \ (\text{frag-of } a) = f \ a$

*<proof>*

**lemma** *frag-extend-cmul*:

$$\text{frag-extend } f \ (\text{frag-cmul } c \ x) = \text{frag-cmul } c \ (\text{frag-extend } f \ x)$$

*<proof>*

**lemma** *frag-extend-minus*:

$$\text{frag-extend } f \ (- \ x) = - \ (\text{frag-extend } f \ x)$$

*<proof>*

**lemma** *frag-extend-add*:

*frag-extend f (a+b) = (frag-extend f a) + (frag-extend f b)*  
 ⟨proof⟩

**lemma** *frag-extend-diff*:

*frag-extend f (a-b) = (frag-extend f a) - (frag-extend f b)*  
 ⟨proof⟩

**lemma** *frag-extend-sum*:

*finite I  $\implies$  frag-extend f ( $\sum_{i \in I} g i$ ) = sum (frag-extend f o g) I*  
 ⟨proof⟩

**lemma** *frag-extend-eq*:

$(\bigwedge f. f \in \text{Poly-Mapping.keys } c \implies g f = h f) \implies \text{frag-extend } g c = \text{frag-extend } h c$   
 ⟨proof⟩

**lemma** *frag-extend-eq-0*:

$(\bigwedge x. x \in \text{Poly-Mapping.keys } c \implies f x = 0) \implies \text{frag-extend } f c = 0$   
 ⟨proof⟩

**lemma** *keys-frag-extend*:  $\text{Poly-Mapping.keys}(\text{frag-extend } f c) \subseteq (\bigcup x \in \text{Poly-Mapping.keys } c. \text{Poly-Mapping.keys}(f x))$

⟨proof⟩

**lemma** *frag-expansion*:  $a = \text{frag-extend } \text{frag-of } a$

⟨proof⟩

**lemma** *frag-closure-minus-cmul*:

**assumes**  $P 0$  **and**  $P: \bigwedge x y. \llbracket P x; P y \rrbracket \implies P(x - y) P c$   
**shows**  $P(\text{frag-cmul } k c)$

⟨proof⟩

**lemma** *frag-induction* [*consumes 1, case-names zero one diff*]:

**assumes** *supp*:  $\text{Poly-Mapping.keys } c \subseteq S$   
**and** *0*:  $P 0$  **and** *sing*:  $\bigwedge x. x \in S \implies P(\text{frag-of } x)$   
**and** *diff*:  $\bigwedge a b. \llbracket P a; P b \rrbracket \implies P(a - b)$   
**shows**  $P c$

⟨proof⟩

**lemma** *frag-extend-compose*:

*frag-extend f (frag-extend (frag-of o g) c) = frag-extend (f o g) c*  
 ⟨proof⟩

**lemma** *frag-split*:

**fixes**  $c :: 'a \Rightarrow_0 \text{int}$

**assumes**  $\text{Poly-Mapping.keys } c \subseteq S \cup T$

**obtains**  $d e$  **where**  $\text{Poly-Mapping.keys } d \subseteq S \text{ Poly-Mapping.keys } e \subseteq T d + e$

= c  
 ⟨proof⟩

**hide-const** (**open**) *lookup single update keys range map map-key degree nth the-value items foldr mapp*

**end**

## 79 Exponentiation by Squaring

**theory** *Power-By-Squaring*

**imports** *Main*

**begin**

**context**

**fixes**  $f :: 'a \Rightarrow 'a \Rightarrow 'a$

**begin**

**function** *efficient-funpow* ::  $'a \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow 'a$  **where**

*efficient-funpow*  $y\ x\ 0 = y$

| *efficient-funpow*  $y\ x\ (\text{Suc } 0) = f\ x\ y$

|  $n \neq 0 \implies \text{even } n \implies \text{efficient-funpow } y\ x\ n = \text{efficient-funpow } y\ (f\ x\ x)\ (n\ \text{div } 2)$

|  $n \neq 1 \implies \text{odd } n \implies \text{efficient-funpow } y\ x\ n = \text{efficient-funpow } (f\ x\ y)\ (f\ x\ x)\ (n\ \text{div } 2)$

⟨proof⟩

**termination** ⟨proof⟩

**lemma** *efficient-funpow-code* [code]:

*efficient-funpow*  $y\ x\ n =$

(if  $n = 0$  then  $y$

else if  $n = 1$  then  $f\ x\ y$

else if even  $n$  then *efficient-funpow*  $y\ (f\ x\ x)\ (n\ \text{div } 2)$

else *efficient-funpow*  $(f\ x\ y)\ (f\ x\ x)\ (n\ \text{div } 2)$ )

⟨proof⟩

**end**

**lemma** *efficient-funpow-correct*:

**assumes**  $f\text{-assoc}: \bigwedge x\ z. f\ x\ (f\ x\ z) = f\ (f\ x\ x)\ z$

**shows** *efficient-funpow*  $f\ y\ x\ n = (f\ x\ \widehat{\widehat{\ }} n)\ y$

⟨proof⟩

**context** *monoid-mult*

**begin**

**lemma** *power-by-squaring*: *efficient-funpow*  $(*)\ (1 :: 'a) = (\widehat{\widehat{\ }})$

⟨proof⟩



end

end

## 80 Preorders with explicit equivalence relation

theory *Preorder*

imports *Main*

begin

class *preorder-equiv* = *preorder*

begin

**definition** *equiv* :: 'a ⇒ 'a ⇒ bool

where *equiv* x y ⟷ x ≤ y ∧ y ≤ x

**notation**

*equiv* (⟨'(*≈*)⟩) and

*equiv* (⟨(⟨notation=*infix* *≈*)⟩-/ *≈* -) [51, 51] 50)

**lemma** *equivD1*: x ≤ y if x ≈ y

⟨proof⟩

**lemma** *equivD2*: y ≤ x if x ≈ y

⟨proof⟩

**lemma** *equiv-refl* [*iff*]: x ≈ x

⟨proof⟩

**lemma** *equiv-sym*: x ≈ y ⟷ y ≈ x

⟨proof⟩

**lemma** *equiv-trans*: x ≈ y ⟹ y ≈ z ⟹ x ≈ z

⟨proof⟩

**lemma** *equiv-antisym*: x ≤ y ⟹ y ≤ x ⟹ x ≈ y

⟨proof⟩

**lemma** *less-le*: x < y ⟷ x ≤ y ∧ ¬ x ≈ y

⟨proof⟩

**lemma** *le-less*: x ≤ y ⟷ x < y ∨ x ≈ y

⟨proof⟩

**lemma** *le-imp-less-or-equiv*: x ≤ y ⟹ x < y ∨ x ≈ y

⟨proof⟩

**lemma** *less-imp-not-equiv*: x < y ⟹ ¬ x ≈ y

⟨proof⟩

**lemma** *not-equiv-le-trans*:  $\neg a \approx b \implies a \leq b \implies a < b$   
 ⟨proof⟩

**lemma** *le-not-equiv-trans*:  $a \leq b \implies \neg a \approx b \implies a < b$   
 ⟨proof⟩

**lemma** *antisym-conv*:  $y \leq x \implies x \leq y \longleftrightarrow x \approx y$   
 ⟨proof⟩

**end**

⟨ML⟩

**end**

## 81 Additive group operations on product types

**theory** *Product-Plus*  
**imports** *Main*  
**begin**

### 81.1 Operations

**instantiation** *prod* :: (*zero*, *zero*) *zero*  
**begin**

**definition** *zero-prod-def*:  $0 = (0, 0)$

**instance** ⟨proof⟩  
**end**

**instantiation** *prod* :: (*plus*, *plus*) *plus*  
**begin**

**definition** *plus-prod-def*:  
 $x + y = (fst\ x + fst\ y, snd\ x + snd\ y)$

**instance** ⟨proof⟩  
**end**

**instantiation** *prod* :: (*minus*, *minus*) *minus*  
**begin**

**definition** *minus-prod-def*:  
 $x - y = (fst\ x - fst\ y, snd\ x - snd\ y)$

**instance** ⟨proof⟩

**end**

**instantiation** *prod* :: (*uminus*, *uminus*) *uminus*  
**begin**

**definition** *uminus-prod-def*:  
 $- x = (- \text{fst } x, - \text{snd } x)$

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

**lemma** *fst-zero* [*simp*]:  $\text{fst } 0 = 0$   
 $\langle \text{proof} \rangle$

**lemma** *snd-zero* [*simp*]:  $\text{snd } 0 = 0$   
 $\langle \text{proof} \rangle$

**lemma** *fst-add* [*simp*]:  $\text{fst } (x + y) = \text{fst } x + \text{fst } y$   
 $\langle \text{proof} \rangle$

**lemma** *snd-add* [*simp*]:  $\text{snd } (x + y) = \text{snd } x + \text{snd } y$   
 $\langle \text{proof} \rangle$

**lemma** *fst-diff* [*simp*]:  $\text{fst } (x - y) = \text{fst } x - \text{fst } y$   
 $\langle \text{proof} \rangle$

**lemma** *snd-diff* [*simp*]:  $\text{snd } (x - y) = \text{snd } x - \text{snd } y$   
 $\langle \text{proof} \rangle$

**lemma** *fst-uminus* [*simp*]:  $\text{fst } (- x) = - \text{fst } x$   
 $\langle \text{proof} \rangle$

**lemma** *snd-uminus* [*simp*]:  $\text{snd } (- x) = - \text{snd } x$   
 $\langle \text{proof} \rangle$

**lemma** *add-Pair* [*simp*]:  $(a, b) + (c, d) = (a + c, b + d)$   
 $\langle \text{proof} \rangle$

**lemma** *diff-Pair* [*simp*]:  $(a, b) - (c, d) = (a - c, b - d)$   
 $\langle \text{proof} \rangle$

**lemma** *uminus-Pair* [*simp*, *code*]:  $-(a, b) = (- a, - b)$   
 $\langle \text{proof} \rangle$

## 81.2 Class instances

**instance** *prod* :: (*semigroup-add*, *semigroup-add*) *semigroup-add*  
 $\langle \text{proof} \rangle$

```

instance prod :: (ab-semigroup-add, ab-semigroup-add) ab-semigroup-add
  ⟨proof⟩

instance prod :: (monoid-add, monoid-add) monoid-add
  ⟨proof⟩

instance prod :: (comm-monoid-add, comm-monoid-add) comm-monoid-add
  ⟨proof⟩

instance prod :: (cancel-semigroup-add, cancel-semigroup-add) cancel-semigroup-add
  ⟨proof⟩

instance prod :: (cancel-ab-semigroup-add, cancel-ab-semigroup-add) cancel-ab-semigroup-add
  ⟨proof⟩

instance prod :: (cancel-comm-monoid-add, cancel-comm-monoid-add) cancel-comm-monoid-add
  ⟨proof⟩

instance prod :: (group-add, group-add) group-add
  ⟨proof⟩

instance prod :: (ab-group-add, ab-group-add) ab-group-add
  ⟨proof⟩

lemma fst-sum: fst (∑ x∈A. f x) = (∑ x∈A. fst (f x))
  ⟨proof⟩

lemma snd-sum: snd (∑ x∈A. f x) = (∑ x∈A. snd (f x))
  ⟨proof⟩

lemma sum-prod: (∑ x∈A. (f x, g x)) = (∑ x∈A. f x, ∑ x∈A. g x)
  ⟨proof⟩

end

```

## 82 Roots of real quadratics

```

theory Quadratic-Discriminant
imports Complex-Main
begin

```

```

definition discrim :: real ⇒ real ⇒ real ⇒ real
  where discrim a b c ≡ b2 - 4 * a * c

```

```

lemma complete-square:
  a ≠ 0 ⇒ a * x2 + b * x + c = 0 ⇔ (2 * a * x + b)2 = discrim a b c
  ⟨proof⟩

```

```

lemma discriminant-negative:

```

**fixes**  $a\ b\ c\ x :: \text{real}$   
**assumes**  $a \neq 0$   
**and**  $\text{discrim } a\ b\ c < 0$   
**shows**  $a * x^2 + b * x + c \neq 0$   
 <proof>

**lemma** *plus-or-minus-sqrt*:  
**fixes**  $x\ y :: \text{real}$   
**assumes**  $y \geq 0$   
**shows**  $x^2 = y \longleftrightarrow x = \text{sqrt } y \vee x = - \text{sqrt } y$   
 <proof>

**lemma** *divide-non-zero*:  
**fixes**  $x\ y\ z :: \text{real}$   
**assumes**  $x \neq 0$   
**shows**  $x * y = z \longleftrightarrow y = z / x$   
 <proof>

**lemma** *discriminant-nonneg*:  
**fixes**  $a\ b\ c\ x :: \text{real}$   
**assumes**  $a \neq 0$   
**and**  $\text{discrim } a\ b\ c \geq 0$   
**shows**  $a * x^2 + b * x + c = 0 \longleftrightarrow$   
 $x = (-b + \text{sqrt } (\text{discrim } a\ b\ c)) / (2 * a) \vee$   
 $x = (-b - \text{sqrt } (\text{discrim } a\ b\ c)) / (2 * a)$   
 <proof>

**lemma** *discriminant-zero*:  
**fixes**  $a\ b\ c\ x :: \text{real}$   
**assumes**  $a \neq 0$   
**and**  $\text{discrim } a\ b\ c = 0$   
**shows**  $a * x^2 + b * x + c = 0 \longleftrightarrow x = -b / (2 * a)$   
 <proof>

**theorem** *discriminant-iff*:  
**fixes**  $a\ b\ c\ x :: \text{real}$   
**assumes**  $a \neq 0$   
**shows**  $a * x^2 + b * x + c = 0 \longleftrightarrow$   
 $\text{discrim } a\ b\ c \geq 0 \wedge$   
 $(x = (-b + \text{sqrt } (\text{discrim } a\ b\ c)) / (2 * a) \vee$   
 $x = (-b - \text{sqrt } (\text{discrim } a\ b\ c)) / (2 * a))$   
 <proof>

**lemma** *discriminant-nonneg-ex*:  
**fixes**  $a\ b\ c :: \text{real}$   
**assumes**  $a \neq 0$   
**and**  $\text{discrim } a\ b\ c \geq 0$   
**shows**  $\exists x. a * x^2 + b * x + c = 0$   
 <proof>

```

lemma discriminant-pos-ex:
  fixes a b c :: real
  assumes a ≠ 0
    and discrim a b c > 0
  shows ∃ x y. x ≠ y ∧ a * x2 + b * x + c = 0 ∧ a * y2 + b * y + c = 0
  ⟨proof⟩

```

```

lemma discriminant-pos-distinct:
  fixes a b c x :: real
  assumes a ≠ 0
    and discrim a b c > 0
  shows ∃ y. x ≠ y ∧ a * y2 + b * y + c = 0
  ⟨proof⟩

```

```

lemma Rats-solution-QE:
  assumes a ∈ ℚ b ∈ ℚ a ≠ 0
    and a*x2 + b*x + c = 0
    and sqrt (discrim a b c) ∈ ℚ
  shows x ∈ ℚ
  ⟨proof⟩

```

```

lemma Rats-solution-QE-converse:
  assumes a ∈ ℚ b ∈ ℚ
    and a*x2 + b*x + c = 0
    and x ∈ ℚ
  shows sqrt (discrim a b c) ∈ ℚ
  ⟨proof⟩

```

end

## 83 Pretty syntax for Quotient operations

```

theory Quotient-Syntax
imports Main
begin

```

```

notation
  rel-conj (infixr ⟨OOO⟩ 75) and
  map-fun (infixr ⟨---->⟩ 55) and
  rel-fun (infixr ⟨===>⟩ 55)

```

end

## 84 Quotient infrastructure for the set type

```

theory Quotient-Set
imports Quotient-Syntax

```

begin

### 84.1 Contravariant set map (vimage) and set relator, rules for the Quotient package

**definition**  $rel\text{-}vset\ R\ xs\ ys \equiv \forall x\ y. R\ x\ y \longrightarrow x \in xs \longleftrightarrow y \in ys$

**lemma**  $rel\text{-}vset\text{-}eq$  [*id-simps*]:

$rel\text{-}vset\ (=) = (=)$   
*<proof>*

**lemma**  $rel\text{-}vset\text{-}equivp$ :

**assumes**  $e: equivp\ R$   
**shows**  $rel\text{-}vset\ R\ xs\ ys \longleftrightarrow xs = ys \wedge (\forall x\ y. x \in xs \longrightarrow R\ x\ y \longrightarrow y \in xs)$   
*<proof>*

**lemma**  $set\text{-}quotient$  [*quot-thm*]:

**assumes**  $Quotient3\ R\ Abs\ Rep$   
**shows**  $Quotient3\ (rel\text{-}vset\ R)\ (vimage\ Rep)\ (vimage\ Abs)$   
*<proof>*

**declare**  $[[mapQ3\ set = (rel\text{-}vset,\ set\text{-}quotient)]]$

**lemma**  $empty\text{-}set\text{-}rsp$ [*quot-respect*]:

$rel\text{-}vset\ R\ \{\}\ \{\}$   
*<proof>*

**lemma**  $collect\text{-}rsp$ [*quot-respect*]:

**assumes**  $Quotient3\ R\ Abs\ Rep$   
**shows**  $((R\ ==> (=)) ==> rel\text{-}vset\ R)\ Collect\ Collect$   
*<proof>*

**lemma**  $collect\text{-}prs$ [*quot-preserve*]:

**assumes**  $Quotient3\ R\ Abs\ Rep$   
**shows**  $((Abs\ ----> id)\ ----> (-'\ Rep)\ Collect = Collect)$   
*<proof>*

**lemma**  $union\text{-}rsp$ [*quot-respect*]:

**assumes**  $Quotient3\ R\ Abs\ Rep$   
**shows**  $(rel\text{-}vset\ R\ ==> rel\text{-}vset\ R\ ==> rel\text{-}vset\ R)\ (U)\ (U)$   
*<proof>*

**lemma**  $union\text{-}prs$ [*quot-preserve*]:

**assumes**  $Quotient3\ R\ Abs\ Rep$   
**shows**  $((-\ '\ Abs\ ----> (-'\ Abs\ ----> (-'\ Rep)\ (U) = (U))$   
*<proof>*

**lemma**  $diff\text{-}rsp$ [*quot-respect*]:

**assumes**  $Quotient3\ R\ Abs\ Rep$

**shows** ( $rel\text{-}vset\ R\ ==\!>\ rel\text{-}vset\ R\ ==\!>\ rel\text{-}vset\ R$ )  $(-)$   $(-)$   
 $\langle proof \rangle$

**lemma** *diff-prs*[*quot-preserve*]:  
**assumes** *Quotient3* *R* *Abs* *Rep*  
**shows**  $((-)\ Abs\ \text{----}\>\ (-)\ Abs\ \text{----}\>\ (-)\ Rep)$   $(-)$   $=$   $(-)$   
 $\langle proof \rangle$

**lemma** *inter-rsp*[*quot-respect*]:  
**assumes** *Quotient3* *R* *Abs* *Rep*  
**shows** ( $rel\text{-}vset\ R\ ==\!>\ rel\text{-}vset\ R\ ==\!>\ rel\text{-}vset\ R$ )  $(\cap)$   $(\cap)$   
 $\langle proof \rangle$

**lemma** *inter-prs*[*quot-preserve*]:  
**assumes** *Quotient3* *R* *Abs* *Rep*  
**shows**  $((-)\ Abs\ \text{----}\>\ (-)\ Abs\ \text{----}\>\ (-)\ Rep)$   $(\cap)$   $=$   $(\cap)$   
 $\langle proof \rangle$

**lemma** *mem-prs*[*quot-preserve*]:  
**assumes** *Quotient3* *R* *Abs* *Rep*  
**shows** ( $Rep\ \text{----}\>\ (-)\ Abs\ \text{----}\>\ id$ )  $(\in)$   $=$   $(\in)$   
 $\langle proof \rangle$

**lemma** *mem-rsp*[*quot-respect*]:  
**shows** ( $R\ ==\!>\ rel\text{-}vset\ R\ ==\!>\ (=)$ )  $(\in)$   $(\in)$   
 $\langle proof \rangle$

end

## 85 Quotient infrastructure for the product type

**theory** *Quotient-Product*  
**imports** *Quotient-Syntax*  
**begin**

### 85.1 Rules for the Quotient package

**lemma** *map-prod-id* [*id-simps*]:  
**shows**  $map\text{-}prod\ id\ id = id$   
 $\langle proof \rangle$

**lemma** *rel-prod-eq* [*id-simps*]:  
**shows**  $rel\text{-}prod\ (=)\ (=)\ =\ (=)$   
 $\langle proof \rangle$

**lemma** *prod-equivp* [*quot-equiv*]:  
**assumes** *equivp* *R1*  
**assumes** *equivp* *R2*  
**shows**  $equivp\ (rel\text{-}prod\ R1\ R2)$



*<proof>*

**lemma** *prod-quotient* [*quot-thm*]:

**assumes** *Quotient3 R1 Abs1 Rep1*

**assumes** *Quotient3 R2 Abs2 Rep2*

**shows** *Quotient3 (rel-prod R1 R2) (map-prod Abs1 Abs2) (map-prod Rep1 Rep2)*

*<proof>*

**declare** [[*mapQ3 prod = (rel-prod, prod-quotient)*]]

**lemma** *Pair-rsp* [*quot-respect*]:

**assumes** *q1: Quotient3 R1 Abs1 Rep1*

**assumes** *q2: Quotient3 R2 Abs2 Rep2*

**shows** (*R1 ===> R2 ===> rel-prod R1 R2*) *Pair Pair*

*<proof>*

**lemma** *Pair-prs* [*quot-preserve*]:

**assumes** *q1: Quotient3 R1 Abs1 Rep1*

**assumes** *q2: Quotient3 R2 Abs2 Rep2*

**shows** (*Rep1 ----> Rep2 ----> (map-prod Abs1 Abs2)*) *Pair = Pair*

*<proof>*

**lemma** *fst-rsp* [*quot-respect*]:

**assumes** *Quotient3 R1 Abs1 Rep1*

**assumes** *Quotient3 R2 Abs2 Rep2*

**shows** (*rel-prod R1 R2 ===> R1*) *fst fst*

*<proof>*

**lemma** *fst-prs* [*quot-preserve*]:

**assumes** *q1: Quotient3 R1 Abs1 Rep1*

**assumes** *q2: Quotient3 R2 Abs2 Rep2*

**shows** (*map-prod Rep1 Rep2 ----> Abs1*) *fst = fst*

*<proof>*

**lemma** *snd-rsp* [*quot-respect*]:

**assumes** *Quotient3 R1 Abs1 Rep1*

**assumes** *Quotient3 R2 Abs2 Rep2*

**shows** (*rel-prod R1 R2 ===> R2*) *snd snd*

*<proof>*

**lemma** *snd-prs* [*quot-preserve*]:

**assumes** *q1: Quotient3 R1 Abs1 Rep1*

**assumes** *q2: Quotient3 R2 Abs2 Rep2*

**shows** (*map-prod Rep1 Rep2 ----> Abs2*) *snd = snd*

*<proof>*

**lemma** *case-prod-rsp* [*quot-respect*]:

**shows** (*(R1 ===> R2 ===> (=)) ===> (rel-prod R1 R2) ===> (=)*)  
*case-prod case-prod*

*<proof>*

**lemma** *split-prs* [*quot-preserve*]:

**assumes** *q1*: *Quotient3 R1 Abs1 Rep1*

**and** *q2*: *Quotient3 R2 Abs2 Rep2*

**shows**  $((Abs1 \text{ ----> } Abs2 \text{ ----> } id) \text{ ----> } map\text{-prod } Rep1 \text{ Rep2} \text{ ----> } id)$   
*case-prod* = *case-prod*

*<proof>*

**lemma** [*quot-respect*]:

**shows**  $((R2 \text{ =====> } R2 \text{ =====> } (=)) \text{ =====> } (R1 \text{ =====> } R1 \text{ =====> } (=)) \text{ =====> } rel\text{-prod } R2 \text{ R1} \text{ =====> } rel\text{-prod } R2 \text{ R1} \text{ =====> } (=)) \text{ rel-prod } rel\text{-prod}$

*<proof>*

**lemma** [*quot-preserve*]:

**assumes** *q1*: *Quotient3 R1 abs1 rep1*

**and** *q2*: *Quotient3 R2 abs2 rep2*

**shows**  $((abs1 \text{ ----> } abs1 \text{ ----> } id) \text{ ----> } (abs2 \text{ ----> } abs2 \text{ ----> } id) \text{ ----> } map\text{-prod } rep1 \text{ rep2} \text{ ----> } map\text{-prod } rep1 \text{ rep2} \text{ ----> } id) \text{ rel-prod} = \text{rel-prod}$

*<proof>*

**lemma** [*quot-preserve*]:

**shows**  $(rel\text{-prod } ((rep1 \text{ ----> } rep1 \text{ ----> } id) \text{ R1}) ((rep2 \text{ ----> } rep2 \text{ ----> } id) \text{ R2})$

$(l1, l2) (r1, r2)) = (R1 (rep1 l1) (rep1 r1) \wedge R2 (rep2 l2) (rep2 r2))$

*<proof>*

**declare** *prod.inject*[*quot-preserve*]

**end**

## 86 Quotient infrastructure for the option type

**theory** *Quotient-Option*

**imports** *Quotient-Syntax*

**begin**

### 86.1 Rules for the Quotient package

**lemma** *rel-option-map1*:

*rel-option* *R* (*map-option* *f* *x*) *y*  $\longleftrightarrow$  *rel-option* ( $\lambda x. R (f x)$ ) *x* *y*

*<proof>*

**lemma** *rel-option-map2*:

*rel-option* *R* *x* (*map-option* *f* *y*)  $\longleftrightarrow$  *rel-option* ( $\lambda x y. R x (f y)$ ) *x* *y*

*<proof>*

**declare**

```
map-option.id [id-simps]
option.rel-eq [id-simps]
```

**lemma** *reflp-rel-option*:  
 $reflp\ R \implies reflp\ (rel-option\ R)$   
 ⟨proof⟩

**lemma** *option-symp*:  
 $symp\ R \implies symp\ (rel-option\ R)$   
 ⟨proof⟩

**lemma** *option-transp*:  
 $transp\ R \implies transp\ (rel-option\ R)$   
 ⟨proof⟩

**lemma** *option-equivp* [quot-equiv]:  
 $equivp\ R \implies equivp\ (rel-option\ R)$   
 ⟨proof⟩

**lemma** *option-quotient* [quot-thm]:  
 assumes  $Quotient3\ R\ Abs\ Rep$   
 shows  $Quotient3\ (rel-option\ R)\ (map-option\ Abs)\ (map-option\ Rep)$   
 ⟨proof⟩

**declare** [[ $mapQ3\ option = (rel-option, option-quotient)$ ]]

**lemma** *option-None-rsp* [quot-respect]:  
 assumes  $q: Quotient3\ R\ Abs\ Rep$   
 shows  $rel-option\ R\ None\ None$   
 ⟨proof⟩

**lemma** *option-Some-rsp* [quot-respect]:  
 assumes  $q: Quotient3\ R\ Abs\ Rep$   
 shows  $(R\ ===> rel-option\ R)\ Some\ Some$   
 ⟨proof⟩

**lemma** *option-None-prs* [quot-preserve]:  
 assumes  $q: Quotient3\ R\ Abs\ Rep$   
 shows  $map-option\ Abs\ None = None$   
 ⟨proof⟩

**lemma** *option-Some-prs* [quot-preserve]:  
 assumes  $q: Quotient3\ R\ Abs\ Rep$   
 shows  $(Rep\ ----> map-option\ Abs)\ Some = Some$   
 ⟨proof⟩

**end**

## 87 Quotient infrastructure for the list type

```
theory Quotient-List
imports Quotient-Set Quotient-Product Quotient-Option
begin
```

### 87.1 Rules for the Quotient package

```
lemma map-id [id-simps]:
```

```
  map id = id
  ⟨proof⟩
```

```
lemma list-all2-eq [id-simps]:
```

```
  list-all2 (=) = (=)
  ⟨proof⟩
```

```
lemma reflp-list-all2:
```

```
  assumes reflp R
  shows reflp (list-all2 R)
  ⟨proof⟩
```

```
lemma list-symp:
```

```
  assumes symp R
  shows symp (list-all2 R)
  ⟨proof⟩
```

```
lemma list-transp:
```

```
  assumes transp R
  shows transp (list-all2 R)
  ⟨proof⟩
```

```
lemma list-equivp [quot-equiv]:
```

```
  equivp R  $\implies$  equivp (list-all2 R)
  ⟨proof⟩
```

```
lemma list-quotient3 [quot-thm]:
```

```
  assumes Quotient3 R Abs Rep
  shows Quotient3 (list-all2 R) (map Abs) (map Rep)
  ⟨proof⟩
```

```
declare [[mapQ3 list = (list-all2, list-quotient3)]]
```

```
lemma cons-prs [quot-preserve]:
```

```
  assumes q: Quotient3 R Abs Rep
  shows (Rep ----> (map Rep) ----> (map Abs)) (#) = (#)
  ⟨proof⟩
```

```
lemma cons-rsp [quot-respect]:
```

```
  assumes q: Quotient3 R Abs Rep
  shows (R ===> list-all2 R ===> list-all2 R) (#) (#)
```

*<proof>*

**lemma** *nil-prs* [*quot-preserve*]:

**assumes** *q*: *Quotient3 R Abs Rep*

**shows** *map Abs [] = []*

*<proof>*

**lemma** *nil-rsp* [*quot-respect*]:

**assumes** *q*: *Quotient3 R Abs Rep*

**shows** *list-all2 R [] []*

*<proof>*

**lemma** *map-prs-aux*:

**assumes** *a*: *Quotient3 R1 abs1 rep1*

**and** *b*: *Quotient3 R2 abs2 rep2*

**shows** *(map abs2) (map ((abs1 ----> rep2) f) (map rep1 l)) = map f l*

*<proof>*

**lemma** *map-prs* [*quot-preserve*]:

**assumes** *a*: *Quotient3 R1 abs1 rep1*

**and** *b*: *Quotient3 R2 abs2 rep2*

**shows** *((abs1 ----> rep2) ----> (map rep1) ----> (map abs2)) map = map*

**and** *((abs1 ----> id) ----> map rep1 ----> id) map = map*

*<proof>*

**lemma** *map-rsp* [*quot-respect*]:

**assumes** *q1*: *Quotient3 R1 Abs1 Rep1*

**and** *q2*: *Quotient3 R2 Abs2 Rep2*

**shows** *((R1 ===> R2) ===> (list-all2 R1) ===> list-all2 R2) map map*

**and** *((R1 ===> (=)) ===> (list-all2 R1) ===> (=)) map map*

*<proof>*

**lemma** *foldr-prs-aux*:

**assumes** *a*: *Quotient3 R1 abs1 rep1*

**and** *b*: *Quotient3 R2 abs2 rep2*

**shows** *abs2 (foldr ((abs1 ----> abs2 ----> rep2) f) (map rep1 l) (rep2 e)) = foldr f l e*

*<proof>*

**lemma** *foldr-prs* [*quot-preserve*]:

**assumes** *a*: *Quotient3 R1 abs1 rep1*

**and** *b*: *Quotient3 R2 abs2 rep2*

**shows** *((abs1 ----> abs2 ----> rep2) ----> (map rep1) ----> rep2 ----> abs2) foldr = foldr*

*<proof>*

**lemma** *foldl-prs-aux*:

**assumes** *a*: *Quotient3 R1 abs1 rep1*

**and** *b*: *Quotient3 R2 abs2 rep2*

**shows**  $abs1$  (foldl (( $abs1 \dashrightarrow abs2 \dashrightarrow rep1$ )  $f$ ) ( $rep1$   $e$ ) ( $map$   $rep2$   $l$ )) =  
 foldl  $f$   $e$   $l$   
 ⟨proof⟩

**lemma** *foldl-prs* [*quot-preserve*]:

**assumes**  $a$ : *Quotient3*  $R1$   $abs1$   $rep1$   
**and**  $b$ : *Quotient3*  $R2$   $abs2$   $rep2$   
**shows** (( $abs1 \dashrightarrow abs2 \dashrightarrow rep1$ )  $\dashrightarrow rep1 \dashrightarrow (map$   $rep2)$   $\dashrightarrow$   
 $abs1$ ) foldl = foldl  
 ⟨proof⟩

**lemma** *foldl-rsp*[*quot-respect*]:

**assumes**  $q1$ : *Quotient3*  $R1$   $Abs1$   $Rep1$   
**and**  $q2$ : *Quotient3*  $R2$   $Abs2$   $Rep2$   
**shows** (( $R1 \equiv R2 \equiv R1$ )  $\equiv R1 \equiv list-all2$   $R2 \equiv R1$ )  
 foldl foldl  
 ⟨proof⟩

**lemma** *foldr-rsp*[*quot-respect*]:

**assumes**  $q1$ : *Quotient3*  $R1$   $Abs1$   $Rep1$   
**and**  $q2$ : *Quotient3*  $R2$   $Abs2$   $Rep2$   
**shows** (( $R1 \equiv R2 \equiv R2$ )  $\equiv list-all2$   $R1 \equiv R2 \equiv R2$ )  
 foldr foldr  
 ⟨proof⟩

**lemma** *list-all2-rsp*:

**assumes**  $r$ :  $\forall x y. R x y \longrightarrow (\forall a b. R a b \longrightarrow S x a = T y b)$   
**and**  $l1$ : *list-all2*  $R$   $x$   $y$   
**and**  $l2$ : *list-all2*  $R$   $a$   $b$   
**shows** *list-all2*  $S$   $x$   $a$  = *list-all2*  $T$   $y$   $b$   
 ⟨proof⟩

**lemma** [*quot-respect*]:

(( $R \equiv R \equiv (=)$ )  $\equiv list-all2$   $R \equiv list-all2$   $R \equiv (=)$ )  
*list-all2* *list-all2*  
 ⟨proof⟩

**lemma** [*quot-preserve*]:

**assumes**  $a$ : *Quotient3*  $R$   $abs1$   $rep1$   
**shows** (( $abs1 \dashrightarrow abs1 \dashrightarrow id$ )  $\dashrightarrow map$   $rep1 \dashrightarrow map$   $rep1 \dashrightarrow$   
 $id$ ) *list-all2* = *list-all2*  
 ⟨proof⟩

**lemma** [*quot-preserve*]:

**assumes**  $a$ : *Quotient3*  $R$   $abs1$   $rep1$   
**shows** (*list-all2* (( $rep1 \dashrightarrow rep1 \dashrightarrow id$ )  $R$ )  $l$   $m$ ) = ( $l = m$ )  
 ⟨proof⟩

**lemma** *list-all2-find-element*:

**assumes**  $a: x \in \text{set } a$   
**and**  $b: \text{list-all2 } R \ a \ b$   
**shows**  $\exists y. (y \in \text{set } b \wedge R \ x \ y)$   
 $\langle \text{proof} \rangle$

**lemma** *list-all2-refl*:  
**assumes**  $a: \bigwedge x \ y. R \ x \ y = (R \ x = R \ y)$   
**shows**  $\text{list-all2 } R \ x \ x$   
 $\langle \text{proof} \rangle$

**end**

## 88 Quotient infrastructure for the sum type

**theory** *Quotient-Sum*  
**imports** *Quotient-Syntax*  
**begin**

### 88.1 Rules for the Quotient package

**lemma** *rel-sum-map1*:  
 $\text{rel-sum } R1 \ R2 \ (\text{map-sum } f1 \ f2 \ x) \ y \longleftrightarrow \text{rel-sum } (\lambda x. R1 \ (f1 \ x)) \ (\lambda x. R2 \ (f2 \ x))$   
 $x \ y$   
 $\langle \text{proof} \rangle$

**lemma** *rel-sum-map2*:  
 $\text{rel-sum } R1 \ R2 \ x \ (\text{map-sum } f1 \ f2 \ y) \longleftrightarrow \text{rel-sum } (\lambda x \ y. R1 \ x \ (f1 \ y)) \ (\lambda x \ y. R2 \ x \ (f2 \ y))$   
 $x \ y$   
 $\langle \text{proof} \rangle$

**lemma** *map-sum-id* [*id-simps*]:  
 $\text{map-sum } id \ id = id$   
 $\langle \text{proof} \rangle$

**lemma** *rel-sum-eq* [*id-simps*]:  
 $\text{rel-sum } (=) \ (=) = (=)$   
 $\langle \text{proof} \rangle$

**lemma** *reflp-rel-sum*:  
 $\text{reflp } R1 \implies \text{reflp } R2 \implies \text{reflp } (\text{rel-sum } R1 \ R2)$   
 $\langle \text{proof} \rangle$

**lemma** *sum-symp*:  
 $\text{symp } R1 \implies \text{symp } R2 \implies \text{symp } (\text{rel-sum } R1 \ R2)$   
 $\langle \text{proof} \rangle$

**lemma** *sum-transp*:  
 $\text{transp } R1 \implies \text{transp } R2 \implies \text{transp } (\text{rel-sum } R1 \ R2)$   
 $\langle \text{proof} \rangle$

```

lemma sum-equivp [quot-equiv]:
  equivp R1  $\implies$  equivp R2  $\implies$  equivp (rel-sum R1 R2)
  <proof>

lemma sum-quotient [quot-thm]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows Quotient3 (rel-sum R1 R2) (map-sum Abs1 Abs2) (map-sum Rep1 Rep2)
  <proof>

declare [[mapQ3 sum = (rel-sum, sum-quotient)]]

lemma sum-Inl-rsp [quot-respect]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (R1  $\implies$  rel-sum R1 R2) Inl Inl
  <proof>

lemma sum-Inr-rsp [quot-respect]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (R2  $\implies$  rel-sum R1 R2) Inr Inr
  <proof>

lemma sum-Inl-prs [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (Rep1  $\dashrightarrow$  map-sum Abs1 Abs2) Inl = Inl
  <proof>

lemma sum-Inr-prs [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (Rep2  $\dashrightarrow$  map-sum Abs1 Abs2) Inr = Inr
  <proof>

end

```

## 89 Quotient types

```

theory Quotient-Type
imports Main
begin

```

We introduce the notion of quotient types over equivalence relations via type classes.



### 89.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations  $\sim :: 'a \Rightarrow 'a \Rightarrow bool$ .

```

class equiv =
  fixes equiv :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool (infixl  $\langle \sim \rangle$  50)

class equiv = equiv +
  assumes equiv-refl [intro]:  $x \sim x$ 
  and equiv-trans [trans]:  $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$ 
  and equiv-sym [sym]:  $x \sim y \Longrightarrow y \sim x$ 
begin

lemma equiv-not-sym [sym]:  $\neg x \sim y \Longrightarrow \neg y \sim x$ 
<proof>

lemma not-equiv-trans1 [trans]:  $\neg x \sim y \Longrightarrow y \sim z \Longrightarrow \neg x \sim z$ 
<proof>

lemma not-equiv-trans2 [trans]:  $x \sim y \Longrightarrow \neg y \sim z \Longrightarrow \neg x \sim z$ 
<proof>

end

The quotient type  $'a \text{ quot}$  consists of all equivalence classes over elements
of the base type  $'a$ .

definition (in equiv) quot =  $\{\{x. a \sim x\} \mid a. True\}$ 

typedef (overloaded) 'a quot = quot :: 'a::equiv set set
<proof>

lemma quotI [intro]:  $\{x. a \sim x\} \in \text{quot}$ 
<proof>

lemma quotE [elim]:
  assumes  $R \in \text{quot}$ 
  obtains  $a$  where  $R = \{x. a \sim x\}$ 
<proof>

Abstracted equivalence classes are the canonical representation of ele-
ments of a quotient type.

definition class :: 'a::equiv  $\Rightarrow$  'a quot ( $\langle \langle \text{open-block notation} = \langle \text{mixfix class} \rangle \rangle [-] \rangle$ )
  where  $[a] = \text{Abs-quot } \{x. a \sim x\}$ 

theorem quot-exhaust:  $\exists a. A = [a]$ 
<proof>

lemma quot-cases [cases type: quot]:
  obtains  $a$  where  $A = [a]$ 
<proof>

```

## 89.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

**theorem** *quot-equality* [*iff?*]:  $\lfloor a \rfloor = \lfloor b \rfloor \iff a \sim b$   
*<proof>*

## 89.3 Picking representing elements

**definition** *pick* :: 'a::equiv quot  $\Rightarrow$  'a  
**where** *pick* A = (SOME a. A =  $\lfloor a \rfloor$ )

**theorem** *pick-equiv* [*intro*]: *pick*  $\lfloor a \rfloor \sim a$   
*<proof>*

**theorem** *pick-inverse* [*intro*]:  $\lfloor \text{pick } A \rfloor = A$   
*<proof>*

The following rules support canonical function definitions on quotient types (with up to two arguments). Note that the stripped-down version without additional conditions is sufficient most of the time.

**theorem** *quot-cond-function*:

**assumes** *eq*:  $\bigwedge X Y. P X Y \implies f X Y \equiv g (\text{pick } X) (\text{pick } Y)$   
**and** *cong*:  $\bigwedge x x' y y'. \lfloor x \rfloor = \lfloor x' \rfloor \implies \lfloor y \rfloor = \lfloor y' \rfloor$   
 $\implies P \lfloor x \rfloor \lfloor y \rfloor \implies P \lfloor x' \rfloor \lfloor y' \rfloor \implies g x y = g x' y'$   
**and** *P*:  $P \lfloor a \rfloor \lfloor b \rfloor$   
**shows**  $f \lfloor a \rfloor \lfloor b \rfloor = g a b$   
*<proof>*

**theorem** *quot-function*:

**assumes**  $\bigwedge X Y. f X Y \equiv g (\text{pick } X) (\text{pick } Y)$   
**and**  $\bigwedge x x' y y'. \lfloor x \rfloor = \lfloor x' \rfloor \implies \lfloor y \rfloor = \lfloor y' \rfloor \implies g x y = g x' y'$   
**shows**  $f \lfloor a \rfloor \lfloor b \rfloor = g a b$   
*<proof>*

**theorem** *quot-function'*:

$(\bigwedge X Y. f X Y \equiv g (\text{pick } X) (\text{pick } Y)) \implies$   
 $(\bigwedge x x' y y'. x \sim x' \implies y \sim y' \implies g x y = g x' y') \implies$   
 $f \lfloor a \rfloor \lfloor b \rfloor = g a b$   
*<proof>*

end

## 90 Ramsey’s Theorem

**theory** *Ramsey*

**imports** *Infinite-Set Equipollence FuncSet*  
**begin**

## 90.1 Preliminary definitions

**abbreviation** *strict-sorted* :: 'a::linorder list  $\Rightarrow$  bool **where**  
*strict-sorted*  $\equiv$  *sorted-wrt* (<)

### 90.1.1 The $n$ -element subsets of a set $A$

**definition** *nsets* :: ['a set, nat]  $\Rightarrow$  'a set set ( $\langle\langle$ notation= $\langle$ mixfix *nsets* $\rangle\rangle[-]^{-}\rangle$   
 [0,999] 999)

**where** *nsets*  $A$   $n \equiv$   $\{N. N \subseteq A \wedge \text{finite } N \wedge \text{card } N = n\}$

**lemma** *finite-imp-finite-nsets*: *finite*  $A \Longrightarrow$  *finite* ( $[A]^k$ )  
*<proof>*

**lemma** *nsets-mono*:  $A \subseteq B \Longrightarrow$  *nsets*  $A$   $n \subseteq$  *nsets*  $B$   $n$   
*<proof>*

**lemma** *nsets-Pi-contr*:  $A' \subseteq A \Longrightarrow$  *Pi* ( $[A]^n$ )  $B \subseteq$  *Pi* ( $[A']^n$ )  $B$   
*<proof>*

**lemma** *nsets-2-eq*:  $[A]^2 = (\bigcup x \in A. \bigcup y \in A - \{x\}. \{\{x, y\}\})$   
*<proof>*

**lemma** *nsets2-E*:  
**assumes**  $e \in [A]^2$   
**obtains**  $x$   $y$  **where**  $e = \{x, y\}$   $x \in A$   $y \in A$   $x \neq y$   
*<proof>*

**lemma** *nsets-doubleton-2-eq* [*simp*]:  $\{\{x, y\}\}^2 = (\text{if } x=y \text{ then } \{\} \text{ else } \{\{x, y\}\})$   
*<proof>*

**lemma** *doubleton-in-nsets-2* [*simp*]:  $\{x, y\} \in [A]^2 \longleftrightarrow x \in A \wedge y \in A \wedge x \neq y$   
*<proof>*

**lemma** *nsets-3-eq*:  $[A]^3 = (\bigcup x \in A. \bigcup y \in A - \{x\}. \bigcup z \in A - \{x, y\}. \{\{x, y, z\}\})$   
*<proof>*

**lemma** *nsets-4-eq*:  $[A]^4 = (\bigcup u \in A. \bigcup x \in A - \{u\}. \bigcup y \in A - \{u, x\}. \bigcup z \in A - \{u, x, y\}. \{\{u, x, y, z\}\})$   
 (**is** - = ?*rhs*)  
*<proof>*

**lemma** *nsets-disjoint-2*:  
 $X \cap Y = \{\} \Longrightarrow [X \cup Y]^2 = [X]^2 \cup [Y]^2 \cup (\bigcup x \in X. \bigcup y \in Y. \{\{x, y\}\})$   
*<proof>*

**lemma** *ordered-nsets-2-eq*:  
**fixes**  $A$  :: 'a::linorder set  
**shows**  $[A]^2 = \{\{x, y\} \mid x$   $y. x \in A \wedge y \in A \wedge x < y\}$   
 (**is** - = ?*rhs*)

⟨proof⟩

**lemma** *ordered-nsets-3-eq*:

**fixes**  $A :: 'a::\text{linorder set}$

**shows**  $[A]^3 = \{\{x,y,z\} \mid x\ y\ z.\ x \in A \wedge y \in A \wedge z \in A \wedge x < y \wedge y < z\}$

(**is** - = ?*rhs*)

⟨proof⟩

**lemma** *ordered-nsets-4-eq*:

**fixes**  $A :: 'a::\text{linorder set}$

**defines**  $\text{rhs} \equiv \lambda U. \exists u\ x\ y\ z. U = \{u,x,y,z\} \wedge u \in A \wedge x \in A \wedge y \in A \wedge z \in A$   
 $\wedge u < x \wedge x < y \wedge y < z$

**shows**  $[A]^4 = \text{Collect rhs}$

⟨proof⟩

**lemma** *ordered-nsets-5-eq*:

**fixes**  $A :: 'a::\text{linorder set}$

**defines**  $\text{rhs} \equiv \lambda U. \exists u\ v\ x\ y\ z. U = \{u,v,x,y,z\} \wedge u \in A \wedge v \in A \wedge x \in A \wedge y$   
 $\in A \wedge z \in A \wedge u < v \wedge v < x \wedge x < y \wedge y < z$

**shows**  $[A]^5 = \text{Collect rhs}$

⟨proof⟩

**lemma** *binomial-eq-nsets*:  $n\ \text{choose}\ k = \text{card}\ (\text{nsets}\ \{0..<n\}\ k)$

⟨proof⟩

**lemma** *nsets-eq-empty-iff*:  $\text{nsets}\ A\ r = \{\} \longleftrightarrow \text{finite}\ A \wedge \text{card}\ A < r$

⟨proof⟩

**lemma** *nsets-eq-empty*:  $\llbracket \text{finite}\ A; \text{card}\ A < r \rrbracket \Longrightarrow \text{nsets}\ A\ r = \{\}$

⟨proof⟩

**lemma** *nsets-empty-iff*:  $\text{nsets}\ \{\}\ r = (\text{if}\ r=0\ \text{then}\ \{\{\}\}\ \text{else}\ \{\})$

⟨proof⟩

**lemma** *nsets-singleton-iff*:  $\text{nsets}\ \{a\}\ r = (\text{if}\ r=0\ \text{then}\ \{\{\}\}\ \text{else}\ \text{if}\ r=1\ \text{then}\ \{\{a\}\}$   
 $\text{else}\ \{\})$

⟨proof⟩

**lemma** *nsets-self* [*simp*]:  $\text{nsets}\ \{..<m\}\ m = \{\{..<m\}\}$

⟨proof⟩

**lemma** *nsets-zero* [*simp*]:  $\text{nsets}\ A\ 0 = \{\{\}\}$

⟨proof⟩

**lemma** *nsets-one*:  $\text{nsets}\ A\ (\text{Suc}\ 0) = (\lambda x. \{x\})\ ` A$

⟨proof⟩

**lemma** *inj-on-nsets*:

**assumes** *inj-on*  $f\ A$

**shows** *inj-on*  $(\lambda X. f \text{ ‘ } X) ([A]^n)$   
 ⟨*proof*⟩

**lemma** *bij-betw-nsets*:

**assumes** *bij-betw*  $f A B$

**shows** *bij-betw*  $(\lambda X. f \text{ ‘ } X) ([A]^n) ([B]^n)$   
 ⟨*proof*⟩

**lemma** *nset-image-obtains*:

**assumes**  $X \in [f \text{ ‘ } A]^k$  *inj-on*  $f A$

**obtains**  $Y$  **where**  $Y \in [A]^k$   $X = f \text{ ‘ } Y$   
 ⟨*proof*⟩

**lemma** *nsets-image-funcset*:

**assumes**  $g \in S \rightarrow T$  **and** *inj-on*  $g S$

**shows**  $(\lambda X. g \text{ ‘ } X) \in [S]^k \rightarrow [T]^k$   
 ⟨*proof*⟩

**lemma** *nsets-compose-image-funcset*:

**assumes**  $f: f \in [T]^k \rightarrow D$  **and**  $g \in S \rightarrow T$  **and** *inj-on*  $g S$

**shows**  $f \circ (\lambda X. g \text{ ‘ } X) \in [S]^k \rightarrow D$   
 ⟨*proof*⟩

### 90.1.2 Further properties, involving equipollence

**lemma** *nsets-lepoll-cong*:

**assumes**  $A \lesssim B$

**shows**  $[A]^k \lesssim [B]^k$   
 ⟨*proof*⟩

**lemma** *nsets-epoll-cong*:

**assumes**  $A \approx B$

**shows**  $[A]^k \approx [B]^k$

⟨*proof*⟩

**lemma** *infinite-imp-infinite-nsets*:

**assumes** *inf*: *infinite*  $A$  **and**  $k > 0$

**shows** *infinite*  $([A]^k)$   
 ⟨*proof*⟩

**lemma** *finite-nsets-iff*:

**assumes**  $k > 0$

**shows** *finite*  $([A]^k) \longleftrightarrow$  *finite*  $A$

⟨*proof*⟩

**lemma** *card-nsets [simp]*:  $\text{card } (nsets A k) = \text{card } A$  *choose*  $k$

⟨*proof*⟩

### 90.1.3 Partition predicates

**definition** *monochromatic*  $\equiv \lambda\beta \alpha \gamma f i. \exists H \in \text{nsets } \beta \alpha. f \text{ ‘ } (\text{nsets } H \gamma) \subseteq \{i\}$   
uniform partition sizes

**definition** *partn*  $:: 'a \text{ set} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'b \text{ set} \Rightarrow \text{bool}$   
**where** *partn*  $\beta \alpha \gamma \delta \equiv \forall f \in \text{nsets } \beta \gamma \rightarrow \delta. \exists \xi \in \delta. \text{monochromatic } \beta \alpha \gamma f \xi$   
partition sizes enumerated in a list

**definition** *partn-1st*  $:: 'a \text{ set} \Rightarrow \text{nat list} \Rightarrow \text{nat} \Rightarrow \text{bool}$   
**where** *partn-1st*  $\beta \alpha \gamma \equiv \forall f \in \text{nsets } \beta \gamma \rightarrow \{..<\text{length } \alpha\}. \exists i < \text{length } \alpha. \text{monochromatic } \beta (\alpha!i) \gamma f i$

There’s always a 0-clique

**lemma** *partn-1st-0*:  $\gamma > 0 \implies \text{partn-1st } \beta (0\#\alpha) \gamma$   
(*proof*)

**lemma** *partn-1st-0'*:  $\gamma > 0 \implies \text{partn-1st } \beta (a\#0\#\alpha) \gamma$   
(*proof*)

**lemma** *partn-1st-greater-resource*:  
**fixes**  $M::\text{nat}$   
**assumes**  $M: \text{partn-1st } \{..<M\} \alpha \gamma$  **and**  $M \leq N$   
**shows**  $\text{partn-1st } \{..<N\} \alpha \gamma$   
(*proof*)

**lemma** *partn-1st-fewer-colours*:  
**assumes** *major*:  $\text{partn-1st } \beta (n\#\alpha) \gamma$  **and**  $n \geq \gamma$   
**shows**  $\text{partn-1st } \beta \alpha \gamma$   
(*proof*)

**lemma** *partn-1st-eq-partn*:  $\text{partn-1st } \{..<n\} [m,m] 2 = \text{partn } \{..<n\} m 2 \{..<2::\text{nat}\}$   
(*proof*)

**lemma** *partn-1stE*:  
**assumes**  $\text{partn-1st } \beta \alpha \gamma f \in \text{nsets } \beta \gamma \rightarrow \{..<l\} \text{length } \alpha = l$   
**obtains**  $i H$  **where**  $i < \text{length } \alpha$   $H \in \text{nsets } \beta (\alpha!i) f \text{ ‘ } (\text{nsets } H \gamma) \subseteq \{i\}$   
(*proof*)

**lemma** *partn-1st-less*:  
**assumes**  $M: \text{partn-1st } \beta \alpha n$  **and** *eq*:  $\text{length } \alpha' = \text{length } \alpha$   
**and** *le*:  $\bigwedge i. i < \text{length } \alpha \implies \alpha!i \leq \alpha'i$   
**shows**  $\text{partn-1st } \beta \alpha' n$   
(*proof*)

## 90.2 Finite versions of Ramsey’s theorem

To distinguish the finite and infinite ones, lower and upper case names are used (ramsey vs Ramsey).

### 90.2.1 The Erds–Szekeres theorem exhibits an upper bound for Ramsey numbers

The Erds–Szekeres bound, essentially extracted from the proof

```

fun ES :: [nat,nat,nat] => nat
  where ES 0 k l = max k l
    |   ES (Suc r) k l =
      (if r=0 then k+l-1
       else if k=0 ∨ l=0 then 1 else Suc (ES r (ES (Suc r) (k-1) l) (ES (Suc
r) k (l-1))))

```

```

declare ES.simps [simp del]

```

```

lemma ES-0 [simp]: ES 0 k l = max k l
  <proof>

```

```

lemma ES-1 [simp]: ES 1 k l = k+l-1
  <proof>

```

```

lemma ES-2: ES 2 k l = (if k=0 ∨ l=0 then 1 else ES 2 (k-1) l + ES 2 k (l-1))
  <proof>

```

The Erds–Szekeres upper bound

```

lemma ES2-choose: ES 2 k l = (k+l) choose k
  <proof>

```

### 90.2.2 Trivial cases

Vacuous, since we are dealing with 0-sets!

```

lemma ramsey0: ∃ N::nat. partn-lst {..N} [q1,q2] 0
  <proof>

```

Just the pigeon hole principle, since we are dealing with 1-sets

```

lemma ramsey1-explicit: partn-lst {..q0 + q1 - Suc 0} [q0,q1] 1
  <proof>

```

```

lemma ramsey1: ∃ N::nat. partn-lst {..N} [q0,q1] 1
  <proof>

```

### 90.2.3 Ramsey’s theorem with TWO colours and arbitrary exponents (hypergraph version)

```

lemma ramsey-induction-step:

```

```

  fixes p::nat

```

```

  assumes p1: partn-lst {..p1} [q1-1,q2] (Suc r) and p2: partn-lst {..p2}
[q1,q2-1] (Suc r)

```

```

  and p: partn-lst {..p} [p1,p2] r

```

```

  and q1>0 q2>0

```

```

  shows partn-lst {..Suc p} [q1, q2] (Suc r)

```

*<proof>*

**proposition** *ramsey2-full*: *partn-1st*  $\{..<ES\ r\ q1\ q2\} [q1,q2]\ r$   
*<proof>*

### 90.2.4 Full Ramsey’s theorem with multiple colours and arbitrary exponents

**theorem** *ramsey-full*:  $\exists N::nat.$  *partn-1st*  $\{..<N\} qs\ r$   
*<proof>*

### 90.2.5 Simple graph version

This is the most basic version in terms of cliques and independent sets, i.e. the version for graphs and 2 colours.

**definition** *clique*  $V\ E \leftrightarrow (\forall v \in V. \forall w \in V. v \neq w \rightarrow \{v, w\} \in E)$

**definition** *indep*  $V\ E \leftrightarrow (\forall v \in V. \forall w \in V. v \neq w \rightarrow \{v, w\} \notin E)$

**lemma** *clique-Un*:  $\llbracket \text{clique } K\ F; \text{clique } L\ F; \forall v \in K. \forall w \in L. v \neq w \rightarrow \{v, w\} \in F \rrbracket$   
 $\implies \text{clique } (K \cup L)\ F$   
*<proof>*

**lemma** *null-clique[simp]*: *clique*  $\{\}\ E$  **and** *null-indep[simp]*: *indep*  $\{\}\ E$   
*<proof>*

**lemma** *smaller-clique*:  $\llbracket \text{clique } R\ E; R' \subseteq R \rrbracket \implies \text{clique } R'\ E$   
*<proof>*

**lemma** *smaller-indep*:  $\llbracket \text{indep } R\ E; R' \subseteq R \rrbracket \implies \text{indep } R'\ E$   
*<proof>*

**lemma** *ramsey2*:

$\exists r \geq 1. \forall (V::'a\ \text{set}) (E::'a\ \text{set set}). \text{finite } V \wedge \text{card } V \geq r \rightarrow$

$(\exists R \subseteq V. \text{card } R = m \wedge \text{clique } R\ E \vee \text{card } R = n \wedge \text{indep } R\ E)$

*<proof>*

## 90.3 Preliminaries for the infinitary version

### 90.3.1 “Axiom” of Dependent Choice

**primrec** *choice* ::  $('a \Rightarrow \text{bool}) \Rightarrow ('a \times 'a)\ \text{set} \Rightarrow \text{nat} \Rightarrow 'a$

**where** — An integer-indexed chain of choices

*choice-0*: *choice*  $P\ r\ 0 = (\text{SOME } x. P\ x)$

| *choice-Suc*: *choice*  $P\ r\ (\text{Suc } n) = (\text{SOME } y. P\ y \wedge (\text{choice } P\ r\ n, y) \in r)$

**lemma** *choice-n*:

**assumes**  $P0$ :  $P\ x0$

**and** *Pstep*:  $\bigwedge x. P\ x \implies \exists y. P\ y \wedge (x, y) \in r$

**shows**  $P\ (\text{choice } P\ r\ n)$



⟨proof⟩

**lemma** *dependent-choice*:

**assumes** *trans*:  $\text{trans } r$

**and**  $P0$ :  $P x0$

**and** *Pstep*:  $\bigwedge x. P x \implies \exists y. P y \wedge (x, y) \in r$

**obtains**  $f :: \text{nat} \Rightarrow 'a$  **where**  $\bigwedge n. P (f n)$  **and**  $\bigwedge n m. n < m \implies (f n, f m) \in r$

⟨proof⟩

### 90.3.2 Partition functions

**definition** *part-fn* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ set} \Rightarrow ('a \text{ set} \Rightarrow \text{nat}) \Rightarrow \text{bool}$

— the function  $f$  partitions the  $r$ -subsets of the typically infinite set  $Y$  into  $s$  distinct categories.

**where**  $\text{part-fn } r s Y f \longleftrightarrow (f \in \text{nsets } Y r \rightarrow \{..<s\})$

For induction, we decrease the value of  $r$  in partitions.

**lemma** *part-fn-Suc-imp-part-fn*:

$\llbracket \text{infinite } Y; \text{part-fn } (\text{Suc } r) s Y f; y \in Y \rrbracket \implies \text{part-fn } r s (Y - \{y\}) (\lambda u. f (\text{insert } y u))$

⟨proof⟩

**lemma** *part-fn-subset*:  $\text{part-fn } r s YY f \implies Y \subseteq YY \implies \text{part-fn } r s Y f$

⟨proof⟩

## 90.4 Ramsey’s Theorem: Infinitary Version

**lemma** *Ramsey-induction*:

**fixes**  $s r :: \text{nat}$

**and**  $YY :: 'a \text{ set}$

**and**  $f :: 'a \text{ set} \Rightarrow \text{nat}$

**assumes**  $\text{infinite } YY$   $\text{part-fn } r s YY f$

**shows**  $\exists Y' t'. Y' \subseteq YY \wedge \text{infinite } Y' \wedge t' < s \wedge (\forall X. X \subseteq Y' \wedge \text{finite } X \wedge \text{card } X = r \longrightarrow f X = t')$

⟨proof⟩

**theorem** *Ramsey*:

**fixes**  $s r :: \text{nat}$

**and**  $Z :: 'a \text{ set}$

**and**  $f :: 'a \text{ set} \Rightarrow \text{nat}$

**shows**

$\llbracket \text{infinite } Z; \forall X. X \subseteq Z \wedge \text{finite } X \wedge \text{card } X = r \longrightarrow f X < s \rrbracket$

$\implies \exists Y t. Y \subseteq Z \wedge \text{infinite } Y \wedge t < s$

$\wedge (\forall X. X \subseteq Y \wedge \text{finite } X \wedge \text{card } X = r \longrightarrow f X = t)$

⟨proof⟩

**corollary** *Ramsey2*:

**fixes**  $s :: \text{nat}$

**and**  $Z :: 'a \text{ set}$   
**and**  $f :: 'a \text{ set} \Rightarrow \text{nat}$   
**assumes**  $\text{inf}Z: \text{infinite } Z$   
**and**  $\text{part}: \forall x \in Z. \forall y \in Z. x \neq y \longrightarrow f \{x, y\} < s$   
**shows**  $\exists Y t. Y \subseteq Z \wedge \text{infinite } Y \wedge t < s \wedge (\forall x \in Y. \forall y \in Y. x \neq y \longrightarrow f \{x, y\} = t)$   
 $\langle \text{proof} \rangle$

**corollary** *Ramsey-nsets:*

**fixes**  $f :: 'a \text{ set} \Rightarrow \text{nat}$   
**assumes**  $\text{infinite } Z f \text{ ' nsets } Z r \subseteq \{..<s\}$   
**obtains**  $Y t \text{ where } Y \subseteq Z \text{ infinite } Y t < s f \text{ ' nsets } Y r \subseteq \{t\}$   
 $\langle \text{proof} \rangle$

## 90.5 Disjunctive Well-Foundedness

An application of Ramsey’s theorem to program termination. See [4].

**definition**  $\text{disj-wf} :: ('a \times 'a) \text{ set} \Rightarrow \text{bool}$

**where**  $\text{disj-wf } r \longleftrightarrow (\exists T. \exists n::\text{nat}. (\forall i < n. \text{wf } (T i)) \wedge r = (\bigcup_{i < n. T i}))$

**definition**  $\text{transition-idx} :: (\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow ('a \times 'a) \text{ set}) \Rightarrow \text{nat set} \Rightarrow \text{nat}$

**where**  $\text{transition-idx } s T A = (\text{LEAST } k. \exists i j. A = \{i, j\} \wedge i < j \wedge (s j, s i) \in T k)$

**lemma** *transition-idx-less:*

**assumes**  $i < j (s j, s i) \in T k k < n$

**shows**  $\text{transition-idx } s T \{i, j\} < n$

$\langle \text{proof} \rangle$

**lemma** *transition-idx-in:*

**assumes**  $i < j (s j, s i) \in T k$

**shows**  $(s j, s i) \in T (\text{transition-idx } s T \{i, j\})$

$\langle \text{proof} \rangle$

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

**lemma**  $\text{disj-wf}: \text{disj-wf } r \longleftrightarrow (\exists T. \exists n::\text{nat}. (\forall i < n. \text{wf}(T i)) \wedge r \subseteq (\bigcup_{i < n. T i}))$

$\langle \text{proof} \rangle$

**theorem** *trans-disj-wf-implies-wf:*

**assumes**  $\text{trans } r$

**and**  $\text{disj-wf } r$

**shows**  $\text{wf } r$

$\langle \text{proof} \rangle$

**end**

## 91 Modulo and congruence on the reals

```
theory Real-Mod
  imports Complex-Main
begin
```

**definition** *rmod* :: *real*  $\Rightarrow$  *real*  $\Rightarrow$  *real* (**infixl**  $\langle$ *rmod* $\rangle$  70) **where**  
 $x \text{ rmod } y = x - |y| * \text{of-int } \lfloor x / |y| \rfloor$

**lemma** *rmod-conv-frac*:  $y \neq 0 \implies x \text{ rmod } y = \text{frac } (x / |y|) * |y|$   
 $\langle$ *proof* $\rangle$

**lemma** *rmod-conv-frac'*:  $x \text{ rmod } y = (\text{if } y = 0 \text{ then } x \text{ else } \text{frac } (x / |y|) * |y|)$   
 $\langle$ *proof* $\rangle$

**lemma** *rmod-rmod* [*simp*]:  $(x \text{ rmod } y) \text{ rmod } y = x \text{ rmod } y$   
 $\langle$ *proof* $\rangle$

**lemma** *rmod-0-right* [*simp*]:  $x \text{ rmod } 0 = x$   
 $\langle$ *proof* $\rangle$

**lemma** *rmod-less*:  $m > 0 \implies x \text{ rmod } m < m$   
 $\langle$ *proof* $\rangle$

**lemma** *rmod-less-abs*:  $m \neq 0 \implies x \text{ rmod } m < |m|$   
 $\langle$ *proof* $\rangle$

**lemma** *rmod-le*:  $m > 0 \implies x \text{ rmod } m \leq m$   
 $\langle$ *proof* $\rangle$

**lemma** *rmod-nonneg*:  $m \neq 0 \implies x \text{ rmod } m \geq 0$   
 $\langle$ *proof* $\rangle$

**lemma** *rmod-unique*:  
**assumes**  $z \in \{0..<|y|\}$   $x = z + \text{of-int } n * y$   
**shows**  $x \text{ rmod } y = z$   
 $\langle$ *proof* $\rangle$

**lemma** *rmod-0* [*simp*]:  $0 \text{ rmod } z = 0$   
 $\langle$ *proof* $\rangle$

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

**lemma** *rmod-diff*:  $(x \text{ rmod } z - y \text{ rmod } z) \text{ rmod } z = (x - y) \text{ rmod } z$   
 $\langle$ *proof* $\rangle$

**lemma** *rmod-self* [*simp*]:  $x \text{ rmod } x = 0$   
 ⟨*proof*⟩

**lemma** *rmod-self-multiple-int* [*simp*]:  $(\text{of-int } n * x) \text{ rmod } x = 0$   
 ⟨*proof*⟩

**lemma** *rmod-self-multiple-nat* [*simp*]:  $(\text{of-nat } n * x) \text{ rmod } x = 0$   
 ⟨*proof*⟩

**lemma** *rmod-self-multiple-numeral* [*simp*]:  $(\text{numeral } n * x) \text{ rmod } x = 0$   
 ⟨*proof*⟩

**lemma** *rmod-self-multiple-int'* [*simp*]:  $(x * \text{of-int } n) \text{ rmod } x = 0$   
 ⟨*proof*⟩

**lemma** *rmod-self-multiple-nat'* [*simp*]:  $(x * \text{of-nat } n) \text{ rmod } x = 0$   
 ⟨*proof*⟩

**lemma** *rmod-self-multiple-numeral'* [*simp*]:  $(x * \text{numeral } n) \text{ rmod } x = 0$   
 ⟨*proof*⟩

**lemma** *rmod-idem* [*simp*]:  $x \in \{0..<|y|\} \implies x \text{ rmod } y = x$   
 ⟨*proof*⟩

**definition** *rcong* ::  $\text{real} \Rightarrow \text{real} \Rightarrow \text{real} \Rightarrow \text{bool}$   
 (⟨⟨*indent*=1 *notation*=⟨*mixfix* *rcong*⟩[- = -] '(*rmod* -)⟩⟩)  
**where**  $[x = y] (\text{rmod } m) \longleftrightarrow x \text{ rmod } m = y \text{ rmod } m$

**named-theorems** *rcong-intros*

**lemma** *rcong-0-right* [*simp*]:  $[x = y] (\text{rmod } 0) \longleftrightarrow x = y$   
 ⟨*proof*⟩

**lemma** *rcong-0-iff*:  $[x = 0] (\text{rmod } m) \longleftrightarrow x \text{ rmod } m = 0$   
**and** *rcong-0-iff'*:  $[0 = x] (\text{rmod } m) \longleftrightarrow x \text{ rmod } m = 0$   
 ⟨*proof*⟩

**lemma** *rcong-refl* [*simp*, *intro!*, *rcong-intros*]:  $[x = x] (\text{rmod } m)$   
 ⟨*proof*⟩

**lemma** *rcong-sym*:  $[y = x] (\text{rmod } m) \implies [x = y] (\text{rmod } m)$   
 ⟨*proof*⟩

**lemma** *rcong-sym-iff*:  $[y = x] (\text{rmod } m) \longleftrightarrow [x = y] (\text{rmod } m)$   
 ⟨*proof*⟩

**lemma** *rcong-trans* [*trans*]:  $[x = y] (\text{rmod } m) \implies [y = z] (\text{rmod } m) \implies [x = z]$

(*rmod m*)  
 ⟨*proof*⟩

**lemma** *rcong-add* [*rcong-intros*]:

$[a = b] (rmod\ m) \implies [c = d] (rmod\ m) \implies [a + c = b + d] (rmod\ m)$   
 ⟨*proof*⟩

**lemma** *rcong-diff* [*rcong-intros*]:

$[a = b] (rmod\ m) \implies [c = d] (rmod\ m) \implies [a - c = b - d] (rmod\ m)$   
 ⟨*proof*⟩

**lemma** *rcong-uminus* [*rcong-intros*]:

$[a = b] (rmod\ m) \implies [-a = -b] (rmod\ m)$   
 ⟨*proof*⟩

**lemma** *rcong-uminus-uminus-iff* [*simp*]:  $[-x = -y] (rmod\ m) \longleftrightarrow [x = y] (rmod\ m)$

⟨*proof*⟩

**lemma** *rcong-uminus-left-iff*:  $[-x = y] (rmod\ m) \longleftrightarrow [x = -y] (rmod\ m)$

⟨*proof*⟩

**lemma** *rcong-add-right-cancel* [*simp*]:  $[a + c = b + c] (rmod\ m) \longleftrightarrow [a = b] (rmod\ m)$

⟨*proof*⟩

**lemma** *rcong-add-left-cancel* [*simp*]:  $[c + a = c + b] (rmod\ m) \longleftrightarrow [a = b] (rmod\ m)$

⟨*proof*⟩

**lemma** *rcong-diff-right-cancel* [*simp*]:  $[a - c = b - c] (rmod\ m) \longleftrightarrow [a = b] (rmod\ m)$

⟨*proof*⟩

**lemma** *rcong-diff-left-cancel* [*simp*]:  $[c - a = c - b] (rmod\ m) \longleftrightarrow [a = b] (rmod\ m)$

⟨*proof*⟩

**lemma** *rcong-rmod-right-iff* [*simp*]:  $[a = (b\ rmod\ m)] (rmod\ m) \longleftrightarrow [a = b] (rmod\ m)$

**and** *rcong-rmod-left-iff* [*simp*]:  $[(a\ rmod\ m) = b] (rmod\ m) \longleftrightarrow [a = b] (rmod\ m)$

⟨*proof*⟩

**lemma** *rcong-rmod-left* [*rcong-intros*]:  $[a = b] (rmod\ m) \implies [(a\ rmod\ m) = b]$

**and** *rcong-rmod-right* [*rcong-intros*]:  $[a = b] (rmod\ m) \implies [a = (b\ rmod\ m)]$

⟨*proof*⟩

**lemma** *rcong-mult-of-int-0-left-left* [*rcong-intros*]:  $[0 = \text{of-int } n * m] \text{ (rmod } m)$   
**and** *rcong-mult-of-int-0-right-left* [*rcong-intros*]:  $[0 = m * \text{of-int } n] \text{ (rmod } m)$   
**and** *rcong-mult-of-int-0-left-right* [*rcong-intros*]:  $[\text{of-int } n * m = 0] \text{ (rmod } m)$   
**and** *rcong-mult-of-int-0-right-right* [*rcong-intros*]:  $[m * \text{of-int } n = 0] \text{ (rmod } m)$   
 ⟨*proof*⟩

**lemma** *rcong-altdef*:  $[a = b] \text{ (rmod } m) \longleftrightarrow (\exists n. b = a + \text{of-int } n * m)$   
 ⟨*proof*⟩

**lemma** *rcong-conv-diff-rmod-eq-0*:  $[x = y] \text{ (rmod } m) \longleftrightarrow (x - y) \text{ rmod } m = 0$   
 ⟨*proof*⟩

**lemma** *rcong-imp-eq*:  
**assumes**  $[x = y] \text{ (rmod } m) \mid x - y < |m|$   
**shows**  $x = y$   
 ⟨*proof*⟩

**lemma** *rcong-mult-modulus*:  
**assumes**  $[a = b] \text{ (rmod } (m / c)) \mid c \neq 0$   
**shows**  $[a * c = b * c] \text{ (rmod } m)$   
 ⟨*proof*⟩

**lemma** *rcong-divide-modulus*:  
**assumes**  $[a = b] \text{ (rmod } (m * c)) \mid c \neq 0$   
**shows**  $[a / c = b / c] \text{ (rmod } m)$   
 ⟨*proof*⟩

end

## 92 Generic reflection and reification

**theory** *Reflection*  
**imports** *Main*  
**begin**

⟨*ML*⟩

end

**theory** *Rewrite*  
**imports** *Main*  
**begin**

**consts** *rewrite-HOLE* :: 'a::{} (↔)

**lemma** *eta-expand*:  
**fixes**  $f :: 'a::{} \Rightarrow 'b::{} \{ \}$

**shows**  $f \equiv \lambda x. f x$   $\langle proof \rangle$

**lemma** *imp-cong-eq*:

$(PROP A \implies (PROP B \implies PROP C)) \equiv (PROP B' \implies PROP C') \equiv$   
 $((PROP B \implies PROP A \implies PROP C) \equiv (PROP B' \implies PROP A \implies PROP$   
 $C'))$   
 $\langle proof \rangle$

$\langle ML \rangle$

**end**

### 93 Assigning lengths to types by type classes

**theory** *Type-Length*  
**imports** *Natural\_Type*  
**begin**

The aim of this is to allow any type as index type, but to provide a default instantiation for numeral types. This independence requires some duplication with the definitions in `Natural_Type.thy`.

**class** *len0* =  
**fixes** *len-of* :: 'a itself  $\Rightarrow$  nat

**syntax** *-type-length* :: type  $\Rightarrow$  nat  $\langle \langle (1LENGTH/(1'(-))) \rangle \rangle$   
**syntax-consts** *-type-length*  $\equiv$  *len-of*  
**translations**  $LENGTH('a) \rightarrow CONST \textit{len-of} \textit{TYPE}('a)$   
 $\langle ML \rangle$

Some theorems are only true on words with length greater 0.

**class** *len* = *len0* +  
**assumes** *len-gt-0* [iff]:  $0 < LENGTH('a)$   
**begin**

**lemma** *len-not-eq-0* [simp]:  
 $LENGTH('a) \neq 0$   
 $\langle proof \rangle$

**end**

**instantiation** *num0* and *num1* :: *len0*  
**begin**

**definition** *len-num0*:  $\textit{len-of} (- :: \textit{num0} \textit{itself}) = 0$

**definition** *len-num1*:  $\textit{len-of} (- :: \textit{num1} \textit{itself}) = 1$

**instance**  $\langle proof \rangle$

**end**

```

instantiation bit0 and bit1 :: (len0) len0
begin

definition len-bit0: len-of (- :: 'a::len0 bit0 itself) = 2 * LENGTH('a)
definition len-bit1: len-of (- :: 'a::len0 bit1 itself) = 2 * LENGTH('a) + 1

instance ⟨proof⟩

end

lemmas len-of-numeral-defs [simp] = len-num0 len-num1 len-bit0 len-bit1

instance num1 :: len
  ⟨proof⟩
instance bit0 :: (len) len
  ⟨proof⟩
instance bit1 :: (len0) len
  ⟨proof⟩

instantiation Enum.finite-1 :: len
begin

definition
  len-of-finite-1 (x :: Enum.finite-1 itself) ≡ (1 :: nat)

instance
  ⟨proof⟩

end

instantiation Enum.finite-2 :: len
begin

definition
  len-of-finite-2 (x :: Enum.finite-2 itself) ≡ (2 :: nat)

instance
  ⟨proof⟩

end

instantiation Enum.finite-3 :: len
begin

definition
  len-of-finite-3 (x :: Enum.finite-3 itself) ≡ (4 :: nat)

instance

```



⟨*proof*⟩

**end**

**lemma** *length-not-greater-eq-2-iff* [*simp*]:  
 ⟨ $\neg 2 \leq \text{LENGTH}('a::\text{len}) \longleftrightarrow \text{LENGTH}('a) = 1$ ⟩  
 ⟨*proof*⟩

**context** *linordered-idom*  
**begin**

**lemma** *two-less-eq-exp-length* [*simp*]:  
 ⟨ $2 \leq 2 \wedge \text{LENGTH}('b::\text{len})$ ⟩  
 ⟨*proof*⟩

**end**

**lemma** *less-eq-decr-length-iff* [*simp*]:  
 ⟨ $n \leq \text{LENGTH}('a::\text{len}) - \text{Suc } 0 \longleftrightarrow n < \text{LENGTH}('a)$ ⟩  
 ⟨*proof*⟩

**lemma** *decr-length-less-iff* [*simp*]:  
 ⟨ $\text{LENGTH}('a::\text{len}) - \text{Suc } 0 < n \longleftrightarrow \text{LENGTH}('a) \leq n$ ⟩  
 ⟨*proof*⟩

**end**

## 94 Saturated arithmetic

**theory** *Saturated*  
**imports** *Natural-Type Type-Length*  
**begin**

### 94.1 The type of saturated naturals

**typedef** (**overloaded**) (*'a::len*) *sat* = {..*LENGTH*('a)}  
**morphisms** *nat-of Abs-sat*  
 ⟨*proof*⟩

**lemma** *sat-eqI*:  
 $\text{nat-of } m = \text{nat-of } n \implies m = n$   
 ⟨*proof*⟩

**lemma** *sat-eq-iff*:  
 $m = n \longleftrightarrow \text{nat-of } m = \text{nat-of } n$   
 ⟨*proof*⟩

**lemma** *Abs-sat-nat-of* [*code abstype*]:  
 $\text{Abs-sat } (\text{nat-of } n) = n$

*<proof>*

**definition** *Abs-sat'* :: *nat*  $\Rightarrow$  *'a::len sat* **where**  
*Abs-sat'* *n* = *Abs-sat* (*min* (*LENGTH*('a)) *n*)

**lemma** *nat-of-Abs-sat'* [*simp*]:  
*nat-of* (*Abs-sat'* *n* :: ('a::len) *sat*) = *min* (*LENGTH*('a)) *n*  
*<proof>*

**lemma** *nat-of-le-len-of* [*simp*]:  
*nat-of* (*n* :: ('a::len) *sat*)  $\leq$  *LENGTH*('a)  
*<proof>*

**lemma** *min-len-of-nat-of* [*simp*]:  
*min* (*LENGTH*('a)) (*nat-of* (*n*::('a::len) *sat*)) = *nat-of* *n*  
*<proof>*

**lemma** *min-nat-of-len-of* [*simp*]:  
*min* (*nat-of* (*n*::('a::len) *sat*)) (*LENGTH*('a)) = *nat-of* *n*  
*<proof>*

**lemma** *Abs-sat'-nat-of* [*simp*]:  
*Abs-sat'* (*nat-of* *n*) = *n*  
*<proof>*

**instantiation** *sat* :: (*len*) *linorder*  
**begin**

**definition**  
*less-eq-sat-def*:  $x \leq y \iff \text{nat-of } x \leq \text{nat-of } y$

**definition**  
*less-sat-def*:  $x < y \iff \text{nat-of } x < \text{nat-of } y$

**instance**  
*<proof>*

**end**

**instantiation** *sat* :: (*len*) {*minus*, *comm-semiring-1*}

**begin**

**definition**  
 $0 = \text{Abs-sat}' 0$

**definition**  
 $1 = \text{Abs-sat}' 1$

**lemma** *nat-of-zero-sat* [*simp*, *code abstract*]:

$\text{nat-of } 0 = 0$   
 $\langle \text{proof} \rangle$

**lemma** *nat-of-one-sat* [*simp, code abstract*]:  
 $\text{nat-of } 1 = \min 1 (\text{LENGTH}('a))$   
 $\langle \text{proof} \rangle$

**definition**  
 $x + y = \text{Abs-sat}' (\text{nat-of } x + \text{nat-of } y)$

**lemma** *nat-of-plus-sat* [*simp, code abstract*]:  
 $\text{nat-of } (x + y) = \min (\text{nat-of } x + \text{nat-of } y) (\text{LENGTH}('a))$   
 $\langle \text{proof} \rangle$

**definition**  
 $x - y = \text{Abs-sat}' (\text{nat-of } x - \text{nat-of } y)$

**lemma** *nat-of-minus-sat* [*simp, code abstract*]:  
 $\text{nat-of } (x - y) = \text{nat-of } x - \text{nat-of } y$   
 $\langle \text{proof} \rangle$

**definition**  
 $x * y = \text{Abs-sat}' (\text{nat-of } x * \text{nat-of } y)$

**lemma** *nat-of-times-sat* [*simp, code abstract*]:  
 $\text{nat-of } (x * y) = \min (\text{nat-of } x * \text{nat-of } y) (\text{LENGTH}('a))$   
 $\langle \text{proof} \rangle$

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

**end**

**instantiation** *sat* :: (*len*) *ordered-comm-semiring*  
**begin**

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

**end**

**lemma** *Abs-sat'-eq-of-nat*:  $\text{Abs-sat}' n = \text{of-nat } n$   
 $\langle \text{proof} \rangle$

**abbreviation** *Sat* ::  $\text{nat} \Rightarrow 'a::\text{len } \text{sat}$  **where**  
 $\text{Sat} \equiv \text{of-nat}$

**lemma** *nat-of-Sat* [*simp*]:  
 $\text{nat-of } (\text{Sat } n :: ('a::\text{len}) \text{ sat}) = \min (\text{LENGTH}('a)) n$

⟨*proof*⟩

**lemma** [*code-abbrev*]:  
*of-nat* (numeral *k*) = (numeral *k* :: 'a::len sat)  
 ⟨*proof*⟩

**context**  
**begin**

**qualified definition** *sat-of-nat* :: nat ⇒ ('a::len) sat  
 where [*code-abbrev*]: *sat-of-nat* = *of-nat*

**lemma** [*code abstract*]:  
*nat-of* (sat-of-nat *n* :: ('a::len) sat) = min (LENGTH('a)) *n*  
 ⟨*proof*⟩

**end**

**instance** *sat* :: (len) finite  
 ⟨*proof*⟩

**instantiation** *sat* :: (len) equal  
**begin**

**definition** *HOL.equal* *A B* ⟷ *nat-of A* = *nat-of B*

**instance**  
 ⟨*proof*⟩

**end**

**instantiation** *sat* :: (len) {bounded-lattice, distrib-lattice}  
**begin**

**definition** (*inf* :: 'a sat ⇒ 'a sat ⇒ 'a sat) = min  
**definition** (*sup* :: 'a sat ⇒ 'a sat ⇒ 'a sat) = max  
**definition** *bot* = (0 :: 'a sat)  
**definition** *top* = Sat (LENGTH('a))

**instance**  
 ⟨*proof*⟩

**end**

**instantiation** *sat* :: (len) {Inf, Sup}  
**begin**

**global-interpretation** *Inf-sat*: semilattice-neutr-set min ⟨*top* :: 'a sat⟩  
 defines *Inf-sat* = *Inf-sat.F*

⟨proof⟩

**global-interpretation** *Sup-sat: semilattice-neutr-set max ⟨bot :: 'a sat⟩*  
**defines** *Sup-sat = Sup-sat.F*  
 ⟨proof⟩

**instance** ⟨proof⟩

**end**

**instance** *sat :: (len) complete-lattice*  
 ⟨proof⟩

**end**

## 95 Set Idioms

**theory** *Set-Idioms*  
**imports** *Countable-Set*

**begin**

### 95.1 Idioms for being a suitable union/intersection of something

**definition** *union-of :: ('a set set ⇒ bool) ⇒ ('a set ⇒ bool) ⇒ 'a set ⇒ bool*  
 (**infixr** ⟨union'-of⟩ 60)  
**where** *P union-of Q ≡ λS. ∃U. P U ∧ U ⊆ Collect Q ∧ ⋃U = S*

**definition** *intersection-of :: ('a set set ⇒ bool) ⇒ ('a set ⇒ bool) ⇒ 'a set ⇒ bool*  
 (**infixr** ⟨intersection'-of⟩ 60)  
**where** *P intersection-of Q ≡ λS. ∃U. P U ∧ U ⊆ Collect Q ∧ ⋂U = S*

**definition** *arbitrary:: 'a set set ⇒ bool* **where** *arbitrary U ≡ True*

**lemma** *union-of-inc: [[P {S}; Q S]] ⇒ (P union-of Q) S*  
 ⟨proof⟩

**lemma** *intersection-of-inc:*  
 [[P {S}; Q S]] ⇒ (P intersection-of Q) S  
 ⟨proof⟩

**lemma** *union-of-mono:*  
 [(P union-of Q) S; ∧x. Q x ⇒ Q' x] ⇒ (P union-of Q') S  
 ⟨proof⟩

**lemma** *intersection-of-mono:*  
 [(P intersection-of Q) S; ∧x. Q x ⇒ Q' x] ⇒ (P intersection-of Q') S  
 ⟨proof⟩

**lemma** *all-union-of*:

$$(\forall S. (P \text{ union-of } Q) S \longrightarrow R S) \longleftrightarrow (\forall T. P T \wedge T \subseteq \text{Collect } Q \longrightarrow R(\bigcup T))$$

*<proof>*

**lemma** *all-intersection-of*:

$$(\forall S. (P \text{ intersection-of } Q) S \longrightarrow R S) \longleftrightarrow (\forall T. P T \wedge T \subseteq \text{Collect } Q \longrightarrow R(\bigcap T))$$

*<proof>*

**lemma** *intersection-ofE*:

$$\llbracket (P \text{ intersection-of } Q) S; \bigwedge T. \llbracket P T; T \subseteq \text{Collect } Q \rrbracket \Longrightarrow R(\bigcap T) \rrbracket \Longrightarrow R S$$

*<proof>*

**lemma** *union-of-empty*:

$$P \{\} \Longrightarrow (P \text{ union-of } Q) \{\}$$

*<proof>*

**lemma** *intersection-of-empty*:

$$P \{\} \Longrightarrow (P \text{ intersection-of } Q) \text{ UNIV}$$

*<proof>*

The arbitrary and finite cases

**lemma** *arbitrary-union-of-alt*:

$$(\text{arbitrary union-of } Q) S \longleftrightarrow (\forall x \in S. \exists U. Q U \wedge x \in U \wedge U \subseteq S)$$

(**is** ?lhs = ?rhs)  
*<proof>*

**lemma** *arbitrary-union-of-empty [simp]*: (*arbitrary union-of*  $P$ )  $\{\}$

*<proof>*

**lemma** *arbitrary-intersection-of-empty [simp]*:

$$(\text{arbitrary intersection-of } P) \text{ UNIV}$$

*<proof>*

**lemma** *arbitrary-union-of-inc*:

$$P S \Longrightarrow (\text{arbitrary union-of } P) S$$

*<proof>*

**lemma** *arbitrary-intersection-of-inc*:

$$P S \Longrightarrow (\text{arbitrary intersection-of } P) S$$

*<proof>*

**lemma** *arbitrary-union-of-complement*:

$$(\text{arbitrary union-of } P) S \longleftrightarrow (\text{arbitrary intersection-of } (\lambda S. P(- S))) (- S)$$

(**is** ?lhs = ?rhs)  
*<proof>*

**lemma** *arbitrary-intersection-of-complement*:

$(\text{arbitrary intersection-of } P) S \longleftrightarrow (\text{arbitrary union-of } (\lambda S. P(- S))) (- S)$   
 ⟨proof⟩

**lemma** *arbitrary-union-of-idempot* [simp]:  
 $\text{arbitrary union-of arbitrary union-of } P = \text{arbitrary union-of } P$   
 ⟨proof⟩

**lemma** *arbitrary-intersection-of-idempot*:  
 $\text{arbitrary intersection-of arbitrary intersection-of } P = \text{arbitrary intersection-of } P$   
 (is ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *arbitrary-union-of-Union*:  
 $(\bigwedge S. S \in \mathcal{U} \implies (\text{arbitrary union-of } P) S) \implies (\text{arbitrary union-of } P) (\bigcup \mathcal{U})$   
 ⟨proof⟩

**lemma** *arbitrary-union-of-Un*:  
 $\llbracket (\text{arbitrary union-of } P) S; (\text{arbitrary union-of } P) T \rrbracket$   
 $\implies (\text{arbitrary union-of } P) (S \cup T)$   
 ⟨proof⟩

**lemma** *arbitrary-intersection-of-Inter*:  
 $(\bigwedge S. S \in \mathcal{U} \implies (\text{arbitrary intersection-of } P) S) \implies (\text{arbitrary intersection-of } P) (\bigcap \mathcal{U})$   
 ⟨proof⟩

**lemma** *arbitrary-intersection-of-Int*:  
 $\llbracket (\text{arbitrary intersection-of } P) S; (\text{arbitrary intersection-of } P) T \rrbracket$   
 $\implies (\text{arbitrary intersection-of } P) (S \cap T)$   
 ⟨proof⟩

**lemma** *arbitrary-union-of-Int-eq*:  
 $(\forall S T. (\text{arbitrary union-of } P) S \wedge (\text{arbitrary union-of } P) T$   
 $\longrightarrow (\text{arbitrary union-of } P) (S \cap T))$   
 $\longleftrightarrow (\forall S T. P S \wedge P T \longrightarrow (\text{arbitrary union-of } P) (S \cap T))$  (is ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *arbitrary-intersection-of-Un-eq*:  
 $(\forall S T. (\text{arbitrary intersection-of } P) S \wedge (\text{arbitrary intersection-of } P) T$   
 $\longrightarrow (\text{arbitrary intersection-of } P) (S \cup T)) \longleftrightarrow$   
 $(\forall S T. P S \wedge P T \longrightarrow (\text{arbitrary intersection-of } P) (S \cup T))$   
 ⟨proof⟩

**lemma** *finite-union-of-empty* [simp]:  $(\text{finite union-of } P) \{\}$   
 ⟨proof⟩

**lemma** *finite-intersection-of-empty* [simp]:  $(\text{finite intersection-of } P) UNIV$   
 ⟨proof⟩

**lemma** *finite-union-of-inc:*

$$P S \implies (\text{finite union-of } P) S$$

*<proof>*

**lemma** *finite-intersection-of-inc:*

$$P S \implies (\text{finite intersection-of } P) S$$

*<proof>*

**lemma** *finite-union-of-complement:*

$$(\text{finite union-of } P) S \longleftrightarrow (\text{finite intersection-of } (\lambda S. P(- S))) (- S)$$

*<proof>*

**lemma** *finite-intersection-of-complement:*

$$(\text{finite intersection-of } P) S \longleftrightarrow (\text{finite union-of } (\lambda S. P(- S))) (- S)$$

*<proof>*

**lemma** *finite-union-of-idempot [simp]:*

$$\text{finite union-of finite union-of } P = \text{finite union-of } P$$

*<proof>*

**lemma** *finite-intersection-of-idempot [simp]:*

$$\text{finite intersection-of finite intersection-of } P = \text{finite intersection-of } P$$

*<proof>*

**lemma** *finite-union-of-Union:*

$$\llbracket \text{finite } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{finite union-of } P) S \rrbracket \implies (\text{finite union-of } P) (\bigcup \mathcal{U})$$

*<proof>*

**lemma** *finite-union-of-Un:*

$$\llbracket (\text{finite union-of } P) S; (\text{finite union-of } P) T \rrbracket \implies (\text{finite union-of } P) (S \cup T)$$

*<proof>*

**lemma** *finite-intersection-of-Inter:*

$$\llbracket \text{finite } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{finite intersection-of } P) S \rrbracket \implies (\text{finite intersection-of } P) (\bigcap \mathcal{U})$$

*<proof>*

**lemma** *finite-intersection-of-Int:*

$$\llbracket (\text{finite intersection-of } P) S; (\text{finite intersection-of } P) T \rrbracket \implies (\text{finite intersection-of } P) (S \cap T)$$

*<proof>*

**lemma** *finite-union-of-Int-eq:*

$$(\forall S T. (\text{finite union-of } P) S \wedge (\text{finite union-of } P) T \longrightarrow (\text{finite union-of } P) (S \cap T))$$

$$\longleftrightarrow (\forall S T. P S \wedge P T \longrightarrow (\text{finite union-of } P) (S \cap T))$$

**(is ?lhs = ?rhs)**  
*<proof>*



**lemma** *finite-intersection-of-Un-eq*:

$$\begin{aligned} & (\forall S T. (\text{finite intersection-of } P) S \wedge \\ & \quad (\text{finite intersection-of } P) T \\ & \quad \longrightarrow (\text{finite intersection-of } P) (S \cup T)) \longleftrightarrow \\ & (\forall S T. P S \wedge P T \longrightarrow (\text{finite intersection-of } P) (S \cup T)) \\ & \langle \text{proof} \rangle \end{aligned}$$

**abbreviation** *finite'* :: 'a set  $\Rightarrow$  bool

**where** *finite'* A  $\equiv$  finite A  $\wedge$  A  $\neq$  {}

**lemma** *finite'-intersection-of-Int*:

$$\begin{aligned} & \llbracket (\text{finite}' \text{ intersection-of } P) S; (\text{finite}' \text{ intersection-of } P) T \rrbracket \\ & \implies (\text{finite}' \text{ intersection-of } P) (S \cap T) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *finite'-intersection-of-inc*:

$$\begin{aligned} & P S \implies (\text{finite}' \text{ intersection-of } P) S \\ & \langle \text{proof} \rangle \end{aligned}$$

## 95.2 The “Relative to” operator

A somewhat cheap but handy way of getting localized forms of various topological concepts (open, closed, borel, fsigma, gdelta etc.)

**definition** *relative-to* :: ['a set  $\Rightarrow$  bool, 'a set, 'a set]  $\Rightarrow$  bool (**infixl** *relative-to* 55)

**where** *P relative-to S*  $\equiv$   $\lambda T. \exists U. P U \wedge S \cap U = T$

**lemma** *relative-to-UNIV* [*simp*]: (*P relative-to UNIV*) S  $\longleftrightarrow$  P S

$\langle \text{proof} \rangle$

**lemma** *relative-to-imp-subset*:

$$\begin{aligned} & (P \text{ relative-to } S) T \implies T \subseteq S \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *all-relative-to*: ( $\forall S. (P \text{ relative-to } U) S \longrightarrow Q S$ )  $\longleftrightarrow$  ( $\forall S. P S \longrightarrow Q(U \cap S)$ )

$\langle \text{proof} \rangle$

**lemma** *relative-toE*:  $\llbracket (P \text{ relative-to } U) S; \bigwedge S. P S \implies Q(U \cap S) \rrbracket \implies Q S$

$\langle \text{proof} \rangle$

**lemma** *relative-to-inc*:

$$\begin{aligned} & P S \implies (P \text{ relative-to } U) (U \cap S) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *relative-to-relative-to* [*simp*]:

$$\begin{aligned} & P \text{ relative-to } S \text{ relative-to } T = P \text{ relative-to } (S \cap T) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *relative-to-compl*:

$$S \subseteq U \implies ((P \text{ relative-to } U) (U - S) \longleftrightarrow ((\lambda c. P(- c)) \text{ relative-to } U) S)$$

*<proof>*

**lemma** *relative-to-subset-trans*:

$$\llbracket (P \text{ relative-to } U) S; S \subseteq T; T \subseteq U \rrbracket \implies (P \text{ relative-to } T) S$$

*<proof>*

**lemma** *relative-to-mono*:

$$\llbracket (P \text{ relative-to } U) S; \bigwedge S. P S \implies Q S \rrbracket \implies (Q \text{ relative-to } U) S$$

*<proof>*

**lemma** *relative-to-subset-inc*:  $\llbracket S \subseteq U; P S \rrbracket \implies (P \text{ relative-to } U) S$

*<proof>*

**lemma** *relative-to-Int*:

$$\begin{aligned} \llbracket (P \text{ relative-to } S) C; (P \text{ relative-to } S) D; \bigwedge X Y. \llbracket P X; P Y \rrbracket \implies P(X \cap Y) \rrbracket \\ \implies (P \text{ relative-to } S) (C \cap D) \end{aligned}$$

*<proof>*

**lemma** *relative-to-Un*:

$$\begin{aligned} \llbracket (P \text{ relative-to } S) C; (P \text{ relative-to } S) D; \bigwedge X Y. \llbracket P X; P Y \rrbracket \implies P(X \cup Y) \rrbracket \\ \implies (P \text{ relative-to } S) (C \cup D) \end{aligned}$$

*<proof>*

**lemma** *arbitrary-union-of-relative-to*:

$$\begin{aligned} ((\text{arbitrary union-of } P) \text{ relative-to } U) = (\text{arbitrary union-of } (P \text{ relative-to } U)) \\ (\text{is ?lhs} = \text{?rhs}) \end{aligned}$$

*<proof>*

**lemma** *finite-union-of-relative-to*:

$$\begin{aligned} ((\text{finite union-of } P) \text{ relative-to } U) = (\text{finite union-of } (P \text{ relative-to } U)) \text{ (is ?lhs} \\ = \text{?rhs)} \end{aligned}$$

*<proof>*

**lemma** *countable-union-of-relative-to*:

$$\begin{aligned} ((\text{countable union-of } P) \text{ relative-to } U) = (\text{countable union-of } (P \text{ relative-to } U)) \\ (\text{is ?lhs} = \text{?rhs}) \end{aligned}$$

*<proof>*

**lemma** *arbitrary-intersection-of-relative-to*:

$$\begin{aligned} ((\text{arbitrary intersection-of } P) \text{ relative-to } U) = ((\text{arbitrary intersection-of } (P \text{ rel-} \\ \text{ative-to } U)) \text{ relative-to } U) \text{ (is ?lhs} = \text{?rhs)} \end{aligned}$$

*<proof>*

**lemma** *finite-intersection-of-relative-to*:

$$((\text{finite intersection-of } P) \text{ relative-to } U) = ((\text{finite intersection-of } (P \text{ relative-to } U))$$

$U$ ) relative-to  $U$ ) (is ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *countable-intersection-of-relative-to*:  
 ((countable intersection-of  $P$ ) relative-to  $U$ ) = ((countable intersection-of ( $P$  relative-to  $U$ )) relative-to  $U$ ) (is ?lhs = ?rhs)  
 ⟨proof⟩

**lemma** *countable-union-of-empty* [simp]: (countable union-of  $P$ ) {}  
 ⟨proof⟩

**lemma** *countable-intersection-of-empty* [simp]: (countable intersection-of  $P$ ) UNIV  
 ⟨proof⟩

**lemma** *countable-union-of-inc*:  $P S \implies$  (countable union-of  $P$ )  $S$   
 ⟨proof⟩

**lemma** *countable-intersection-of-inc*:  $P S \implies$  (countable intersection-of  $P$ )  $S$   
 ⟨proof⟩

**lemma** *countable-union-of-complement*:  
 (countable union-of  $P$ )  $S \longleftrightarrow$  (countable intersection-of ( $\lambda S. P(-S)$ )) ( $-S$ )  
 (is ?lhs=?rhs)  
 ⟨proof⟩

**lemma** *countable-intersection-of-complement*:  
 (countable intersection-of  $P$ )  $S \longleftrightarrow$  (countable union-of ( $\lambda S. P(-S)$ )) ( $-S$ )  
 ⟨proof⟩

**lemma** *countable-union-of-explicit*:  
 assumes  $P \{\}$   
 shows (countable union-of  $P$ )  $S \longleftrightarrow$   
 ( $\exists T. (\forall n::nat. P(T n)) \wedge \bigcup(\text{range } T) = S$ ) (is ?lhs=?rhs)  
 ⟨proof⟩

**lemma** *countable-union-of-ascending*:  
 assumes empty:  $P \{\}$  and  $Un: \bigwedge T U. \llbracket P T; P U \rrbracket \implies P(T \cup U)$   
 shows (countable union-of  $P$ )  $S \longleftrightarrow$   
 ( $\exists T. (\forall n. P(T n)) \wedge (\forall n. T n \subseteq T(\text{Suc } n)) \wedge \bigcup(\text{range } T) = S$ ) (is  
 ?lhs=?rhs)  
 ⟨proof⟩

**lemma** *countable-union-of-idem* [simp]:  
 countable union-of countable union-of  $P =$  countable union-of  $P$  (is ?lhs=?rhs)  
 ⟨proof⟩

**lemma** *countable-intersection-of-idem* [simp]:  
 countable intersection-of countable intersection-of  $P =$   
 countable intersection-of  $P$

*<proof>*

**lemma** *countable-union-of-Union:*

$\llbracket \text{countable } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{countable union-of } P) S \rrbracket$   
 $\implies (\text{countable union-of } P) (\bigcup \mathcal{U})$

*<proof>*

**lemma** *countable-union-of-UN:*

$\llbracket \text{countable } I; \bigwedge i. i \in I \implies (\text{countable union-of } P) (U i) \rrbracket$   
 $\implies (\text{countable union-of } P) (\bigcup_{i \in I} U i)$

*<proof>*

**lemma** *countable-union-of-Un:*

$\llbracket (\text{countable union-of } P) S; (\text{countable union-of } P) T \rrbracket$   
 $\implies (\text{countable union-of } P) (S \cup T)$

*<proof>*

**lemma** *countable-intersection-of-Inter:*

$\llbracket \text{countable } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{countable intersection-of } P) S \rrbracket$   
 $\implies (\text{countable intersection-of } P) (\bigcap \mathcal{U})$

*<proof>*

**lemma** *countable-intersection-of-INT:*

$\llbracket \text{countable } I; \bigwedge i. i \in I \implies (\text{countable intersection-of } P) (U i) \rrbracket$   
 $\implies (\text{countable intersection-of } P) (\bigcap_{i \in I} U i)$

*<proof>*

**lemma** *countable-intersection-of-inter:*

$\llbracket (\text{countable intersection-of } P) S; (\text{countable intersection-of } P) T \rrbracket$   
 $\implies (\text{countable intersection-of } P) (S \cap T)$

*<proof>*

**lemma** *countable-union-of-Int:*

**assumes**  $S: (\text{countable union-of } P) S$  **and**  $T: (\text{countable union-of } P) T$

**and**  $Int: \bigwedge S T. P S \wedge P T \implies P(S \cap T)$

**shows**  $(\text{countable union-of } P) (S \cap T)$

*<proof>*

**lemma** *countable-intersection-of-union:*

**assumes**  $S: (\text{countable intersection-of } P) S$  **and**  $T: (\text{countable intersection-of } P) T$

**and**  $Un: \bigwedge S T. P S \wedge P T \implies P(S \cup T)$

**shows**  $(\text{countable intersection-of } P) (S \cup T)$

*<proof>*

**end**

## 96 Signed division: negative results rounded towards zero rather than minus infinity.

```

theory Signed-Division
  imports Main
begin

class signed-divide =
  fixes signed-divide :: ‹'a ⇒ 'a ⇒ 'a› (infixl ‹sdiv› 70)

class signed-modulo =
  fixes signed-modulo :: ‹'a ⇒ 'a ⇒ 'a› (infixl ‹smod› 70)

class signed-division = comm-semiring-1-cancel + signed-divide + signed-modulo
+
  assumes sdiv-mult-smod-eq: ‹a sdiv b * b + a smod b = a›
begin

lemma mult-sdiv-smod-eq:
  ‹b * (a sdiv b) + a smod b = a›
  ‹proof›

lemma smod-sdiv-mult-eq:
  ‹a smod b + a sdiv b * b = a›
  ‹proof›

lemma smod-mult-sdiv-eq:
  ‹a smod b + b * (a sdiv b) = a›
  ‹proof›

lemma minus-sdiv-mult-eq-smod:
  ‹a - a sdiv b * b = a smod b›
  ‹proof›

lemma minus-mult-sdiv-eq-smod:
  ‹a - b * (a sdiv b) = a smod b›
  ‹proof›

lemma minus-smod-eq-sdiv-mult:
  ‹a - a smod b = a sdiv b * b›
  ‹proof›

lemma minus-smod-eq-mult-sdiv:
  ‹a - a smod b = b * (a sdiv b)›
  ‹proof›

end

```

The following specification of division is named “T-division” in [2]. It is

motivated by ISO C99, which in turn adopted the typical behavior of hardware modern in the beginning of the 1990ies; but note ISO C99 describes the instance on machine words, not mathematical integers.

**instantiation** *int* :: *signed-division*  
**begin**

**definition** *signed-divide-int* ::  $\langle int \Rightarrow int \Rightarrow int \rangle$   
**where**  $\langle k \text{ sdiv } l = \text{sgn } k * \text{sgn } l * (|k| \text{ div } |l|) \rangle$  **for**  $k \ l :: int$

**definition** *signed-modulo-int* ::  $\langle int \Rightarrow int \Rightarrow int \rangle$   
**where**  $\langle k \text{ smod } l = \text{sgn } k * (|k| \text{ mod } |l|) \rangle$  **for**  $k \ l :: int$

**instance**  $\langle proof \rangle$

**end**

**lemma** *divide-int-eq-signed-divide-int*:  
 $\langle k \text{ div } l = k \text{ sdiv } l - \text{of-bool } (l \neq 0 \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$   
**for**  $k \ l :: int$   
 $\langle proof \rangle$

**lemma** *signed-divide-int-eq-divide-int*:  
 $\langle k \text{ sdiv } l = k \text{ div } l + \text{of-bool } (l \neq 0 \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$   
**for**  $k \ l :: int$   
 $\langle proof \rangle$

**lemma** *modulo-int-eq-signed-modulo-int*:  
 $\langle k \text{ mod } l = k \text{ smod } l + l * \text{of-bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$   
**for**  $k \ l :: int$   
 $\langle proof \rangle$

**lemma** *signed-modulo-int-eq-modulo-int*:  
 $\langle k \text{ smod } l = k \text{ mod } l - l * \text{of-bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$   
**for**  $k \ l :: int$   
 $\langle proof \rangle$

**lemma** *sdiv-int-div-0*:  
 $(x :: int) \text{ sdiv } 0 = 0$   
 $\langle proof \rangle$

**lemma** *sdiv-int-0-div* [simp]:  
 $0 \text{ sdiv } (x :: int) = 0$   
 $\langle proof \rangle$

**lemma** *smod-int-alt-def*:  
 $(a :: int) \text{ smod } b = \text{sgn } (a) * (abs \ a \ \text{mod} \ abs \ b)$   
 $\langle proof \rangle$

**lemma** *int-sdiv-simps* [simp]:

$(a :: \text{int}) \text{ sdiv } 1 = a$   
 $(a :: \text{int}) \text{ sdiv } 0 = 0$   
 $(a :: \text{int}) \text{ sdiv } -1 = -a$   
 ⟨proof⟩

**lemma** *smod-int-mod-0* [simp]:

$x \text{ smod } (0 :: \text{int}) = x$   
 ⟨proof⟩

**lemma** *smod-int-0-mod* [simp]:

$0 \text{ smod } (x :: \text{int}) = 0$   
 ⟨proof⟩

**lemma** *sgn-sdiv-eq-sgn-mult*:

$a \text{ sdiv } b \neq 0 \implies \text{sgn } ((a :: \text{int}) \text{ sdiv } b) = \text{sgn } (a * b)$   
 ⟨proof⟩

**lemma** *int-sdiv-same-is-1* [simp]:

**assumes**  $a \neq 0$   
**shows**  $((a :: \text{int}) \text{ sdiv } b = a) = (b = 1)$   
 ⟨proof⟩

**lemma** *int-sdiv-negated-is-minus1* [simp]:

$a \neq 0 \implies ((a :: \text{int}) \text{ sdiv } b = -a) = (b = -1)$   
 ⟨proof⟩

**lemma** *sdiv-int-range*:

$\langle a \text{ sdiv } b \in \{-|a|..|a|\} \rangle$  **for**  $a \ b :: \text{int}$   
 ⟨proof⟩

**lemma** *smod-int-range*:

$\langle a \text{ smod } b \in \{-|b| + 1..|b| - 1\} \rangle$   
**if**  $\langle b \neq 0 \rangle$  **for**  $a \ b :: \text{int}$   
 ⟨proof⟩

**lemma** *smod-int-compares*:

$\llbracket 0 \leq a; 0 < b \rrbracket \implies (a :: \text{int}) \text{ smod } b < b$   
 $\llbracket 0 \leq a; 0 < b \rrbracket \implies 0 \leq (a :: \text{int}) \text{ smod } b$   
 $\llbracket a \leq 0; 0 < b \rrbracket \implies -b < (a :: \text{int}) \text{ smod } b$   
 $\llbracket a \leq 0; 0 < b \rrbracket \implies (a :: \text{int}) \text{ smod } b \leq 0$   
 $\llbracket 0 \leq a; b < 0 \rrbracket \implies (a :: \text{int}) \text{ smod } b < -b$   
 $\llbracket 0 \leq a; b < 0 \rrbracket \implies 0 \leq (a :: \text{int}) \text{ smod } b$   
 $\llbracket a \leq 0; b < 0 \rrbracket \implies (a :: \text{int}) \text{ smod } b \leq 0$   
 $\llbracket a \leq 0; b < 0 \rrbracket \implies b \leq (a :: \text{int}) \text{ smod } b$   
 ⟨proof⟩

**lemma** *smod-mod-positive*:

$\llbracket 0 \leq (a :: \text{int}); 0 \leq b \rrbracket \implies a \text{ smod } b = a \text{ mod } b$   
 ⟨proof⟩

```

lemma minus-sdiv-eq [simp]:
  ⟨ $- k \text{ sdiv } l = - (k \text{ sdiv } l)$ ⟩ for  $k \ l :: \text{int}$ 
  ⟨proof⟩

lemma sdiv-minus-eq [simp]:
  ⟨ $k \text{ sdiv } - l = - (k \text{ sdiv } l)$ ⟩ for  $k \ l :: \text{int}$ 
  ⟨proof⟩

lemma sdiv-int-numeral-numeral [simp]:
  ⟨ $\text{numeral } m \text{ sdiv numeral } n = \text{numeral } m \text{ div } (\text{numeral } n :: \text{int})$ ⟩
  ⟨proof⟩

lemma minus-smod-eq [simp]:
  ⟨ $- k \text{ smod } l = - (k \text{ smod } l)$ ⟩ for  $k \ l :: \text{int}$ 
  ⟨proof⟩

lemma smod-minus-eq [simp]:
  ⟨ $k \text{ smod } - l = k \text{ smod } l$ ⟩ for  $k \ l :: \text{int}$ 
  ⟨proof⟩

lemma smod-int-numeral-numeral [simp]:
  ⟨ $\text{numeral } m \text{ smod numeral } n = \text{numeral } m \text{ mod } (\text{numeral } n :: \text{int})$ ⟩
  ⟨proof⟩

end

```

## 97 State monad

```

theory State-Monad
imports Monad-Syntax
begin

datatype ( $'s, 'a$ ) state = State (run-state:  $'s \Rightarrow ('a \times 's)$ )

lemma set-state-iff:  $x \in \text{set-state } m \iff (\exists s \ s'. \text{run-state } m \ s = (x, s'))$ 
  ⟨proof⟩

lemma pred-stateI[intro]:
  assumes  $\bigwedge a \ s \ s'. \text{run-state } m \ s = (a, s') \implies P \ a$ 
  shows pred-state  $P \ m$ 
  ⟨proof⟩

lemma pred-stateD[dest]:
  assumes pred-state  $P \ m$   $\text{run-state } m \ s = (a, s')$ 
  shows  $P \ a$ 
  ⟨proof⟩

lemma pred-state-run-state: pred-state  $P \ m \implies P \ (\text{fst } (\text{run-state } m \ s))$ 

```



*<proof>*

**definition** *state-io-rel* :: (*'s* ⇒ *'s* ⇒ *bool*) ⇒ (*'s*, *'a*) *state* ⇒ *bool* **where**  
*state-io-rel* *P m* = (∀ *s*. *P s* (snd (run-state *m s*)))

**lemma** *state-io-relI[intro]*:

assumes  $\bigwedge a\ s\ s'.\ \text{run-state } m\ s = (a, s') \implies P\ s\ s'$

shows *state-io-rel* *P m*

*<proof>*

**lemma** *state-io-relD[dest]*:

assumes *state-io-rel* *P m* run-state *m s* = (*a*, *s'*)

shows *P s s'*

*<proof>*

**lemma** *state-io-rel-mono[mono]*:  $P \leq Q \implies \text{state-io-rel } P \leq \text{state-io-rel } Q$

*<proof>*

**lemma** *state-ext*:

assumes  $\bigwedge s.\ \text{run-state } m\ s = \text{run-state } n\ s$

shows *m* = *n*

*<proof>*

**context begin**

**qualified definition** *return* :: *'a* ⇒ (*'s*, *'a*) *state* **where**

*return a* = *State (Pair a)*

**lemma** *run-state-return[simp]*: run-state (*return x*) *s* = (*x*, *s*)

*<proof>* **definition** *ap* :: (*'s*, *'a* ⇒ *'b*) *state* ⇒ (*'s*, *'a*) *state* ⇒ (*'s*, *'b*) *state* **where**  
*ap f x* = *State* ( $\lambda s.\ \text{case run-state } f\ s\ \text{of } (g, s') \Rightarrow \text{case run-state } x\ s'\ \text{of } (y, s'') \Rightarrow (g\ y, s'')$ )

**lemma** *run-state-ap[simp]*:

run-state (*ap f x*) *s* = (*case run-state f s of* (*g*, *s'*) ⇒ *case run-state x s' of* (*y*, *s''*) ⇒ (*g y*, *s''*))

*<proof>* **definition** *bind* :: (*'s*, *'a*) *state* ⇒ (*'a* ⇒ (*'s*, *'b*) *state*) ⇒ (*'s*, *'b*) *state* **where**

*bind x f* = *State* ( $\lambda s.\ \text{case run-state } x\ s\ \text{of } (a, s') \Rightarrow \text{run-state } (f\ a)\ s'$ )

**lemma** *run-state-bind[simp]*:

run-state (*bind x f*) *s* = (*case run-state x s of* (*a*, *s'*) ⇒ *run-state (f a) s'*)

*<proof>*

**adhoc-overloading** *Monad-Syntax.bind* ⇒ *bind*

**lemma** *bind-left-identity[simp]*: *bind* (*return a*) *f* = *f a*

*<proof>*

**lemma** *bind-right-identity[simp]*:  $\text{bind } m \text{ return} = m$   
 ⟨proof⟩

**lemma** *bind-assoc[simp]*:  $\text{bind } (\text{bind } m \text{ } f) \text{ } g = \text{bind } m \text{ } (\lambda x. \text{bind } (f \ x) \ g)$   
 ⟨proof⟩

**lemma** *bind-predI[intro]*:  
**assumes** *pred-state*  $(\lambda x. \text{pred-state } P \ (f \ x)) \ m$   
**shows** *pred-state*  $P \ (\text{bind } m \ f)$   
 ⟨proof⟩ **definition** *get* ::  $('s, 's) \text{ state}$  **where**  
*get* = *State*  $(\lambda s. (s, s))$

**lemma** *run-state-get[simp]*:  $\text{run-state } \text{get} \ s = (s, s)$   
 ⟨proof⟩ **definition** *set* ::  $'s \Rightarrow ('s, \text{unit}) \text{ state}$  **where**  
*set*  $s' = \text{State } (\lambda-. ((), s'))$

**lemma** *run-state-set[simp]*:  $\text{run-state } (\text{set } s') \ s = ((), s')$   
 ⟨proof⟩

**lemma** *get-set[simp]*:  $\text{bind } \text{get} \ \text{set} = \text{return } ()$   
 ⟨proof⟩

**lemma** *set-set[simp]*:  $\text{bind } (\text{set } s) \ (\lambda-. \text{set } s') = \text{set } s'$   
 ⟨proof⟩

**lemma** *get-bind-set[simp]*:  $\text{bind } \text{get} \ (\lambda s. \text{bind } (\text{set } s) \ (f \ s)) = \text{bind } \text{get} \ (\lambda s. f \ s \ ())$   
 ⟨proof⟩

**lemma** *get-const[simp]*:  $\text{bind } \text{get} \ (\lambda-. m) = m$   
 ⟨proof⟩

**fun** *traverse-list* ::  $('a \Rightarrow ('b, 'c) \text{ state}) \Rightarrow 'a \text{ list} \Rightarrow ('b, 'c) \text{ list} \text{ state}$  **where**  
*traverse-list* - [] = *return* [] |  
*traverse-list*  $f \ (x \# \ xs) = \text{do} \{$   
    $x \leftarrow f \ x;$   
    $\xs \leftarrow \text{traverse-list } f \ \xs;$   
    $\text{return } (x \# \ xs)$   
 $\}$

**lemma** *traverse-list-app[simp]*:  $\text{traverse-list } f \ (\xs \ @ \ ys) = \text{do} \{$   
    $\xs \leftarrow \text{traverse-list } f \ \xs;$   
    $\ys \leftarrow \text{traverse-list } f \ \ys;$   
    $\text{return } (\xs \ @ \ ys)$   
 $\}$   
 ⟨proof⟩

**lemma** *traverse-comp[simp]*:  $\text{traverse-list } (g \circ f) \ \xs = \text{traverse-list } g \ (\text{map } f \ \xs)$   
 ⟨proof⟩

**abbreviation** *mono-state* :: ('s::preorder, 'a) state  $\Rightarrow$  bool **where**  
*mono-state*  $\equiv$  state-io-rel ( $\leq$ )

**abbreviation** *strict-mono-state* :: ('s::preorder, 'a) state  $\Rightarrow$  bool **where**  
*strict-mono-state*  $\equiv$  state-io-rel ( $<$ )

**corollary** *strict-mono-implies-mono*: *strict-mono-state*  $m \Longrightarrow$  *mono-state*  $m$   
 $\langle$ proof $\rangle$

**lemma** *return-mono*[simp, intro]: *mono-state* (return  $x$ )  
 $\langle$ proof $\rangle$

**lemma** *get-mono*[simp, intro]: *mono-state* get  
 $\langle$ proof $\rangle$

**lemma** *put-mono*:  
 assumes  $\bigwedge x. s' \geq x$   
 shows *mono-state* (set  $s'$ )  
 $\langle$ proof $\rangle$

**lemma** *map-mono*[intro]: *mono-state*  $m \Longrightarrow$  *mono-state* (map-state  $f$   $m$ )  
 $\langle$ proof $\rangle$

**lemma** *map-strict-mono*[intro]: *strict-mono-state*  $m \Longrightarrow$  *strict-mono-state* (map-state  
 $f$   $m$ )  
 $\langle$ proof $\rangle$

**lemma** *bind-mono-strong*:  
 assumes *mono-state*  $m$   
 assumes  $\bigwedge x s s'. \text{run-state } m s = (x, s') \Longrightarrow$  *mono-state* ( $f$   $x$ )  
 shows *mono-state* (bind  $m$   $f$ )  
 $\langle$ proof $\rangle$

**lemma** *bind-strict-mono-strong1*:  
 assumes *mono-state*  $m$   
 assumes  $\bigwedge x s s'. \text{run-state } m s = (x, s') \Longrightarrow$  *strict-mono-state* ( $f$   $x$ )  
 shows *strict-mono-state* (bind  $m$   $f$ )  
 $\langle$ proof $\rangle$

**lemma** *bind-strict-mono-strong2*:  
 assumes *strict-mono-state*  $m$   
 assumes  $\bigwedge x s s'. \text{run-state } m s = (x, s') \Longrightarrow$  *mono-state* ( $f$   $x$ )  
 shows *strict-mono-state* (bind  $m$   $f$ )  
 $\langle$ proof $\rangle$

**corollary** *bind-strict-mono-strong*:  
 assumes *strict-mono-state*  $m$   
 assumes  $\bigwedge x s s'. \text{run-state } m s = (x, s') \Longrightarrow$  *strict-mono-state* ( $f$   $x$ )  
 shows *strict-mono-state* (bind  $m$   $f$ )

*<proof>* **definition** *update* :: ('s ⇒ 's) ⇒ ('s, unit) state **where**  
*update f = bind get (set ∘ f)*

**lemma** *update-id[simp]*: *update (λx. x) = return ()*  
*<proof>*

**lemma** *update-comp[simp]*: *bind (update f) (λ-. update g) = update (g ∘ f)*  
*<proof>*

**lemma** *set-update[simp]*: *bind (set s) (λ-. update f) = set (f s)*  
*<proof>*

**lemma** *set-bind-update[simp]*: *bind (set s) (λ-. bind (update f) g) = bind (set (f s)) g*  
*<proof>*

**lemma** *update-mono*:  
**assumes**  $\bigwedge x. x \leq f x$   
**shows** *mono-state (update f)*  
*<proof>*

**lemma** *update-strict-mono*:  
**assumes**  $\bigwedge x. x < f x$   
**shows** *strict-mono-state (update f)*  
*<proof>*

**end**

**end**

**theory** *Comparator*  
**imports** *Main*  
**begin**

## 98 Comparators on linear quasi-orders

### 98.1 Basic properties

**datatype** *comp* = *Less* | *Equiv* | *Greater*

**locale** *comparator* =  
**fixes** *cmp* :: 'a ⇒ 'a ⇒ *comp*  
**assumes** *refl [simp]*:  $\bigwedge a. cmp a a = Equiv$   
**and** *trans-equiv*:  $\bigwedge a b c. cmp a b = Equiv \implies cmp b c = Equiv \implies cmp a c = Equiv$   
**assumes** *trans-less*:  $cmp a b = Less \implies cmp b c = Less \implies cmp a c = Less$   
**and** *greater-iff-sym-less*:  $\bigwedge b a. cmp b a = Greater \iff cmp a b = Less$   
**begin**

Dual properties

**lemma** *trans-greater*:

$cmp\ a\ c = Greater$  **if**  $cmp\ a\ b = Greater$   $cmp\ b\ c = Greater$   
 ⟨proof⟩

**lemma** *less-iff-sym-greater*:

$cmp\ b\ a = Less \longleftrightarrow cmp\ a\ b = Greater$   
 ⟨proof⟩

The equivalence part

**lemma** *sym*:

$cmp\ b\ a = Equiv \longleftrightarrow cmp\ a\ b = Equiv$   
 ⟨proof⟩

**lemma** *reflp*:

$reflp\ (\lambda a\ b. cmp\ a\ b = Equiv)$   
 ⟨proof⟩

**lemma** *symp*:

$symp\ (\lambda a\ b. cmp\ a\ b = Equiv)$   
 ⟨proof⟩

**lemma** *transp*:

$transp\ (\lambda a\ b. cmp\ a\ b = Equiv)$   
 ⟨proof⟩

**lemma** *equivp*:

$equivp\ (\lambda a\ b. cmp\ a\ b = Equiv)$   
 ⟨proof⟩

The strict part

**lemma** *irreflp-less*:

$irreflp\ (\lambda a\ b. cmp\ a\ b = Less)$   
 ⟨proof⟩

**lemma** *irreflp-greater*:

$irreflp\ (\lambda a\ b. cmp\ a\ b = Greater)$   
 ⟨proof⟩

**lemma** *asym-less*:

$cmp\ b\ a \neq Less$  **if**  $cmp\ a\ b = Less$   
 ⟨proof⟩

**lemma** *asym-greater*:

$cmp\ b\ a \neq Greater$  **if**  $cmp\ a\ b = Greater$   
 ⟨proof⟩

**lemma** *asym-less*:

$asym\ (\lambda a\ b. cmp\ a\ b = Less)$

*<proof>*

**lemma** *asympt-greater:*

*asympt* ( $\lambda a b. \text{cmp } a b = \text{Greater}$ )

*<proof>*

**lemma** *trans-equiv-less:*

*cmp*  $a c = \text{Less}$  **if** *cmp*  $a b = \text{Equiv}$  **and** *cmp*  $b c = \text{Less}$

*<proof>*

**lemma** *trans-less-equiv:*

*cmp*  $a c = \text{Less}$  **if** *cmp*  $a b = \text{Less}$  **and** *cmp*  $b c = \text{Equiv}$

*<proof>*

**lemma** *trans-equiv-greater:*

*cmp*  $a c = \text{Greater}$  **if** *cmp*  $a b = \text{Equiv}$  **and** *cmp*  $b c = \text{Greater}$

*<proof>*

**lemma** *trans-greater-equiv:*

*cmp*  $a c = \text{Greater}$  **if** *cmp*  $a b = \text{Greater}$  **and** *cmp*  $b c = \text{Equiv}$

*<proof>*

**lemma** *transp-less:*

*transp* ( $\lambda a b. \text{cmp } a b = \text{Less}$ )

*<proof>*

**lemma** *transp-greater:*

*transp* ( $\lambda a b. \text{cmp } a b = \text{Greater}$ )

*<proof>*

The reflexive part

**lemma** *reflp-not-less:*

*reflp* ( $\lambda a b. \text{cmp } a b \neq \text{Less}$ )

*<proof>*

**lemma** *reflp-not-greater:*

*reflp* ( $\lambda a b. \text{cmp } a b \neq \text{Greater}$ )

*<proof>*

**lemma** *quasisym-not-less:*

*cmp*  $a b = \text{Equiv}$  **if** *cmp*  $a b \neq \text{Less}$  **and** *cmp*  $b a \neq \text{Less}$

*<proof>*

**lemma** *quasisym-not-greater:*

*cmp*  $a b = \text{Equiv}$  **if** *cmp*  $a b \neq \text{Greater}$  **and** *cmp*  $b a \neq \text{Greater}$

*<proof>*

**lemma** *trans-not-less:*

*cmp*  $a c \neq \text{Less}$  **if** *cmp*  $a b \neq \text{Less}$  *cmp*  $b c \neq \text{Less}$

⟨proof⟩

**lemma** *trans-not-greater*:

*cmp a c ≠ Greater* **if** *cmp a b ≠ Greater* *cmp b c ≠ Greater*  
 ⟨proof⟩

**lemma** *transp-not-less*:

*transp (λa b. cmp a b ≠ Less)*  
 ⟨proof⟩

**lemma** *transp-not-greater*:

*transp (λa b. cmp a b ≠ Greater)*  
 ⟨proof⟩

Substitution under equivalences

**lemma** *equiv-subst-left*:

*cmp z y = comp*  $\longleftrightarrow$  *cmp x y = comp* **if** *cmp z x = Equiv* **for** *comp*  
 ⟨proof⟩

**lemma** *equiv-subst-right*:

*cmp x z = comp*  $\longleftrightarrow$  *cmp x y = comp* **if** *cmp z y = Equiv* **for** *comp*  
 ⟨proof⟩

**end**

**typedef** *'a comparator* = {*cmp* :: *'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *comp. comparator cmp*}

**morphisms** *compare Abs-comparator*

⟨proof⟩

**setup-lifting** *type-definition-comparator*

**global-interpretation** *compare: comparator compare cmp*

⟨proof⟩

**lift-definition** *flat* :: *'a comparator*

**is**  $\lambda-$  -. *Equiv* ⟨proof⟩

**instantiation** *comparator* :: (*linorder*) *default*

**begin**

**lift-definition** *default-comparator* :: *'a comparator*

**is**  $\lambda x y.$  *if*  $x < y$  *then Less* *else if*  $x > y$  *then Greater* *else Equiv*  
 ⟨proof⟩

**instance** ⟨proof⟩

**end**

A rudimentary quickcheck setup

**instantiation** *comparator* :: (*enum*) *equal*

**begin**

**lift-definition** *equal-comparator* :: 'a comparator  $\Rightarrow$  'a comparator  $\Rightarrow$  bool  
 is  $\lambda f g. \forall x \in \text{set Enum.enum. } f x = g x$  *<proof>*

**instance**  
*<proof>*

**end**

**lemma** [*code*]:  
*HOL.equal cmp1 cmp2*  $\longleftrightarrow$  *Enum.enum-all* ( $\lambda x. \text{compare } cmp1 x = \text{compare } cmp2 x$ )  
*<proof>*

**lemma** [*code nbe*]:  
*HOL.equal* (*cmp* :: 'a::enum comparator) *cmp*  $\longleftrightarrow$  *True*  
*<proof>*

**instantiation** *comparator* :: (*linorder, typerep*) *full-exhaustive*  
**begin**

**definition** *full-exhaustive-comparator* ::  
 ('a comparator  $\times$  (unit  $\Rightarrow$  term)  $\Rightarrow$  (bool  $\times$  term list) option)  
 $\Rightarrow$  natural  $\Rightarrow$  (bool  $\times$  term list) option  
**where** *full-exhaustive-comparator f s* =  
*Quickcheck-Exhaustive.orelse*  
 (*f flat*, ( $\lambda u. \text{Code-Evaluation.Const (STR "Comparator.flat") TYPEREPEP('a$   
*comparator))*))  
 (*f (default*, ( $\lambda u. \text{Code-Evaluation.Const (STR "HOL.default-class.default")$   
*TYPEREPEP('a comparator))*))

**instance** *<proof>*

**end**

## 98.2 Fundamental comparator combinators

**lift-definition** *reversed* :: 'a comparator  $\Rightarrow$  'a comparator  
 is  $\lambda \text{cmp } a b. \text{cmp } b a$   
*<proof>*

**lift-definition** *key* :: ('b  $\Rightarrow$  'a)  $\Rightarrow$  'a comparator  $\Rightarrow$  'b comparator  
 is  $\lambda f \text{cmp } a b. \text{cmp } (f a) (f b)$   
*<proof>*



### 98.3 Direct implementations for linear orders on selected types

**definition** *comparator-bool* :: *bool comparator*  
**where** [*simp*, *code-abbrev*]: *comparator-bool* = *default*

**lemma** *compare-comparator-bool* [*code abstract*]:  
*compare comparator-bool* = ( $\lambda p q.$   
*if p then if q then Equiv else Greater*  
*else if q then Less else Equiv*)  
 ⟨*proof*⟩

**definition** *raw-comparator-nat* :: *nat ⇒ nat ⇒ comp*  
**where** [*simp*]: *raw-comparator-nat* = *compare default*

**lemma** *default-comparator-nat* [*simp*, *code*]:  
*raw-comparator-nat* (0::nat) 0 = *Equiv*  
*raw-comparator-nat* (Suc m) 0 = *Greater*  
*raw-comparator-nat* 0 (Suc n) = *Less*  
*raw-comparator-nat* (Suc m) (Suc n) = *raw-comparator-nat* m n  
 ⟨*proof*⟩

**definition** *comparator-nat* :: *nat comparator*  
**where** [*simp*, *code-abbrev*]: *comparator-nat* = *default*

**lemma** *compare-comparator-nat* [*code abstract*]:  
*compare comparator-nat* = *raw-comparator-nat*  
 ⟨*proof*⟩

**definition** *comparator-linordered-group* :: '*a*::*linordered-ab-group-add comparator*  
**where** [*simp*, *code-abbrev*]: *comparator-linordered-group* = *default*

**lemma** *comparator-linordered-group* [*code abstract*]:  
*compare comparator-linordered-group* = ( $\lambda a b.$   
*let c = a - b in if c < 0 then Less*  
*else if c = 0 then Equiv else Greater*)  
 ⟨*proof*⟩

**end**

**theory** *Sorting-Algorithms*  
**imports** *Main Multiset Comparator*  
**begin**

## 99 Stably sorted lists

**abbreviation** (*input*) *stable-segment* :: '*a comparator* ⇒ '*a* ⇒ '*a list* ⇒ '*a list*  
**where** *stable-segment cmp* x ≡ *filter* ( $\lambda y. compare\ cmp\ x\ y = Equiv$ )

**fun** *sorted* :: 'a comparator  $\Rightarrow$  'a list  $\Rightarrow$  bool  
**where** *sorted-Nil*: sorted cmp []  $\longleftrightarrow$  True  
| *sorted-single*: sorted cmp [x]  $\longleftrightarrow$  True  
| *sorted-rec*: sorted cmp (y # x # xs)  $\longleftrightarrow$  compare cmp y x  $\neq$  Greater  $\wedge$  sorted  
cmp (x # xs)

**lemma** *sorted-ConsI*:  
sorted cmp (x # xs) **if** sorted cmp xs  
**and**  $\bigwedge y$  ys. xs = y # ys  $\implies$  compare cmp x y  $\neq$  Greater  
⟨proof⟩

**lemma** *sorted-Cons-imp-sorted*:  
sorted cmp xs **if** sorted cmp (x # xs)  
⟨proof⟩

**lemma** *sorted-Cons-imp-not-less*:  
compare cmp y x  $\neq$  Greater **if** sorted cmp (y # xs)  
**and**  $x \in \text{set } xs$   
⟨proof⟩

**lemma** *sorted-induct* [consumes 1, case-names Nil Cons, induct pred: sorted]:  
P xs **if** sorted cmp xs **and** P []  
**and** \*:  $\bigwedge x$  xs. sorted cmp xs  $\implies$  P xs  
 $\implies$  ( $\bigwedge y$ . y  $\in \text{set } xs \implies$  compare cmp x y  $\neq$  Greater)  $\implies$  P (x # xs)  
⟨proof⟩

**lemma** *sorted-induct-remove1* [consumes 1, case-names Nil minimum]:  
P xs **if** sorted cmp xs **and** P []  
**and** \*:  $\bigwedge x$  xs. sorted cmp xs  $\implies$  P (remove1 x xs)  
 $\implies$   $x \in \text{set } xs \implies$  hd (stable-segment cmp x xs) = x  $\implies$  ( $\bigwedge y$ . y  $\in \text{set } xs \implies$   
compare cmp x y  $\neq$  Greater)  
 $\implies$  P xs  
⟨proof⟩

**lemma** *sorted-remove1*:  
sorted cmp (remove1 x xs) **if** sorted cmp xs  
⟨proof⟩

**lemma** *sorted-stable-segment*:  
sorted cmp (stable-segment cmp x xs)  
⟨proof⟩

**primrec** *insort* :: 'a comparator  $\Rightarrow$  'a  $\Rightarrow$  'a list  $\Rightarrow$  'a list  
**where** *insort* cmp y [] = [y]  
| *insort* cmp y (x # xs) = (if compare cmp y x  $\neq$  Greater  
then y # x # xs  
else x # insort cmp y xs)

- lemma** *mset-insort* [*simp*]:  
 $mset (insort\ cmp\ x\ xs) = add\ mset\ x\ (mset\ xs)$   
 ⟨*proof*⟩
- lemma** *length-insort* [*simp*]:  
 $length (insort\ cmp\ x\ xs) = Suc\ (length\ xs)$   
 ⟨*proof*⟩
- lemma** *sorted-insort*:  
 $sorted\ cmp\ (insort\ cmp\ x\ xs)$  **if**  $sorted\ cmp\ xs$   
 ⟨*proof*⟩
- lemma** *stable-insort-equiv*:  
 $stable\ segment\ cmp\ y\ (insort\ cmp\ x\ xs) = x \# stable\ segment\ cmp\ y\ xs$   
**if**  $compare\ cmp\ y\ x = Equiv$   
 ⟨*proof*⟩
- lemma** *stable-insort-not-equiv*:  
 $stable\ segment\ cmp\ y\ (insort\ cmp\ x\ xs) \neq stable\ segment\ cmp\ y\ xs$   
**if**  $compare\ cmp\ y\ x \neq Equiv$   
 ⟨*proof*⟩
- lemma** *remove1-insort-same-eq* [*simp*]:  
 $remove1\ x\ (insort\ cmp\ x\ xs) = xs$   
 ⟨*proof*⟩
- lemma** *insort-eq-ConsI*:  
 $insort\ cmp\ x\ xs = x \# xs$   
**if**  $sorted\ cmp\ xs \wedge y. y \in set\ xs \implies compare\ cmp\ x\ y \neq Greater$   
 ⟨*proof*⟩
- lemma** *remove1-insort-not-same-eq* [*simp*]:  
 $remove1\ y\ (insort\ cmp\ x\ xs) = insort\ cmp\ x\ (remove1\ y\ xs)$   
**if**  $sorted\ cmp\ xs \wedge x \neq y$   
 ⟨*proof*⟩
- lemma** *insort-remove1-same-eq*:  
 $insort\ cmp\ x\ (remove1\ x\ xs) = xs$   
**if**  $sorted\ cmp\ xs$  **and**  $x \in set\ xs$  **and**  $hd\ (stable\ segment\ cmp\ x\ xs) = x$   
 ⟨*proof*⟩
- lemma** *sorted-append-iff*:  
 $sorted\ cmp\ (xs\ @\ ys) \iff sorted\ cmp\ xs \wedge sorted\ cmp\ ys$   
 $\wedge (\forall x \in set\ xs. \forall y \in set\ ys. compare\ cmp\ x\ y \neq Greater)$  (**is**  $?P \iff ?R \wedge ?S \wedge ?Q$ )  
 ⟨*proof*⟩
- definition** *sort* :: 'a comparator  $\Rightarrow$  'a list  $\Rightarrow$  'a list  
**where**  $sort\ cmp\ xs = foldr\ (insort\ cmp)\ xs\ []$

**lemma** *sort-simps* [*simp*]:

$sort\ cmp\ [] = []$   
 $sort\ cmp\ (x\ \#\ xs) = insert\ cmp\ x\ (sort\ cmp\ xs)$   
 ⟨*proof*⟩

**lemma** *mset-sort* [*simp*]:

$mset\ (sort\ cmp\ xs) = mset\ xs$   
 ⟨*proof*⟩

**lemma** *length-sort* [*simp*]:

$length\ (sort\ cmp\ xs) = length\ xs$   
 ⟨*proof*⟩

**lemma** *sorted-sort* [*simp*]:

$sorted\ cmp\ (sort\ cmp\ xs)$   
 ⟨*proof*⟩

**lemma** *stable-sort*:

$stable-segment\ cmp\ x\ (sort\ cmp\ xs) = stable-segment\ cmp\ x\ xs$   
 ⟨*proof*⟩

**lemma** *sort-remove1-eq* [*simp*]:

$sort\ cmp\ (remove1\ x\ xs) = remove1\ x\ (sort\ cmp\ xs)$   
 ⟨*proof*⟩

**lemma** *set-insort* [*simp*]:

$set\ (insert\ cmp\ x\ xs) = insert\ x\ (set\ xs)$   
 ⟨*proof*⟩

**lemma** *set-sort* [*simp*]:

$set\ (sort\ cmp\ xs) = set\ xs$   
 ⟨*proof*⟩

**lemma** *sort-eqI*:

$sort\ cmp\ ys = xs$   
**if** *permutation*:  $mset\ ys = mset\ xs$   
**and** *sorted*:  $sorted\ cmp\ xs$   
**and** *stable*:  $\bigwedge y. y \in set\ ys \implies$   
 $stable-segment\ cmp\ y\ ys = stable-segment\ cmp\ y\ xs$   
 ⟨*proof*⟩

**lemma** *filter-insort*:

$filter\ P\ (insert\ cmp\ x\ xs) = insert\ cmp\ x\ (filter\ P\ xs)$   
**if** *sorted*  $cmp\ xs$  **and**  $P\ x$   
 ⟨*proof*⟩

**lemma** *filter-insort-triv*:

$filter\ P\ (insert\ cmp\ x\ xs) = filter\ P\ xs$

**if**  $\neg P x$   
 ⟨proof⟩

**lemma** *filter-sort*:  
 $filter P (sort cmp xs) = sort cmp (filter P xs)$   
 ⟨proof⟩

## 100 Alternative sorting algorithms

### 100.1 Quicksort

**definition** *quicksort* :: 'a comparator  $\Rightarrow$  'a list  $\Rightarrow$  'a list  
**where** *quicksort-is-sort* [simp]: *quicksort* = *sort*

**lemma** *sort-by-quicksort*:  
 $sort = quicksort$   
 ⟨proof⟩

**lemma** *sort-by-quicksort-rec*:  
 $sort cmp xs = sort cmp [x \leftarrow xs. compare cmp x (xs ! (length xs div 2)) = Less]$   
 @ *stable-segment cmp (xs ! (length xs div 2)) xs*  
 @  $sort cmp [x \leftarrow xs. compare cmp x (xs ! (length xs div 2)) = Greater]$  (**is** - =  
 ?rhs)  
 ⟨proof⟩

**context**  
**begin**

**qualified definition** *partition* :: 'a comparator  $\Rightarrow$  'a  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\times$  'a list  
 $\times$  'a list  
**where** *partition cmp pivot xs* =  
 ( $[x \leftarrow xs. compare cmp x pivot = Less]$ , *stable-segment cmp pivot xs*,  $[x \leftarrow xs. compare cmp x pivot = Greater]$ )

**qualified lemma** *partition-code* [code]:  
 $partition cmp pivot [] = ([], [], [])$   
 $partition cmp pivot (x \# xs) =$   
 (let (lts, eqs, gts) = *partition cmp pivot xs*  
 in case compare cmp x pivot of  
 Less  $\Rightarrow$  (x # lts, eqs, gts)  
 | Equiv  $\Rightarrow$  (lts, x # eqs, gts)  
 | Greater  $\Rightarrow$  (lts, eqs, x # gts))  
 ⟨proof⟩

**lemma** *quicksort-code* [code]:  
 $quicksort cmp xs =$   
 (case xs of  
 []  $\Rightarrow$  []  
 | [x]  $\Rightarrow$  xs

```

| [x, y] ⇒ (if compare cmp x y ≠ Greater then xs else [y, x])
| - ⇒
  let (lts, eqs, gts) = partition cmp (xs ! (length xs div 2)) xs
  in quicksort cmp lts @ eqs @ quicksort cmp gts)
⟨proof⟩

```

**end**

## 100.2 Mergesort

**definition** *mergesort* :: 'a comparator ⇒ 'a list ⇒ 'a list  
**where** *mergesort-is-sort* [*simp*]: *mergesort* = *sort*

**lemma** *sort-by-mergesort*:

```

sort = mergesort
⟨proof⟩

```

**context**

```

fixes cmp :: 'a comparator

```

**begin**

**qualified function** *merge* :: 'a list ⇒ 'a list ⇒ 'a list

```

where merge [] ys = ys
| merge xs [] = xs
| merge (x # xs) (y # ys) = (if compare cmp x y = Greater
  then y # merge (x # xs) ys else x # merge xs (y # ys))
⟨proof⟩ termination ⟨proof⟩

```

**lemma** *mset-merge*:

```

mset (merge xs ys) = mset xs + mset ys
⟨proof⟩

```

**lemma** *merge-eq-Cons-imp*:

```

xs ≠ [] ∧ z = hd xs ∨ ys ≠ [] ∧ z = hd ys
if merge xs ys = z # zs
⟨proof⟩

```

**lemma** *filter-merge*:

```

filter P (merge xs ys) = merge (filter P xs) (filter P ys)
if sorted cmp xs and sorted cmp ys
⟨proof⟩

```

**lemma** *sorted-merge*:

```

sorted cmp (merge xs ys) if sorted cmp xs and sorted cmp ys
⟨proof⟩

```

**lemma** *merge-eq-appendI*:

```

merge xs ys = xs @ ys
if ∧x y. x ∈ set xs ⇒ y ∈ set ys ⇒ compare cmp x y ≠ Greater

```

*<proof>*

**lemma** *merge-stable-segments*:

*merge (stable-segment cmp l xs) (stable-segment cmp l ys) =*  
*stable-segment cmp l xs @ stable-segment cmp l ys*  
*<proof>*

**lemma** *sort-by-mergesort-rec*:

*sort cmp xs =*  
*merge (sort cmp (take (length xs div 2) xs))*  
*(sort cmp (drop (length xs div 2) xs)) (is - = ?rhs)*  
*<proof>*

**lemma** *mergesort-code* [code]:

*mergesort cmp xs =*  
*(case xs of*  
*[] => []*  
*[x] => xs*  
*[x, y] => (if compare cmp x y ≠ Greater then xs else [y, x])*  
*\_ =>*  
*let*  
*half = length xs div 2;*  
*ys = take half xs;*  
*zs = drop half xs*  
*in merge (mergesort cmp ys) (mergesort cmp zs))*  
*<proof>*

**end**

**end**

## 101 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming

**theory** *Sum-of-Squares*

**imports** *Complex-Main*

**begin**

*<ML>*

**end**

## 102 A table-based implementation of the reflexive transitive closure

**theory** *Transitive-Closure-Table*  
**imports** *Main*  
**begin**

**inductive** *rtrancl-path* :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a list ⇒ 'a ⇒ bool  
**for** *r* :: 'a ⇒ 'a ⇒ bool

**where**

*base*: *rtrancl-path* *r* *x* [] *x*

| *step*: *r* *x* *y* ⇒ *rtrancl-path* *r* *y* *ys* *z* ⇒ *rtrancl-path* *r* *x* (*y* # *ys*) *z*

**lemma** *rtranclp-eq-rtrancl-path*:  $r^{**} x y \longleftrightarrow (\exists xs. rtrancl-path r x xs y)$   
 ⟨*proof*⟩

**lemma** *rtrancl-path-trans*:

**assumes** *xy*: *rtrancl-path* *r* *x* *xs* *y*

**and** *yz*: *rtrancl-path* *r* *y* *ys* *z*

**shows** *rtrancl-path* *r* *x* (*xs* @ *ys*) *z* ⟨*proof*⟩

**lemma** *rtrancl-path-appendE*:

**assumes** *xz*: *rtrancl-path* *r* *x* (*xs* @ *y* # *ys*) *z*

**obtains** *rtrancl-path* *r* *x* (*xs* @ [*y*]) *y* **and** *rtrancl-path* *r* *y* *ys* *z*

⟨*proof*⟩

**lemma** *rtrancl-path-distinct*:

**assumes** *xy*: *rtrancl-path* *r* *x* *xs* *y*

**obtains** *xs'* **where** *rtrancl-path* *r* *x* *xs'* *y* **and** *distinct* (*x* # *xs'*) **and** *set* *xs'* ⊆ *set* *xs*

⟨*proof*⟩

**inductive** *rtrancl-tab* :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a ⇒ 'a ⇒ bool

**for** *r* :: 'a ⇒ 'a ⇒ bool

**where**

*base*: *rtrancl-tab* *r* *xs* *x* *x*

| *step*: *x* ∉ *set* *xs* ⇒ *r* *x* *y* ⇒ *rtrancl-tab* *r* (*x* # *xs*) *y* *z* ⇒ *rtrancl-tab* *r* *xs* *x* *z*

**lemma** *rtrancl-path-imp-rtrancl-tab*:

**assumes** *path*: *rtrancl-path* *r* *x* *xs* *y*

**and** *x*: *distinct* (*x* # *xs*)

**and** *ys*: (*{x}* ∪ *set* *xs*) ∩ *set* *ys* = {}

**shows** *rtrancl-tab* *r* *ys* *x* *y*

⟨*proof*⟩

**lemma** *rtrancl-tab-imp-rtrancl-path*:

**assumes** *tab*: *rtrancl-tab* *r* *ys* *x* *y*

**obtains** *xs* **where** *rtrancl-path* *r* *x* *xs* *y*

⟨*proof*⟩



**lemma** *rtranclp-eq-rtrancl-tab-nil*:  $r^{**} x y \longleftrightarrow rtrancl\text{-}tab\ r \ []\ x\ y$   
 <proof>

**declare** *rtranclp-rtrancl-eq* [code del]  
**declare** *rtranclp-eq-rtrancl-tab-nil* [THEN iffD2, code-pred-intro]

**code-pred** *rtranclp*  
 <proof>

**lemma** *rtrancl-path-Range*:  $\llbracket rtrancl\text{-}path\ R\ x\ xs\ y; z \in set\ xs \rrbracket \Longrightarrow Rangep\ R\ z$   
 <proof>

**lemma** *rtrancl-path-Range-end*:  $\llbracket rtrancl\text{-}path\ R\ x\ xs\ y; xs \neq [] \rrbracket \Longrightarrow Rangep\ R\ y$   
 <proof>

**lemma** *rtrancl-path-nth*:  
 $\llbracket rtrancl\text{-}path\ R\ x\ xs\ y; i < length\ xs \rrbracket \Longrightarrow R\ ((x \# xs) ! i)\ (xs ! i)$   
 <proof>

**lemma** *rtrancl-path-last*:  $\llbracket rtrancl\text{-}path\ R\ x\ xs\ y; xs \neq [] \rrbracket \Longrightarrow last\ xs = y$   
 <proof>

**lemma** *rtrancl-path-mono*:  
 $\llbracket rtrancl\text{-}path\ R\ x\ p\ y; \bigwedge x\ y. R\ x\ y \Longrightarrow S\ x\ y \rrbracket \Longrightarrow rtrancl\text{-}path\ S\ x\ p\ y$   
 <proof>

end

## 103 Binary Tree

**theory** *Tree*  
**imports** *Main*  
**begin**

**datatype** *'a tree* =  
 Leaf  $\langle \langle \rangle \rangle$  |  
 Node *'a tree* (value: *'a*) *'a tree* ( $\langle \langle indent=1\ notation=\langle mixfix\ Node \rangle \langle -, / -, / - \rangle \rangle$ )  
**datatype-compact** *tree*

**primrec** *left* :: *'a tree*  $\Rightarrow$  *'a tree* **where**  
*left* (Node *l v r*) = *l* |  
*left* Leaf = Leaf

**primrec** *right* :: *'a tree*  $\Rightarrow$  *'a tree* **where**  
*right* (Node *l v r*) = *r* |  
*right* Leaf = Leaf

Counting the number of leaves rather than nodes:

```
fun size1 :: 'a tree  $\Rightarrow$  nat where
size1  $\langle \rangle$  = 1 |
size1  $\langle l, x, r \rangle$  = size1 l + size1 r
```

```
fun subtrees :: 'a tree  $\Rightarrow$  'a tree set where
subtrees  $\langle \rangle$  = { $\langle \rangle$ } |
subtrees  $\langle l, a, r \rangle$  = { $\langle l, a, r \rangle$ }  $\cup$  subtrees l  $\cup$  subtrees r
```

```
fun mirror :: 'a tree  $\Rightarrow$  'a tree where
mirror  $\langle \rangle$  = Leaf |
mirror  $\langle l, x, r \rangle$  =  $\langle$ mirror r, x, mirror l $\rangle$ 
```

```
class height = fixes height :: 'a  $\Rightarrow$  nat
```

```
instantiation tree :: (type)height
begin
```

```
fun height-tree :: 'a tree  $\Rightarrow$  nat where
height Leaf = 0 |
height (Node l a r) = max (height l) (height r) + 1
```

```
instance  $\langle$ proof $\rangle$ 
```

```
end
```

```
fun min-height :: 'a tree  $\Rightarrow$  nat where
min-height Leaf = 0 |
min-height (Node l - r) = min (min-height l) (min-height r) + 1
```

```
fun complete :: 'a tree  $\Rightarrow$  bool where
complete Leaf = True |
complete (Node l x r) = (height l = height r  $\wedge$  complete l  $\wedge$  complete r)
```

Almost complete:

```
definition acomplete :: 'a tree  $\Rightarrow$  bool where
acomplete t = (height t - min-height t  $\leq$  1)
```

Weight balanced:

```
fun wbalanced :: 'a tree  $\Rightarrow$  bool where
wbalanced Leaf = True |
wbalanced (Node l x r) = (abs(int(size l) - int(size r))  $\leq$  1  $\wedge$  wbalanced l  $\wedge$ 
wbalanced r)
```

Internal path length:

```
fun ipl :: 'a tree  $\Rightarrow$  nat where
ipl Leaf = 0 |
ipl (Node l - r) = ipl l + size l + ipl r + size r
```

```
fun preorder :: 'a tree  $\Rightarrow$  'a list where
```

*preorder*  $\langle \rangle = []$  |  
*preorder*  $\langle l, x, r \rangle = x \# \text{preorder } l @ \text{preorder } r$

**fun** *inorder* :: 'a tree  $\Rightarrow$  'a list **where**  
*inorder*  $\langle \rangle = []$  |  
*inorder*  $\langle l, x, r \rangle = \text{inorder } l @ [x] @ \text{inorder } r$

A linear version avoiding append:

**fun** *inorder2* :: 'a tree  $\Rightarrow$  'a list  $\Rightarrow$  'a list **where**  
*inorder2*  $\langle \rangle xs = xs$  |  
*inorder2*  $\langle l, x, r \rangle xs = \text{inorder2 } l (x \# \text{inorder2 } r xs)$

**fun** *postorder* :: 'a tree  $\Rightarrow$  'a list **where**  
*postorder*  $\langle \rangle = []$  |  
*postorder*  $\langle l, x, r \rangle = \text{postorder } l @ \text{postorder } r @ [x]$

Binary Search Tree:

**fun** *bst-wrt* :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a tree  $\Rightarrow$  bool **where**  
*bst-wrt* *P*  $\langle \rangle \longleftrightarrow \text{True}$  |  
*bst-wrt* *P*  $\langle l, a, r \rangle \longleftrightarrow$   
 $(\forall x \in \text{set-tree } l. P x a) \wedge (\forall x \in \text{set-tree } r. P a x) \wedge \text{bst-wrt } P l \wedge \text{bst-wrt } P r$

**abbreviation** *bst* :: ('a::linorder) tree  $\Rightarrow$  bool **where**  
*bst*  $\equiv \text{bst-wrt } (<)$

**fun** (in *linorder*) *heap* :: 'a tree  $\Rightarrow$  bool **where**  
*heap* *Leaf* = *True* |  
*heap* (Node *l m r*) =  
 $((\forall x \in \text{set-tree } l \cup \text{set-tree } r. m \leq x) \wedge \text{heap } l \wedge \text{heap } r)$

### 103.1 map-tree

**lemma** *eq-map-tree-Leaf[simp]*: *map-tree* *f t* = *Leaf*  $\longleftrightarrow t = \text{Leaf}$   
 $\langle \text{proof} \rangle$

**lemma** *eq-Leaf-map-tree[simp]*: *Leaf* = *map-tree* *f t*  $\longleftrightarrow t = \text{Leaf}$   
 $\langle \text{proof} \rangle$

### 103.2 size

**lemma** *size1-size*: *size1* *t* = *size* *t* + 1  
 $\langle \text{proof} \rangle$

**lemma** *size1-ge0[simp]*: 0 < *size1* *t*  
 $\langle \text{proof} \rangle$

**lemma** *eq-size-0[simp]*: *size* *t* = 0  $\longleftrightarrow t = \text{Leaf}$   
 $\langle \text{proof} \rangle$

**lemma** *eq-0-size[simp]*: 0 = *size* *t*  $\longleftrightarrow t = \text{Leaf}$

*<proof>*

**lemma** *neq-Leaf-iff*:  $(t \neq \langle \rangle) = (\exists l a r. t = \langle l, a, r \rangle)$   
*<proof>*

**lemma** *size-map-tree[simp]*:  $\text{size} (\text{map-tree } f \ t) = \text{size } t$   
*<proof>*

**lemma** *size1-map-tree[simp]*:  $\text{size1} (\text{map-tree } f \ t) = \text{size1 } t$   
*<proof>*

### 103.3 *set-tree*

**lemma** *eq-set-tree-empty[simp]*:  $\text{set-tree } t = \{\} \longleftrightarrow t = \text{Leaf}$   
*<proof>*

**lemma** *eq-empty-set-tree[simp]*:  $\{\} = \text{set-tree } t \longleftrightarrow t = \text{Leaf}$   
*<proof>*

**lemma** *finite-set-tree[simp]*:  $\text{finite}(\text{set-tree } t)$   
*<proof>*

### 103.4 *subtrees*

**lemma** *neq-subtrees-empty[simp]*:  $\text{subtrees } t \neq \{\}$   
*<proof>*

**lemma** *neq-empty-subtrees[simp]*:  $\{\} \neq \text{subtrees } t$   
*<proof>*

**lemma** *size-subtrees*:  $s \in \text{subtrees } t \implies \text{size } s \leq \text{size } t$   
*<proof>*

**lemma** *set-treeE*:  $a \in \text{set-tree } t \implies \exists l r. \langle l, a, r \rangle \in \text{subtrees } t$   
*<proof>*

**lemma** *Node-notin-subtrees-if[simp]*:  $a \notin \text{set-tree } t \implies \text{Node } l \ a \ r \notin \text{subtrees } t$   
*<proof>*

**lemma** *in-set-tree-if*:  $\langle l, a, r \rangle \in \text{subtrees } t \implies a \in \text{set-tree } t$   
*<proof>*

### 103.5 *height and min-height*

**lemma** *eq-height-0[simp]*:  $\text{height } t = 0 \longleftrightarrow t = \text{Leaf}$   
*<proof>*

**lemma** *eq-0-height[simp]*:  $0 = \text{height } t \longleftrightarrow t = \text{Leaf}$   
*<proof>*

**lemma** *height-map-tree[simp]*:  $\text{height} (\text{map-tree } f \ t) = \text{height } t$   
 ⟨proof⟩

**lemma** *height-le-size-tree*:  $\text{height } t \leq \text{size } (t::'a \ \text{tree})$   
 ⟨proof⟩

**lemma** *size1-height*:  $\text{size1 } t \leq 2^{\text{height } (t::'a \ \text{tree})}$   
 ⟨proof⟩

**corollary** *size-height*:  $\text{size } t \leq 2^{\text{height } (t::'a \ \text{tree})} - 1$   
 ⟨proof⟩

**lemma** *height-subtrees*:  $s \in \text{subtrees } t \implies \text{height } s \leq \text{height } t$   
 ⟨proof⟩

**lemma** *min-height-le-height*:  $\text{min-height } t \leq \text{height } t$   
 ⟨proof⟩

**lemma** *min-height-map-tree[simp]*:  $\text{min-height} (\text{map-tree } f \ t) = \text{min-height } t$   
 ⟨proof⟩

**lemma** *min-height-size1*:  $2^{\text{min-height } t} \leq \text{size1 } t$   
 ⟨proof⟩

### 103.6 complete

**lemma** *complete-iff-height*:  $\text{complete } t \iff (\text{min-height } t = \text{height } t)$   
 ⟨proof⟩

**lemma** *size1-if-complete*:  $\text{complete } t \implies \text{size1 } t = 2^{\text{height } t}$   
 ⟨proof⟩

**lemma** *size-if-complete*:  $\text{complete } t \implies \text{size } t = 2^{\text{height } t} - 1$   
 ⟨proof⟩

**lemma** *size1-height-if-incomplete*:  
 $\neg \text{complete } t \implies \text{size1 } t < 2^{\text{height } t}$   
 ⟨proof⟩

**lemma** *complete-iff-min-height*:  $\text{complete } t \iff (\text{height } t = \text{min-height } t)$   
 ⟨proof⟩

**lemma** *min-height-size1-if-incomplete*:  
 $\neg \text{complete } t \implies 2^{\text{min-height } t} < \text{size1 } t$   
 ⟨proof⟩

**lemma** *complete-if-size1-height*:  $\text{size1 } t = 2^{\text{height } t} \implies \text{complete } t$   
 ⟨proof⟩

**lemma** *complete-if-size1-min-height*:  $\text{size1 } t = 2^{\wedge} \text{min-height } t \implies \text{complete } t$   
 ⟨proof⟩

**lemma** *complete-iff-size1*:  $\text{complete } t \longleftrightarrow \text{size1 } t = 2^{\wedge} \text{height } t$   
 ⟨proof⟩

### 103.7 *acomplete*

**lemma** *acomplete-subtreeL*:  $\text{acomplete } (\text{Node } l \ x \ r) \implies \text{acomplete } l$   
 ⟨proof⟩

**lemma** *acomplete-subtreeR*:  $\text{acomplete } (\text{Node } l \ x \ r) \implies \text{acomplete } r$   
 ⟨proof⟩

**lemma** *acomplete-subtrees*:  $\llbracket \text{acomplete } t; s \in \text{subtrees } t \rrbracket \implies \text{acomplete } s$   
 ⟨proof⟩

Balanced trees have optimal height:

**lemma** *acomplete-optimal*:  
**fixes**  $t :: 'a \text{ tree}$  **and**  $t' :: 'b \text{ tree}$   
**assumes**  $\text{acomplete } t$   $\text{size } t \leq \text{size } t'$  **shows**  $\text{height } t \leq \text{height } t'$   
 ⟨proof⟩

### 103.8 *wbalanced*

**lemma** *wbalanced-subtrees*:  $\llbracket \text{wbalanced } t; s \in \text{subtrees } t \rrbracket \implies \text{wbalanced } s$   
 ⟨proof⟩

### 103.9 *ipl*

The internal path length of a tree:

**lemma** *ipl-if-complete-int*:  
 $\text{complete } t \implies \text{int}(\text{ipl } t) = (\text{int}(\text{height } t) - 2) * 2^{\wedge}(\text{height } t) + 2$   
 ⟨proof⟩

### 103.10 List of entries

**lemma** *eq-inorder-Nil[simp]*:  $\text{inorder } t = [] \longleftrightarrow t = \text{Leaf}$   
 ⟨proof⟩

**lemmas** *eq-Nil-inorder[simp]* = *eq-inorder-Nil[THEN eq-iff-swap]*

**lemma** *set-inorder[simp]*:  $\text{set } (\text{inorder } t) = \text{set-tree } t$   
 ⟨proof⟩

**lemma** *preorder-eq-Nil-iff[simp]*:  $(\text{preorder } t = []) = (t = \langle \rangle)$   
 ⟨proof⟩

**lemmas** *Nil-eq-preorder-iff* [simp] = *preorder-eq-Nil-iff* [THEN *eq-iff-swap*]

**lemma** *preorder-eq-Cons-iff*:

$preorder\ t = x \# xs \longleftrightarrow (\exists\ l\ r.\ t = \langle l, x, r \rangle \wedge xs = preorder\ l @ preorder\ r)$   
 ⟨proof⟩

**lemmas** *Cons-eq-preorder-iff* = *preorder-eq-Cons-iff* [THEN *eq-iff-swap*]

**lemma** *set-preorder* [simp]:  $set\ (preorder\ t) = set-tree\ t$

⟨proof⟩

**lemma** *set-postorder* [simp]:  $set\ (postorder\ t) = set-tree\ t$

⟨proof⟩

**lemma** *length-preorder* [simp]:  $length\ (preorder\ t) = size\ t$

⟨proof⟩

**lemma** *length-inorder* [simp]:  $length\ (inorder\ t) = size\ t$

⟨proof⟩

**lemma** *length-postorder* [simp]:  $length\ (postorder\ t) = size\ t$

⟨proof⟩

**lemma** *preorder-map*:  $preorder\ (map-tree\ f\ t) = map\ f\ (preorder\ t)$

⟨proof⟩

**lemma** *inorder-map*:  $inorder\ (map-tree\ f\ t) = map\ f\ (inorder\ t)$

⟨proof⟩

**lemma** *postorder-map*:  $postorder\ (map-tree\ f\ t) = map\ f\ (postorder\ t)$

⟨proof⟩

**lemma** *inorder2-inorder*:  $inorder2\ t\ xs = inorder\ t @ xs$

⟨proof⟩

### 103.11 Binary Search Tree

**lemma** *bst-wrt-mono*:  $(\bigwedge x\ y.\ P\ x\ y \implies Q\ x\ y) \implies bst-wrt\ P\ t \implies bst-wrt\ Q\ t$

⟨proof⟩

**lemma** *bst-wrt-le-if-bst*:  $bst\ t \implies bst-wrt\ (\le)\ t$

⟨proof⟩

**lemma** *bst-wrt-le-iff-sorted*:  $bst-wrt\ (\le)\ t \longleftrightarrow sorted\ (inorder\ t)$

⟨proof⟩

**lemma** *bst-iff-sorted-wrt-less*:  $bst\ t \longleftrightarrow sorted-wrt\ (<) (inorder\ t)$

⟨proof⟩

**103.12** *heap*

**103.13** *mirror*

**lemma** *mirror-Leaf[simp]*:  $\text{mirror } t = \langle \rangle \longleftrightarrow t = \langle \rangle$   
*<proof>*

**lemma** *Leaf-mirror[simp]*:  $\langle \rangle = \text{mirror } t \longleftrightarrow t = \langle \rangle$   
*<proof>*

**lemma** *size-mirror[simp]*:  $\text{size}(\text{mirror } t) = \text{size } t$   
*<proof>*

**lemma** *size1-mirror[simp]*:  $\text{size1}(\text{mirror } t) = \text{size1 } t$   
*<proof>*

**lemma** *height-mirror[simp]*:  $\text{height}(\text{mirror } t) = \text{height } t$   
*<proof>*

**lemma** *min-height-mirror [simp]*:  $\text{min-height } (\text{mirror } t) = \text{min-height } t$   
*<proof>*

**lemma** *ipl-mirror [simp]*:  $\text{ipl } (\text{mirror } t) = \text{ipl } t$   
*<proof>*

**lemma** *inorder-mirror*:  $\text{inorder}(\text{mirror } t) = \text{rev}(\text{inorder } t)$   
*<proof>*

**lemma** *map-mirror*:  $\text{map-tree } f (\text{mirror } t) = \text{mirror } (\text{map-tree } f t)$   
*<proof>*

**lemma** *mirror-mirror[simp]*:  $\text{mirror}(\text{mirror } t) = t$   
*<proof>*

**end**

## 104 Multiset of Elements of Binary Tree

**theory** *Tree-Multiset*  
**imports** *Multiset Tree*  
**begin**

Kept separate from theory *HOL-Library.Tree* to avoid importing all of theory *HOL-Library.Multiset* into *HOL-Library.Tree*. Should be merged if *HOL-Library.Multiset* ever becomes part of *Main*.

**fun** *mset-tree* ::  $'a \text{ tree} \Rightarrow 'a \text{ multiset}$  **where**  
*mset-tree Leaf* =  $\{\#\}$  |  
*mset-tree (Node l a r)* =  $\{\#a\#\} + \text{mset-tree } l + \text{mset-tree } r$



```

fun subtrees-mset :: 'a tree  $\Rightarrow$  'a tree multiset where
  subtrees-mset Leaf = {#Leaf#} |
  subtrees-mset (Node l x r) = add-mset (Node l x r) (subtrees-mset l + subtrees-mset
  r)

```

```

lemma mset-tree-empty-iff[simp]: mset-tree t = {#}  $\longleftrightarrow$  t = Leaf
<proof>

```

```

lemma set-mset-tree[simp]: set-mset (mset-tree t) = set-tree t
<proof>

```

```

lemma size-mset-tree[simp]: size(mset-tree t) = size t
<proof>

```

```

lemma mset-map-tree: mset-tree (map-tree f t) = image-mset f (mset-tree t)
<proof>

```

```

lemma mset-iff-set-tree:  $x \in \#$  mset-tree t  $\longleftrightarrow$   $x \in$  set-tree t
<proof>

```

```

lemma mset-preorder[simp]: mset (preorder t) = mset-tree t
<proof>

```

```

lemma mset-inorder[simp]: mset (inorder t) = mset-tree t
<proof>

```

```

lemma map-mirror: mset-tree (mirror t) = mset-tree t
<proof>

```

```

lemma in-subtrees-mset-iff[simp]:  $s \in \#$  subtrees-mset t  $\longleftrightarrow$   $s \in$  subtrees t
<proof>

```

**end**

**theory** Tree-Real

**imports**

Complex-Main

Tree

**begin**

This theory is separate from *HOL-Library.Tree* because the former is discrete and builds on *Main* whereas this theory builds on *Complex-Main*.

```

lemma size1-height-log:  $\log 2$  (size1 t)  $\leq$  height t
<proof>

```

```

lemma min-height-size1-log: min-height t  $\leq$   $\log 2$  (size1 t)
<proof>

```

**lemma** *size1-log-if-complete*:  $\text{complete } t \implies \text{height } t = \log 2 (\text{size1 } t)$   
 ⟨proof⟩

**lemma** *min-height-size1-log-if-incomplete*:  
 $\neg \text{complete } t \implies \text{min-height } t < \log 2 (\text{size1 } t)$   
 ⟨proof⟩

**lemma** *min-height-acomplete*: **assumes** *acomplete*  $t$   
**shows**  $\text{min-height } t = \text{nat}(\text{floor}(\log 2 (\text{size1 } t)))$   
 ⟨proof⟩

**lemma** *height-acomplete*: **assumes** *acomplete*  $t$   
**shows**  $\text{height } t = \text{nat}(\text{ceiling}(\log 2 (\text{size1 } t)))$   
 ⟨proof⟩

**lemma** *acomplete-Node-if-wbal1*:  
**assumes** *acomplete*  $l$  *acomplete*  $r$   $\text{size } l = \text{size } r + 1$   
**shows** *acomplete*  $\langle l, x, r \rangle$   
 ⟨proof⟩

**lemma** *acomplete-sym*: *acomplete*  $\langle l, x, r \rangle \implies \text{acomplete } \langle r, y, l \rangle$   
 ⟨proof⟩

**lemma** *acomplete-Node-if-wbal2*:  
**assumes** *acomplete*  $l$  *acomplete*  $r$   $\text{abs}(\text{int}(\text{size } l) - \text{int}(\text{size } r)) \leq 1$   
**shows** *acomplete*  $\langle l, x, r \rangle$   
 ⟨proof⟩

**lemma** *acomplete-if-wbalanced*: *wbalanced*  $t \implies \text{acomplete } t$   
 ⟨proof⟩

end

## 105 Unordered pairs

**theory** *Uprod* **imports** *Main* **begin**

**typedef**  $\langle 'a, 'b \rangle \text{ commute} = \{f :: 'a \Rightarrow 'a \Rightarrow 'b. \forall x y. f x y = f y x\}$   
**morphisms** *apply-commute* *Abs-commute*  
 ⟨proof⟩

**setup-lifting** *type-definition-commute*

**lemma** *apply-commute-commute*: *apply-commute*  $f x y = \text{apply-commute } f y x$   
 ⟨proof⟩

**context** **includes** *lifting-syntax* **begin**

**lift-definition** *rel-commute* :: ('a ⇒ 'b ⇒ bool) ⇒ ('c ⇒ 'd ⇒ bool) ⇒ ('a, 'c) commute ⇒ ('b, 'd) commute ⇒ bool  
**is** λA B. A ==> A ==> B ⟨proof⟩

**end**

**definition** *eq-upair* :: ('a × 'a) ⇒ ('a × 'a) ⇒ bool  
**where** *eq-upair* = (λ(a, b) (c, d). a = c ∧ b = d ∨ a = d ∧ b = c)

**lemma** *eq-upair-simps* [simp]:  
*eq-upair* (a, b) (c, d) ⟷ a = c ∧ b = d ∨ a = d ∧ b = c  
 ⟨proof⟩

**lemma** *equivp-eq-upair*: *equivp eq-upair*  
 ⟨proof⟩

**quotient-type** 'a *uprod* = 'a × 'a / *eq-upair* ⟨proof⟩

**lift-definition** *Upair* :: 'a ⇒ 'a ⇒ 'a *uprod* **is** *Pair* **parametric** *Pair-transfer*[of A A for A] ⟨proof⟩

**lemma** *uprod-exhaust* [case-names *Upair*, cases type: *uprod*]:  
**obtains** a b **where** x = *Upair* a b  
 ⟨proof⟩

**lemma** *Upair-inject* [simp]: *Upair* a b = *Upair* c d ⟷ a = c ∧ b = d ∨ a = d ∧ b = c  
 ⟨proof⟩

**code-datatype** *Upair*

**lift-definition** *case-uprod* :: ('a, 'b) commute ⇒ 'a *uprod* ⇒ 'b **is** *case-prod* **parametric** *case-prod-transfer*[of A A for A] ⟨proof⟩

**lemma** *case-uprod-simps* [simp, code]: *case-uprod* f (*Upair* x y) = *apply-commute* f x y  
 ⟨proof⟩

**lemma** *uprod-split*: P (*case-uprod* f x) ⟷ (∀ a b. x = *Upair* a b ⟶ P (*apply-commute* f a b))  
 ⟨proof⟩

**lemma** *uprod-split-asm*: P (*case-uprod* f x) ⟷ ¬ (∃ a b. x = *Upair* a b ∧ ¬ P (*apply-commute* f a b))  
 ⟨proof⟩

**lift-definition** *not-equal* :: ('a, bool) commute **is** (≠) ⟨proof⟩

**lemma** *apply-not-equal* [*simp*]: *apply-commute not-equal*  $x\ y \longleftrightarrow x \neq y$   
 ⟨*proof*⟩

**definition** *proper-uprod* :: 'a uprod  $\Rightarrow$  bool  
**where** *proper-uprod* = *case-uprod not-equal*

**lemma** *proper-uprod-simps* [*simp*, *code*]: *proper-uprod* (*Upair*  $x\ y$ )  $\longleftrightarrow x \neq y$   
 ⟨*proof*⟩

**context includes** *lifting-syntax* **begin**

**private lemma** *set-uprod-parametric'*:  
 (*rel-prod*  $A\ A \implies \text{rel-set } A$ ) ( $\lambda(a, b). \{a, b\}$ ) ( $\lambda(a, b). \{a, b\}$ )  
 ⟨*proof*⟩

**lift-definition** *set-uprod* :: 'a uprod  $\Rightarrow$  'a set **is**  $\lambda(a, b). \{a, b\}$   
**parametric** *set-uprod-parametric'* ⟨*proof*⟩

**lemma** *set-uprod-simps* [*simp*, *code*]: *set-uprod* (*Upair*  $x\ y$ ) =  $\{x, y\}$   
 ⟨*proof*⟩

**lemma** *finite-set-uprod* [*simp*]: *finite* (*set-uprod*  $x$ )  
 ⟨*proof*⟩ **lemma** *map-uprod-parametric'*:  
 (( $A \implies B \implies \text{rel-prod } A\ A \implies \text{rel-prod } B\ B$ ) ( $\lambda f. \text{map-prod } f\ f$ ) ( $\lambda f. \text{map-prod } f\ f$ )  
 ⟨*proof*⟩

**lift-definition** *map-uprod* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a uprod  $\Rightarrow$  'b uprod **is**  $\lambda f. \text{map-prod } f\ f$   
**parametric** *map-uprod-parametric'* ⟨*proof*⟩

**lemma** *map-uprod-simps* [*simp*, *code*]: *map-uprod*  $f$  (*Upair*  $x\ y$ ) = *Upair* ( $f\ x$ ) ( $f\ y$ )  
 ⟨*proof*⟩ **lemma** *rel-uprod-transfer'*:  
 (( $A \implies B \implies (=) \implies \text{rel-prod } A\ A \implies \text{rel-prod } B\ B \implies (=)$ )  
 ( $\lambda R (a, b) (c, d). R\ a\ c \wedge R\ b\ d \vee R\ a\ d \wedge R\ b\ c$ ) ( $\lambda R (a, b) (c, d). R\ a\ c \wedge R\ b\ d \vee R\ a\ d \wedge R\ b\ c$ )  
 ⟨*proof*⟩

**lift-definition** *rel-uprod* :: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool)  $\Rightarrow$  'a uprod  $\Rightarrow$  'b uprod  $\Rightarrow$  bool  
**is**  $\lambda R (a, b) (c, d). R\ a\ c \wedge R\ b\ d \vee R\ a\ d \wedge R\ b\ c$  **parametric** *rel-uprod-transfer'*  
 ⟨*proof*⟩

**lemma** *rel-uprod-simps* [*simp*, *code*]:  
*rel-uprod*  $R$  (*Upair*  $a\ b$ ) (*Upair*  $c\ d$ )  $\longleftrightarrow R\ a\ c \wedge R\ b\ d \vee R\ a\ d \wedge R\ b\ c$   
 ⟨*proof*⟩

**lemma** *Upair-parametric* [*transfer-rule*]: ( $A \implies A \implies \text{rel-uprod } A$ ) *Upair*  
*Upair*

⟨proof⟩

**lemma** *case-uprod-parametric* [transfer-rule]:

(*rel-commute*  $A B \implies \text{rel-uprod } A \implies B$ ) *case-uprod case-uprod*  
 ⟨proof⟩

**end**

**bnf** *uprod*: 'a uprod

map: map-uprod

sets: set-uprod

bd: natLeq

rel: rel-uprod

⟨proof⟩

**lemma** *pred-uprod-code* [simp, code]: *pred-uprod*  $P (U\text{pair } x y) \iff P x \wedge P y$

⟨proof⟩

**instantiation** *uprod* :: (equal) equal **begin**

**definition** *equal-uprod* :: 'a uprod  $\Rightarrow$  'a uprod  $\Rightarrow$  bool

**where** *equal-uprod* = (=)

**lemma** *equal-uprod-code* [code]:

*HOL.equal* ( $U\text{pair } x y$ ) ( $U\text{pair } z u$ )  $\iff x = z \wedge y = u \vee x = u \wedge y = z$   
 ⟨proof⟩

**instance** ⟨proof⟩

**end**

**quickcheck-generator** *uprod constructors*: *Upair*

**lemma** *UNIV-uprod*:  $UNIV = (\lambda x. U\text{pair } x x) \text{ ' } UNIV \cup (\lambda(x, y). U\text{pair } x y) \text{ ' } \Sigma UNIV (\lambda x. UNIV - \{x\})$

⟨proof⟩

**context** **begin**

**private lift-definition** *upair-inv* :: 'a uprod  $\Rightarrow$  'a

**is**  $\lambda(x, y). \text{if } x = y \text{ then } x \text{ else undefined}$  ⟨proof⟩

**lemma** *finite-UNIV-prod* [simp]:

*finite* ( $UNIV :: 'a \text{ uprod set}$ )  $\iff$  *finite* ( $UNIV :: 'a \text{ set}$ ) (**is** ?lhs = ?rhs)  
 ⟨proof⟩

**end**

**lemma** *card-UNIV-uprod*:

$\text{card } (UNIV :: 'a \text{ uprod set}) = \text{card } (UNIV :: 'a \text{ set}) * (\text{card } (UNIV :: 'a \text{ set}) + 1) \text{ div } 2$

```
(is ?UPROD = ?A * - div -)
⟨proof⟩
```

```
end
```

## 106 A type of finite bit strings

```
theory Word
imports
  HOL-Library.Type-Length
begin
```

### 106.1 Preliminaries

```
lemma signed-take-bit-decr-length-iff:
  ⟨signed-take-bit (LENGTH('a::len) - Suc 0) k = signed-take-bit (LENGTH('a)
- Suc 0) l
  ⟷ take-bit LENGTH('a) k = take-bit LENGTH('a) l⟩
⟨proof⟩
```

### 106.2 Fundamentals

#### 106.2.1 Type definition

```
quotient-type (overloaded) 'a word = int / ⟨λk l. take-bit LENGTH('a) k =
take-bit LENGTH('a::len) l⟩
morphisms rep Word ⟨proof⟩
```

**hide-const (open)** *rep* — only for foundational purpose

**hide-const (open)** *Word* — only for code generation

#### 106.2.2 Basic arithmetic

```
instantiation word :: (len) comm-ring-1
begin
```

```
lift-definition zero-word :: ⟨'a word⟩
is 0 ⟨proof⟩
```

```
lift-definition one-word :: ⟨'a word⟩
is 1 ⟨proof⟩
```

```
lift-definition plus-word :: ⟨'a word ⇒ 'a word ⇒ 'a word⟩
is ⟨(+)⟩
⟨proof⟩
```

```
lift-definition minus-word :: ⟨'a word ⇒ 'a word ⇒ 'a word⟩
is ⟨(-)⟩
⟨proof⟩
```

```

lift-definition uminus-word :: ‹'a word ⇒ 'a word›
  is uminus
  ‹proof›

lift-definition times-word :: ‹'a word ⇒ 'a word ⇒ 'a word›
  is ‹(*)›
  ‹proof›

instance
  ‹proof›

end

context
  includes lifting-syntax
  notes
    power-transfer [transfer-rule]
    transfer-rule-of-bool [transfer-rule]
    transfer-rule-numeral [transfer-rule]
    transfer-rule-of-nat [transfer-rule]
    transfer-rule-of-int [transfer-rule]
begin

lemma power-transfer-word [transfer-rule]:
  ‹(pcr-word ==> (=) ==> pcr-word) (∧) (∧)›
  ‹proof›

lemma [transfer-rule]:
  ‹((=) ==> pcr-word) of-bool of-bool›
  ‹proof›

lemma [transfer-rule]:
  ‹((=) ==> pcr-word) numeral numeral›
  ‹proof›

lemma [transfer-rule]:
  ‹((=) ==> pcr-word) int of-nat›
  ‹proof›

lemma [transfer-rule]:
  ‹((=) ==> pcr-word) (λk. k) of-int›
  ‹proof›

lemma [transfer-rule]:
  ‹(pcr-word ==> (⟷)) even ((dvd) 2 :: 'a::len word ⇒ bool)›
  ‹proof›

end

```

**lemma** *exp-eq-zero-iff* [*simp*]:  
 $\langle 2 \wedge n = (0 :: 'a::len\ word) \longleftrightarrow n \geq LENGTH('a) \rangle$   
 $\langle proof \rangle$

**lemma** *word-exp-length-eq-0* [*simp*]:  
 $\langle (2 :: 'a::len\ word) \wedge LENGTH('a) = 0 \rangle$   
 $\langle proof \rangle$

### 106.2.3 Basic tool setup

$\langle ML \rangle$

### 106.2.4 Basic code generation setup

**context**

**begin**

**qualified lift-definition** *the-int* ::  $\langle 'a::len\ word \Rightarrow int \rangle$   
**is**  $\langle take-bit\ LENGTH('a) \rangle$   $\langle proof \rangle$

**end**

**lemma** [*code abstype*]:  
 $\langle Word.Word\ (Word.the-int\ w) = w \rangle$   
 $\langle proof \rangle$

**lemma** *Word-eq-word-of-int* [*code-post, simp*]:  
 $\langle Word.Word = of-int \rangle$   
 $\langle proof \rangle$

**quickcheck-generator** *word*  
*constructors*:  
 $\langle 0 :: 'a::len\ word \rangle,$   
 $\langle numeral :: num \Rightarrow 'a::len\ word \rangle$

**instantiation** *word* ::  $(len)\ equal$   
**begin**

**lift-definition** *equal-word* ::  $\langle 'a\ word \Rightarrow 'a\ word \Rightarrow bool \rangle$   
**is**  $\langle \lambda k\ l.\ take-bit\ LENGTH('a)\ k = take-bit\ LENGTH('a)\ l \rangle$   
 $\langle proof \rangle$

**instance**  
 $\langle proof \rangle$

**end**

**lemma** [*code*]:  
 $\langle HOL.equal\ v\ w \longleftrightarrow HOL.equal\ (Word.the-int\ v)\ (Word.the-int\ w) \rangle$   
 $\langle proof \rangle$



**lemma** [code]:

⟨Word.the-int 0 = 0⟩  
⟨proof⟩

**lemma** [code]:

⟨Word.the-int 1 = 1⟩  
⟨proof⟩

**lemma** [code]:

⟨Word.the-int (v + w) = take-bit LENGTH('a) (Word.the-int v + Word.the-int w)⟩  
**for** v w :: ⟨'a::len word⟩  
⟨proof⟩

**lemma** [code]:

⟨Word.the-int (− w) = (let k = Word.the-int w in if w = 0 then 0 else 2 ^ LENGTH('a) − k)⟩  
**for** w :: ⟨'a::len word⟩  
⟨proof⟩

**lemma** [code]:

⟨Word.the-int (v − w) = take-bit LENGTH('a) (Word.the-int v − Word.the-int w)⟩  
**for** v w :: ⟨'a::len word⟩  
⟨proof⟩

**lemma** [code]:

⟨Word.the-int (v \* w) = take-bit LENGTH('a) (Word.the-int v \* Word.the-int w)⟩  
**for** v w :: ⟨'a::len word⟩  
⟨proof⟩

### 106.2.5 Basic conversions

**abbreviation** word-of-nat :: ⟨nat ⇒ 'a::len word⟩

**where** ⟨word-of-nat ≡ of-nat⟩

**abbreviation** word-of-int :: ⟨int ⇒ 'a::len word⟩

**where** ⟨word-of-int ≡ of-int⟩

**lemma** word-of-nat-eq-iff:

⟨word-of-nat m = (word-of-nat n :: 'a::len word) ⟷ take-bit LENGTH('a) m = take-bit LENGTH('a) n⟩  
⟨proof⟩

**lemma** word-of-int-eq-iff:

⟨word-of-int k = (word-of-int l :: 'a::len word) ⟷ take-bit LENGTH('a) k = take-bit LENGTH('a) l⟩

⟨proof⟩

**lemma** *word-of-nat-eq-0-iff*:

⟨word-of-nat  $n = (0 :: 'a::len\ word) \longleftrightarrow 2 \wedge LENGTH('a)\ dvd\ n$ ⟩  
 ⟨proof⟩

**lemma** *word-of-int-eq-0-iff*:

⟨word-of-int  $k = (0 :: 'a::len\ word) \longleftrightarrow 2 \wedge LENGTH('a)\ dvd\ k$ ⟩  
 ⟨proof⟩

**context** *semiring-1*

**begin**

**lift-definition** *unsigned* ::  $'b::len\ word \Rightarrow 'a$

**is** ⟨of-nat  $\circ$  nat  $\circ$  take-bit  $LENGTH('b)$ ⟩  
 ⟨proof⟩

**lemma** *unsigned-0* [*simp*]:

⟨unsigned  $0 = 0$ ⟩  
 ⟨proof⟩

**lemma** *unsigned-1* [*simp*]:

⟨unsigned  $1 = 1$ ⟩  
 ⟨proof⟩

**lemma** *unsigned-numeral* [*simp*]:

⟨unsigned (numeral  $n :: 'b::len\ word$ ) = of-nat (take-bit  $LENGTH('b)$  (numeral  $n$ ))⟩  
 ⟨proof⟩

**lemma** *unsigned-neg-numeral* [*simp*]:

⟨unsigned ( $-$  numeral  $n :: 'b::len\ word$ ) = of-nat (nat (take-bit  $LENGTH('b)$  ( $-$  numeral  $n$ )))⟩  
 ⟨proof⟩

**end**

**context** *semiring-1*

**begin**

**lemma** *unsigned-of-nat*:

⟨unsigned (word-of-nat  $n :: 'b::len\ word$ ) = of-nat (take-bit  $LENGTH('b)$   $n$ )⟩  
 ⟨proof⟩

**lemma** *unsigned-of-int*:

⟨unsigned (word-of-int  $k :: 'b::len\ word$ ) = of-nat (nat (take-bit  $LENGTH('b)$   $k$ ))⟩  
 ⟨proof⟩

**end**

**context** *semiring-char-0*

**begin**

**lemma** *unsigned-word-eqI*:

$\langle v = w \rangle$  **if**  $\langle \text{unsigned } v = \text{unsigned } w \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-eq-iff-unsigned*:

$\langle v = w \leftrightarrow \text{unsigned } v = \text{unsigned } w \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *inj-unsigned* [*simp*]:

$\langle \text{inj unsigned} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *unsigned-eq-0-iff*:

$\langle \text{unsigned } w = 0 \leftrightarrow w = 0 \rangle$   
 $\langle \text{proof} \rangle$

**end**

**context** *ring-1*

**begin**

**lift-definition** *signed* ::  $\langle 'b::\text{len word} \Rightarrow 'a \rangle$

**is**  $\langle \text{of-int} \circ \text{signed-take-bit } (\text{LENGTH } 'b) - \text{Suc } 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-0* [*simp*]:

$\langle \text{signed } 0 = 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-1* [*simp*]:

$\langle \text{signed } (1 :: 'b::\text{len word}) = (\text{if } \text{LENGTH } 'b = 1 \text{ then } - 1 \text{ else } 1) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-minus-1* [*simp*]:

$\langle \text{signed } (- 1 :: 'b::\text{len word}) = - 1 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-numeral* [*simp*]:

$\langle \text{signed } (\text{numeral } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH } 'b) - 1)$   
 $(\text{numeral } n) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-neg-numeral* [*simp*]:

$\langle \text{signed } (- \text{numeral } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH } 'b) -$   
 $1) (- \text{numeral } n) \rangle$

⟨proof⟩

**lemma** *signed-of-nat*:

⟨signed (word-of-nat  $n :: 'b::\text{len word}$ ) = of-int (signed-take-bit (LENGTH('b) – Suc 0) (int n))⟩

⟨proof⟩

**lemma** *signed-of-int*:

⟨signed (word-of-int  $n :: 'b::\text{len word}$ ) = of-int (signed-take-bit (LENGTH('b) – Suc 0) n)⟩

⟨proof⟩

**end**

**context** *ring-char-0*

**begin**

**lemma** *signed-word-eqI*:

⟨ $v = w$ ⟩ **if** ⟨signed  $v = \text{signed } w$ ⟩

⟨proof⟩

**lemma** *word-eq-iff-signed*:

⟨ $v = w \longleftrightarrow \text{signed } v = \text{signed } w$ ⟩

⟨proof⟩

**lemma** *inj-signed* [*simp*]:

⟨inj signed⟩

⟨proof⟩

**lemma** *signed-eq-0-iff*:

⟨signed  $w = 0 \longleftrightarrow w = 0$ ⟩

⟨proof⟩

**end**

**abbreviation** *unat* :: ⟨ $'a::\text{len word} \Rightarrow \text{nat}$ ⟩

**where** ⟨ $\text{unat} \equiv \text{unsigned}$ ⟩

**abbreviation** *uint* :: ⟨ $'a::\text{len word} \Rightarrow \text{int}$ ⟩

**where** ⟨ $\text{uint} \equiv \text{unsigned}$ ⟩

**abbreviation** *sint* :: ⟨ $'a::\text{len word} \Rightarrow \text{int}$ ⟩

**where** ⟨ $\text{sint} \equiv \text{signed}$ ⟩

**abbreviation** *ucast* :: ⟨ $'a::\text{len word} \Rightarrow 'b::\text{len word}$ ⟩

**where** ⟨ $\text{ucast} \equiv \text{unsigned}$ ⟩

**abbreviation** *scast* :: ⟨ $'a::\text{len word} \Rightarrow 'b::\text{len word}$ ⟩

**where** ⟨ $\text{scast} \equiv \text{signed}$ ⟩

**context**

**includes** *lifting-syntax*

**begin**

**lemma** [*transfer-rule*]:

⟨(*pcr-word* ==> (=)) (*nat* ∘ *take-bit* *LENGTH*('a)) (*unat* :: 'a::len word ⇒ *nat*)⟩  
 ⟨*proof*⟩

**lemma** [*transfer-rule*]:

⟨(*pcr-word* ==> (=)) (*take-bit* *LENGTH*('a)) (*uint* :: 'a::len word ⇒ *int*)⟩  
 ⟨*proof*⟩

**lemma** [*transfer-rule*]:

⟨(*pcr-word* ==> (=)) (*signed-take-bit* (*LENGTH*('a) − *Suc* 0)) (*sint* :: 'a::len word ⇒ *int*)⟩  
 ⟨*proof*⟩

**lemma** [*transfer-rule*]:

⟨(*pcr-word* ==> *pcr-word*) (*take-bit* *LENGTH*('a)) (*ucast* :: 'a::len word ⇒ 'b::len word)⟩  
 ⟨*proof*⟩

**lemma** [*transfer-rule*]:

⟨(*pcr-word* ==> *pcr-word*) (*signed-take-bit* (*LENGTH*('a) − *Suc* 0)) (*scast* :: 'a::len word ⇒ 'b::len word)⟩  
 ⟨*proof*⟩

**end**

**lemma** *of-nat-unat* [*simp*]:

⟨*of-nat* (*unat* w) = *unsigned* w⟩  
 ⟨*proof*⟩

**lemma** *of-int-uint* [*simp*]:

⟨*of-int* (*uint* w) = *unsigned* w⟩  
 ⟨*proof*⟩

**lemma** *of-int-sint* [*simp*]:

⟨*of-int* (*sint* a) = *signed* a⟩  
 ⟨*proof*⟩

**lemma** *nat-uint-eq* [*simp*]:

⟨*nat* (*uint* w) = *unat* w⟩  
 ⟨*proof*⟩

**lemma** *nat-of-natural-unsigned-eq* [*simp*]:

⟨*nat-of-natural* (*unsigned* w) = *unat* w⟩

⟨proof⟩

**lemma** *int-of-integer-unsigned-eq* [simp]:  
 ⟨int-of-integer (unsigned w) = uint w⟩  
 ⟨proof⟩

**lemma** *int-of-integer-signed-eq* [simp]:  
 ⟨int-of-integer (signed w) = sint w⟩  
 ⟨proof⟩

**lemma** *sgn-uint-eq* [simp]:  
 ⟨sgn (uint w) = of-bool (w ≠ 0)⟩  
 ⟨proof⟩

Aliases only for code generation

**context**

**begin**

**qualified lift-definition** *of-int* :: ⟨int ⇒ 'a::len word⟩  
**is** ⟨take-bit LENGTH('a)⟩ ⟨proof⟩ **lift-definition** *of-nat* :: ⟨nat ⇒ 'a::len word⟩  
**is** ⟨int ∘ take-bit LENGTH('a)⟩ ⟨proof⟩ **lift-definition** *the-nat* :: ⟨'a::len word  
 ⇒ nat⟩  
**is** ⟨nat ∘ take-bit LENGTH('a)⟩ ⟨proof⟩ **lift-definition** *the-signed-int* :: ⟨'a::len  
 word ⇒ int⟩  
**is** ⟨signed-take-bit (LENGTH('a) - Suc 0)⟩ ⟨proof⟩ **lift-definition** *cast* :: ⟨'a::len  
 word ⇒ 'b::len word⟩  
**is** ⟨take-bit LENGTH('a)⟩ ⟨proof⟩ **lift-definition** *signed-cast* :: ⟨'a::len word ⇒  
 'b::len word⟩  
**is** ⟨signed-take-bit (LENGTH('a) - Suc 0)⟩ ⟨proof⟩

**end**

**lemma** [code-abbrev, simp]:  
 ⟨Word.the-int = uint⟩  
 ⟨proof⟩

**lemma** [code]:  
 ⟨Word.the-int (Word.of-int k :: 'a::len word) = take-bit LENGTH('a) k⟩  
 ⟨proof⟩

**lemma** [code-abbrev, simp]:  
 ⟨Word.of-int = word-of-int⟩  
 ⟨proof⟩

**lemma** [code]:  
 ⟨Word.the-int (Word.of-nat n :: 'a::len word) = take-bit LENGTH('a) (int n)⟩  
 ⟨proof⟩

**lemma** [code-abbrev, simp]:

⟨ *Word.of-nat = word-of-nat* ⟩  
 ⟨ *proof* ⟩

**lemma** [*code*]:  
 ⟨ *Word.the-nat w = nat (Word.the-int w)* ⟩  
 ⟨ *proof* ⟩

**lemma** [*code-abbrev, simp*]:  
 ⟨ *Word.the-nat = unat* ⟩  
 ⟨ *proof* ⟩

**lemma** [*code*]:  
 ⟨ *Word.the-signed-int w = signed-take-bit (LENGTH('a) - Suc 0) (Word.the-int w)* ⟩  
**for** *w :: 'a::len word*  
 ⟨ *proof* ⟩

**lemma** [*code-abbrev, simp*]:  
 ⟨ *Word.the-signed-int = sint* ⟩  
 ⟨ *proof* ⟩

**lemma** [*code*]:  
 ⟨ *Word.the-int (Word.cast w :: 'b::len word) = take-bit LENGTH('b) (Word.the-int w)* ⟩  
**for** *w :: 'a::len word*  
 ⟨ *proof* ⟩

**lemma** [*code-abbrev, simp*]:  
 ⟨ *Word.cast = ucast* ⟩  
 ⟨ *proof* ⟩

**lemma** [*code*]:  
 ⟨ *Word.the-int (Word.signed-cast w :: 'b::len word) = take-bit LENGTH('b) (Word.the-signed-int w)* ⟩  
**for** *w :: 'a::len word*  
 ⟨ *proof* ⟩

**lemma** [*code-abbrev, simp*]:  
 ⟨ *Word.signed-cast = scast* ⟩  
 ⟨ *proof* ⟩

**lemma** [*code*]:  
 ⟨ *unsigned w = of-nat (nat (Word.the-int w))* ⟩  
 ⟨ *proof* ⟩

**lemma** [*code*]:  
 ⟨ *signed w = of-int (Word.the-signed-int w)* ⟩  
 ⟨ *proof* ⟩

**106.2.6 Basic ordering**

**instantiation** *word* :: (*len*) *linorder*  
**begin**

**lift-definition** *less-eq-word* :: '*a word* ⇒ '*a word* ⇒ *bool*  
**is**  $\lambda a b. \text{take-bit LENGTH('a) } a \leq \text{take-bit LENGTH('a) } b$   
 ⟨*proof*⟩

**lift-definition** *less-word* :: '*a word* ⇒ '*a word* ⇒ *bool*  
**is**  $\lambda a b. \text{take-bit LENGTH('a) } a < \text{take-bit LENGTH('a) } b$   
 ⟨*proof*⟩

**instance**  
 ⟨*proof*⟩

**end**

**interpretation** *word-order*: *ordering-top* ⟨( $\leq$ )⟩ ⟨( $<$ )⟩ ⟨ $- 1$  :: '*a::len word*⟩  
 ⟨*proof*⟩

**interpretation** *word-coorder*: *ordering-top* ⟨( $\geq$ )⟩ ⟨( $>$ )⟩ ⟨ $0$  :: '*a::len word*⟩  
 ⟨*proof*⟩

**lemma** *word-of-nat-less-eq-iff*:  
 ⟨*word-of-nat*  $m \leq (\text{word-of-nat } n :: \text{'a::len word}) \longleftrightarrow \text{take-bit LENGTH('a) } m$   
 $\leq \text{take-bit LENGTH('a) } n$ ⟩  
 ⟨*proof*⟩

**lemma** *word-of-int-less-eq-iff*:  
 ⟨*word-of-int*  $k \leq (\text{word-of-int } l :: \text{'a::len word}) \longleftrightarrow \text{take-bit LENGTH('a) } k \leq$   
 $\text{take-bit LENGTH('a) } l$ ⟩  
 ⟨*proof*⟩

**lemma** *word-of-nat-less-iff*:  
 ⟨*word-of-nat*  $m < (\text{word-of-nat } n :: \text{'a::len word}) \longleftrightarrow \text{take-bit LENGTH('a) } m$   
 $< \text{take-bit LENGTH('a) } n$ ⟩  
 ⟨*proof*⟩

**lemma** *word-of-int-less-iff*:  
 ⟨*word-of-int*  $k < (\text{word-of-int } l :: \text{'a::len word}) \longleftrightarrow \text{take-bit LENGTH('a) } k <$   
 $\text{take-bit LENGTH('a) } l$ ⟩  
 ⟨*proof*⟩

**lemma** *word-le-def* [*code*]:  
 $a \leq b \longleftrightarrow \text{uint } a \leq \text{uint } b$   
 ⟨*proof*⟩

**lemma** *word-less-def* [*code*]:  
 $a < b \longleftrightarrow \text{uint } a < \text{uint } b$



⟨proof⟩

**lemma** *word-greater-zero-iff*:

⟨ $a > 0 \iff a \neq 0$ ⟩ **for**  $a :: \langle 'a :: \text{len word} \rangle$

⟨proof⟩

**lemma** *of-nat-word-less-eq-iff*:

⟨ $\text{of-nat } m \leq (\text{of-nat } n :: 'a :: \text{len word}) \iff \text{take-bit LENGTH('a) } m \leq \text{take-bit LENGTH('a) } n$ ⟩

⟨proof⟩

**lemma** *of-nat-word-less-iff*:

⟨ $\text{of-nat } m < (\text{of-nat } n :: 'a :: \text{len word}) \iff \text{take-bit LENGTH('a) } m < \text{take-bit LENGTH('a) } n$ ⟩

⟨proof⟩

**lemma** *of-int-word-less-eq-iff*:

⟨ $\text{of-int } k \leq (\text{of-int } l :: 'a :: \text{len word}) \iff \text{take-bit LENGTH('a) } k \leq \text{take-bit LENGTH('a) } l$ ⟩

⟨proof⟩

**lemma** *of-int-word-less-iff*:

⟨ $\text{of-int } k < (\text{of-int } l :: 'a :: \text{len word}) \iff \text{take-bit LENGTH('a) } k < \text{take-bit LENGTH('a) } l$ ⟩

⟨proof⟩

### 106.3 Enumeration

**lemma** *inj-on-word-of-nat*:

⟨ $\text{inj-on } (\text{word-of-nat} :: \text{nat} \Rightarrow 'a :: \text{len word}) \{0..<2 \wedge \text{LENGTH('a)}\}$ ⟩

⟨proof⟩

**lemma** *UNIV-word-eq-word-of-nat*:

⟨ $(\text{UNIV} :: 'a :: \text{len word set}) = \text{word-of-nat } \{0..<2 \wedge \text{LENGTH('a)}\}$  (is  $\langle - = ?A \rangle$ )

⟨proof⟩

**instantiation** *word* :: (len) enum

**begin**

**definition** *enum-word* :: ⟨'a word list⟩

**where** ⟨ $\text{enum-word} = \text{map word-of-nat } [0..<2 \wedge \text{LENGTH('a)}]$ ⟩

**definition** *enum-all-word* :: ⟨('a word  $\Rightarrow$  bool)  $\Rightarrow$  bool⟩

**where** ⟨ $\text{enum-all-word} = \text{All}$ ⟩

**definition** *enum-ex-word* :: ⟨('a word  $\Rightarrow$  bool)  $\Rightarrow$  bool⟩

**where** ⟨ $\text{enum-ex-word} = \text{Ex}$ ⟩

**instance**  
 ⟨*proof*⟩

**end**

**lemma** [*code*]:  
 ⟨*Enum.enum-all*  $P \longleftrightarrow \text{list-all } P \text{ Enum.enum}$ ⟩  
 ⟨*Enum.enum-ex*  $P \longleftrightarrow \text{list-ex } P \text{ Enum.enum}$ ⟩ **for**  $P :: \langle 'a::\text{len word} \Rightarrow \text{bool} \rangle$   
 ⟨*proof*⟩

## 106.4 Bit-wise operations

The following specification of word division just lifts the pre-existing division on integers named “F-Division” in [2].

**instantiation**  $\text{word} :: (\text{len}) \text{ semiring-modulo}$   
**begin**

**lift-definition**  $\text{divide-word} :: \langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is**  $\langle \lambda a b. \text{take-bit } \text{LENGTH}('a) a \text{ div take-bit } \text{LENGTH}('a) b \rangle$   
 ⟨*proof*⟩

**lift-definition**  $\text{modulo-word} :: \langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is**  $\langle \lambda a b. \text{take-bit } \text{LENGTH}('a) a \text{ mod take-bit } \text{LENGTH}('a) b \rangle$   
 ⟨*proof*⟩

**instance** ⟨*proof*⟩

**end**

**lemma** *unat-div-distrib*:  
 ⟨*unat*  $(v \text{ div } w) = \text{unat } v \text{ div unat } w$ ⟩  
 ⟨*proof*⟩

**lemma** *unat-mod-distrib*:  
 ⟨*unat*  $(v \text{ mod } w) = \text{unat } v \text{ mod unat } w$ ⟩  
 ⟨*proof*⟩

**instance**  $\text{word} :: (\text{len}) \text{ semiring-parity}$   
 ⟨*proof*⟩

**lemma** *word-bit-induct* [*case-names zero even odd*]:  
 ⟨ $P a$ ⟩ **if** *word-zero*:  $\langle P 0 \rangle$   
**and** *word-even*:  $\langle \bigwedge a. P a \implies 0 < a \implies a < 2^{\wedge}(\text{LENGTH}('a) - \text{Suc } 0) \implies P (2 * a) \rangle$   
**and** *word-odd*:  $\langle \bigwedge a. P a \implies a < 2^{\wedge}(\text{LENGTH}('a) - \text{Suc } 0) \implies P (1 + 2 * a) \rangle$   
**for**  $P$  **and**  $a :: \langle 'a::\text{len word} \rangle$   
 ⟨*proof*⟩

**lemma** *bit-word-half-eq*:

⟨(of-bool b + a \* 2) div 2 = a⟩  
 if ⟨a < 2 ^ (LENGTH('a) - Suc 0)⟩  
 for a :: ⟨'a::len word⟩  
 ⟨proof⟩

**lemma** *even-mult-exp-div-word-iff*:

⟨even (a \* 2 ^ m div 2 ^ n) ⟷ ¬ (m ≤ n ∧ n < LENGTH('a) ∧ odd (a div 2 ^ (n - m)))⟩ for a :: ⟨'a::len word⟩  
 ⟨proof⟩

**instantiation** word :: (len) semiring-bits  
**begin**

**lift-definition** *bit-word* :: ⟨'a word ⇒ nat ⇒ bool⟩

is ⟨λk n. n < LENGTH('a) ∧ bit k n⟩  
 ⟨proof⟩

**instance** ⟨proof⟩

**end**

**lemma** *bit-word-eqI*:

⟨a = b⟩ if ⟨∧n. n < LENGTH('a) ⟹ bit a n ⟷ bit b n⟩  
 for a b :: ⟨'a::len word⟩  
 ⟨proof⟩

**lemma** *bit-imp-le-length*: ⟨n < LENGTH('a)⟩ if ⟨bit w n⟩ for w :: ⟨'a::len word⟩  
 ⟨proof⟩

**lemma** *not-bit-length* [simp]:

⟨¬ bit w LENGTH('a)⟩ for w :: ⟨'a::len word⟩  
 ⟨proof⟩

**lemma** *finite-bit-word* [simp]:

⟨finite {n. bit w n}⟩  
 for w :: ⟨'a::len word⟩  
 ⟨proof⟩

**lemma** *bit-numeral-word-iff* [simp]:

⟨bit (numeral w :: 'a::len word) n ⟷ n < LENGTH('a) ∧ bit (numeral w :: int) n⟩  
 ⟨proof⟩

**lemma** *bit-neg-numeral-word-iff* [simp]:

⟨bit (- numeral w :: 'a::len word) n ⟷ n < LENGTH('a) ∧ bit (- numeral w :: int) n⟩  
 ⟨proof⟩

**instantiation** *word* :: (*len*) *ring-bit-operations*  
**begin**

**lift-definition** *not-word* ::  $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is** *not*  
 $\langle \text{proof} \rangle$

**lift-definition** *and-word* ::  $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is**  $\langle \text{and} \rangle$   
 $\langle \text{proof} \rangle$

**lift-definition** *or-word* ::  $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is** *or*  
 $\langle \text{proof} \rangle$

**lift-definition** *xor-word* ::  $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is** *xor*  
 $\langle \text{proof} \rangle$

**lift-definition** *mask-word* ::  $\langle \text{nat} \Rightarrow 'a \text{ word} \rangle$   
**is** *mask*  
 $\langle \text{proof} \rangle$

**lift-definition** *set-bit-word* ::  $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is** *set-bit*  
 $\langle \text{proof} \rangle$

**lift-definition** *unset-bit-word* ::  $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is** *unset-bit*  
 $\langle \text{proof} \rangle$

**lift-definition** *flip-bit-word* ::  $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is** *flip-bit*  
 $\langle \text{proof} \rangle$

**lift-definition** *push-bit-word* ::  $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is** *push-bit*  
 $\langle \text{proof} \rangle$

**lift-definition** *drop-bit-word* ::  $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is**  $\langle \lambda n. \text{drop-bit } n \circ \text{take-bit } \text{LENGTH}('a) \rangle$   
 $\langle \text{proof} \rangle$

**lift-definition** *take-bit-word* ::  $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$   
**is**  $\langle \lambda n. \text{take-bit } (\text{min } \text{LENGTH}('a) \ n) \rangle$   
 $\langle \text{proof} \rangle$

**context**

**includes** *bit-operations-syntax*  
**begin**

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

**end**

**end**

**lemma** [*code*]:  
 $\langle \text{push-bit } n \ w = w * 2 \wedge n \ \mathbf{for} \ w :: \langle 'a::\text{len word} \rangle \rangle$   
 $\langle \text{proof} \rangle$

**lemma** [*code*]:  
 $\langle \text{Word.the-int } (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{Word.the-int } w) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** [*code*]:  
 $\langle \text{Word.the-int } (\text{take-bit } n \ w) = (\text{if } n < \text{LENGTH}('a::\text{len}) \ \text{then } \text{take-bit } n \ (\text{Word.the-int } w) \ \text{else } \text{Word.the-int } w) \rangle$   
 $\mathbf{for} \ w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** [*code-abbrev*]:  
 $\langle \text{push-bit } n \ 1 = (2 :: 'a::\text{len word}) \wedge n \rangle$   
 $\langle \text{proof} \rangle$

**context**  
**includes** *bit-operations-syntax*  
**begin**

**lemma** [*code*]:  
 $\langle \text{NOT } w = \text{Word.of-int } (\text{NOT } (\text{Word.the-int } w)) \rangle$   
 $\mathbf{for} \ w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** [*code*]:  
 $\langle \text{Word.the-int } (v \ \text{AND} \ w) = \text{Word.the-int } v \ \text{AND} \ \text{Word.the-int } w \rangle$   
 $\langle \text{proof} \rangle$

**lemma** [*code*]:  
 $\langle \text{Word.the-int } (v \ \text{OR} \ w) = \text{Word.the-int } v \ \text{OR} \ \text{Word.the-int } w \rangle$   
 $\langle \text{proof} \rangle$

**lemma** [*code*]:  
 $\langle \text{Word.the-int } (v \ \text{XOR} \ w) = \text{Word.the-int } v \ \text{XOR} \ \text{Word.the-int } w \rangle$   
 $\langle \text{proof} \rangle$

**lemma** [*code*]:

⟨*Word.the-int* (*mask*  $n :: 'a::\text{len word}$ ) = *mask* (*min LENGTH*( $'a$ )  $n$ )⟩  
 ⟨*proof*⟩

**lemma** [*code*]:  
 ⟨*set-bit*  $n w = w$  *OR* *push-bit*  $n 1$ ⟩ **for**  $w :: 'a::\text{len word}$ ⟩  
 ⟨*proof*⟩

**lemma** [*code*]:  
 ⟨*unset-bit*  $n w = w$  *AND NOT* (*push-bit*  $n 1$ )⟩ **for**  $w :: 'a::\text{len word}$ ⟩  
 ⟨*proof*⟩

**lemma** [*code*]:  
 ⟨*flip-bit*  $n w = w$  *XOR* *push-bit*  $n 1$ ⟩ **for**  $w :: 'a::\text{len word}$ ⟩  
 ⟨*proof*⟩

**context**  
**includes** *lifting-syntax*  
**begin**

**lemma** *set-bit-word-transfer* [*transfer-rule*]:  
 ⟨ $((=) \implies \text{pcr-word} \implies \text{pcr-word})$  *set-bit set-bit*⟩  
 ⟨*proof*⟩

**lemma** *unset-bit-word-transfer* [*transfer-rule*]:  
 ⟨ $((=) \implies \text{pcr-word} \implies \text{pcr-word})$  *unset-bit unset-bit*⟩  
 ⟨*proof*⟩

**lemma** *flip-bit-word-transfer* [*transfer-rule*]:  
 ⟨ $((=) \implies \text{pcr-word} \implies \text{pcr-word})$  *flip-bit flip-bit*⟩  
 ⟨*proof*⟩

**lemma** *signed-take-bit-word-transfer* [*transfer-rule*]:  
 ⟨ $((=) \implies \text{pcr-word} \implies \text{pcr-word})$   
 ( $\lambda n k. \text{signed-take-bit } n (\text{take-bit LENGTH}('a::\text{len}) k)$ )  
 (*signed-take-bit*  $:: \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word}$ )⟩  
 ⟨*proof*⟩

**end**

**end**

## 106.5 Conversions including casts

### 106.5.1 Generic unsigned conversion

**context** *semiring-bits*  
**begin**

**lemma** *bit-unsigned-iff* [*bit-simps*]:  
 ⟨*bit* (*unsigned*  $w$ )  $n \iff$  *possible-bit* *TYPE*( $'a$ )  $n \wedge$  *bit*  $w n$ ⟩

```

for  $w :: \langle 'b::len \text{ word} \rangle$ 
   $\langle \text{proof} \rangle$ 

end

lemma possible-bit-word[simp]:
   $\langle \text{possible-bit TYPE}(\langle 'a :: len \rangle \text{ word}) \ m \longleftrightarrow m < LENGTH('a) \rangle$ 
   $\langle \text{proof} \rangle$ 

context semiring-bit-operations
begin

lemma unsigned-minus-1-eq-mask:
   $\langle \text{unsigned} \ (- \ 1 :: 'b::len \text{ word}) = \text{mask } LENGTH('b) \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma unsigned-push-bit-eq:
   $\langle \text{unsigned} \ (\text{push-bit } n \ w) = \text{take-bit } LENGTH('b) \ (\text{push-bit } n \ (\text{unsigned } w)) \rangle$ 
  for  $w :: \langle 'b::len \text{ word} \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma unsigned-take-bit-eq:
   $\langle \text{unsigned} \ (\text{take-bit } n \ w) = \text{take-bit } n \ (\text{unsigned } w) \rangle$ 
  for  $w :: \langle 'b::len \text{ word} \rangle$ 
   $\langle \text{proof} \rangle$ 

end

context linordered-euclidean-semiring-bit-operations
begin

lemma unsigned-drop-bit-eq:
   $\langle \text{unsigned} \ (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{take-bit } LENGTH('b) \ (\text{unsigned } w)) \rangle$ 
  for  $w :: \langle 'b::len \text{ word} \rangle$ 
   $\langle \text{proof} \rangle$ 

end

lemma ucast-drop-bit-eq:
   $\langle \text{ucast} \ (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{ucast } w :: 'b::len \text{ word}) \rangle$ 
  if  $\langle LENGTH('a) \leq LENGTH('b) \rangle$  for  $w :: \langle 'a::len \text{ word} \rangle$ 
   $\langle \text{proof} \rangle$ 

context semiring-bit-operations
begin

context
  includes bit-operations-syntax
begin

```

```

lemma unsigned-and-eq:
  ⟨unsigned (v AND w) = unsigned v AND unsigned w⟩
  for v w :: ⟨'b::len word⟩
  ⟨proof⟩

lemma unsigned-or-eq:
  ⟨unsigned (v OR w) = unsigned v OR unsigned w⟩
  for v w :: ⟨'b::len word⟩
  ⟨proof⟩

lemma unsigned-xor-eq:
  ⟨unsigned (v XOR w) = unsigned v XOR unsigned w⟩
  for v w :: ⟨'b::len word⟩
  ⟨proof⟩

end

end

context ring-bit-operations
begin

context
  includes bit-operations-syntax
begin

lemma unsigned-not-eq:
  ⟨unsigned (NOT w) = take-bit LENGTH('b) (NOT (unsigned w))⟩
  for w :: ⟨'b::len word⟩
  ⟨proof⟩

end

end

context linordered-euclidean-semiring
begin

lemma unsigned-greater-eq [simp]:
  ⟨ $0 \leq \text{unsigned } w$ ⟩ for w :: ⟨'b::len word⟩
  ⟨proof⟩

lemma unsigned-less [simp]:
  ⟨unsigned w <  $2 \wedge \text{LENGTH}('b)$ ⟩ for w :: ⟨'b::len word⟩
  ⟨proof⟩

end

```



**context** *linordered-semidom*  
**begin**

**lemma** *word-less-eq-iff-unsigned*:  
 $a \leq b \longleftrightarrow \text{unsigned } a \leq \text{unsigned } b$   
 ⟨*proof*⟩

**lemma** *word-less-iff-unsigned*:  
 $a < b \longleftrightarrow \text{unsigned } a < \text{unsigned } b$   
 ⟨*proof*⟩

**end**

### 106.5.2 Generic signed conversion

**context** *ring-bit-operations*  
**begin**

**lemma** *bit-signed-iff* [*bit-simps*]:  
 ⟨*bit* (*signed* *w*) *n*  $\longleftrightarrow$  *possible-bit* *TYPE*('a) *n*  $\wedge$  *bit* *w* (*min* (*LENGTH*('b) – *Suc* 0) *n*)⟩  
**for** *w* :: ⟨'b::len word⟩  
 ⟨*proof*⟩

**lemma** *signed-push-bit-eq*:  
 ⟨*signed* (*push-bit* *n* *w*) = *signed-take-bit* (*LENGTH*('b) – *Suc* 0) (*push-bit* *n* (*signed* *w* :: 'a))⟩  
**for** *w* :: ⟨'b::len word⟩  
 ⟨*proof*⟩

**lemma** *signed-take-bit-eq*:  
 ⟨*signed* (*take-bit* *n* *w*) = (if *n* < *LENGTH*('b) then *take-bit* *n* (*signed* *w*) else *signed* *w*)⟩  
**for** *w* :: ⟨'b::len word⟩  
 ⟨*proof*⟩

**context**  
**includes** *bit-operations-syntax*  
**begin**

**lemma** *signed-not-eq*:  
 ⟨*signed* (*NOT* *w*) = *signed-take-bit* *LENGTH*('b) (*NOT* (*signed* *w*))⟩  
**for** *w* :: ⟨'b::len word⟩  
 ⟨*proof*⟩

**lemma** *signed-and-eq*:  
 ⟨*signed* (*v* *AND* *w*) = *signed* *v* *AND* *signed* *w*⟩  
**for** *v* *w* :: ⟨'b::len word⟩  
 ⟨*proof*⟩

**lemma** *signed-or-eq*:

⟨*signed* (*v OR w*) = *signed v OR signed w*⟩  
**for** *v w* :: ⟨'b::len word⟩  
 ⟨*proof*⟩

**lemma** *signed-xor-eq*:

⟨*signed* (*v XOR w*) = *signed v XOR signed w*⟩  
**for** *v w* :: ⟨'b::len word⟩  
 ⟨*proof*⟩

**end**

**end**

### 106.5.3 More

**lemma** *sint-greater-eq*:

⟨ $-(2 \wedge (\text{LENGTH}(a) - \text{Suc } 0)) \leq \text{sint } w$ ⟩ **for** *w* :: ⟨'a::len word⟩  
 ⟨*proof*⟩

**lemma** *sint-less*:

⟨*sint w* <  $2 \wedge (\text{LENGTH}(a) - \text{Suc } 0)$ ⟩ **for** *w* :: ⟨'a::len word⟩  
 ⟨*proof*⟩

**lemma** *uint-div-distrib*:

⟨*uint* (*v div w*) = *uint v div uint w*⟩  
 ⟨*proof*⟩

**lemma** *unat-drop-bit-eq*:

⟨*unat* (*drop-bit n w*) = *drop-bit n* (*unat w*)⟩  
 ⟨*proof*⟩

**lemma** *uint-mod-distrib*:

⟨*uint* (*v mod w*) = *uint v mod uint w*⟩  
 ⟨*proof*⟩

**context** *semiring-bit-operations*

**begin**

**lemma** *unsigned-ucast-eq*:

⟨*unsigned* (*ucast w* :: 'c::len word) = *take-bit LENGTH('c)* (*unsigned w*)⟩  
**for** *w* :: ⟨'b::len word⟩  
 ⟨*proof*⟩

**end**

**context** *ring-bit-operations*

**begin**

**lemma** *signed-ucast-eq*:

⟨*signed* (*ucast*  $w :: 'c::len$  word) = *signed-take-bit* (*LENGTH*('c) - *Suc* 0)  
 (*unsigned*  $w$ )⟩  
**for**  $w :: 'b::len$  word  
 ⟨*proof*⟩

**lemma** *signed-scast-eq*:

⟨*signed* (*scast*  $w :: 'c::len$  word) = *signed-take-bit* (*LENGTH*('c) - *Suc* 0) (*signed*  
 $w$ )⟩  
**for**  $w :: 'b::len$  word  
 ⟨*proof*⟩

**end**

**lemma** *uint-nonnegative*:  $0 \leq \text{uint } w$

⟨*proof*⟩

**lemma** *uint-bounded*:  $\text{uint } w < 2^{\wedge} \text{LENGTH}('a)$

**for**  $w :: 'a::len$  word  
 ⟨*proof*⟩

**lemma** *uint-idem*:  $\text{uint } w \bmod 2^{\wedge} \text{LENGTH}('a) = \text{uint } w$

**for**  $w :: 'a::len$  word  
 ⟨*proof*⟩

**lemma** *word-uint-eqI*:  $\text{uint } a = \text{uint } b \implies a = b$

⟨*proof*⟩

**lemma** *word-uint-eq-iff*:  $a = b \iff \text{uint } a = \text{uint } b$

⟨*proof*⟩

**lemma** *uint-word-of-int-eq*:

⟨*uint* (*word-of-int*  $k :: 'a::len$  word) = *take-bit* *LENGTH*('a)  $k$ ⟩  
 ⟨*proof*⟩

**lemma** *uint-word-of-int*:  $\text{uint} (\text{word-of-int } k :: 'a::len \text{ word}) = k \bmod 2^{\wedge} \text{LENGTH}('a)$

⟨*proof*⟩

**lemma** *word-of-int-uint*:  $\text{word-of-int} (\text{uint } w) = w$

⟨*proof*⟩

**lemma** *word-div-def* [*code*]:

$a \text{ div } b = \text{word-of-int} (\text{uint } a \text{ div } \text{uint } b)$   
 ⟨*proof*⟩

**lemma** *word-mod-def* [*code*]:

$a \text{ mod } b = \text{word-of-int} (\text{uint } a \text{ mod } \text{uint } b)$   
 ⟨*proof*⟩

**lemma** *split-word-all*:  $(\bigwedge x :: 'a :: \text{len word. PROP } P \ x) \equiv (\bigwedge x. \text{PROP } P \ (\text{word-of-int } x))$   
 ⟨proof⟩

**lemma** *sint-uint*:  
 ⟨*sint*  $w = \text{signed-take-bit } (\text{LENGTH } ('a) - \text{Suc } 0) \ (\text{uint } w)$ ⟩  
**for**  $w :: 'a :: \text{len word}$   
 ⟨proof⟩

**lemma** *unat-eq-nat-uint*:  
 ⟨*unat*  $w = \text{nat } (\text{uint } w)$ ⟩  
 ⟨proof⟩

**lemma** *ucast-eq*:  
 ⟨*ucast*  $w = \text{word-of-int } (\text{uint } w)$ ⟩  
 ⟨proof⟩

**lemma** *scast-eq*:  
 ⟨*scast*  $w = \text{word-of-int } (\text{sint } w)$ ⟩  
 ⟨proof⟩

**lemma** *uint-0-eq*:  
 ⟨*uint*  $0 = 0$ ⟩  
 ⟨proof⟩

**lemma** *uint-1-eq*:  
 ⟨*uint*  $1 = 1$ ⟩  
 ⟨proof⟩

**lemma** *word-m1-wi*:  $- 1 = \text{word-of-int } (- 1)$   
 ⟨proof⟩

**lemma** *uint-0-iff*:  $\text{uint } x = 0 \longleftrightarrow x = 0$   
 ⟨proof⟩

**lemma** *unat-0-iff*:  $\text{unat } x = 0 \longleftrightarrow x = 0$   
 ⟨proof⟩

**lemma** *unat-0*:  $\text{unat } 0 = 0$   
 ⟨proof⟩

**lemma** *unat-gt-0*:  $0 < \text{unat } x \longleftrightarrow x \neq 0$   
 ⟨proof⟩

**lemma** *ucast-0*:  $\text{ucast } 0 = 0$   
 ⟨proof⟩

**lemma** *sint-0*:  $\text{sint } 0 = 0$

⟨proof⟩

**lemma** *scast-0*: *scast 0 = 0*

⟨proof⟩

**lemma** *sint-n1*: *sint (- 1) = - 1*

⟨proof⟩

**lemma** *scast-n1*: *scast (- 1) = - 1*

⟨proof⟩

**lemma** *wint-1*: *wint (1::'a::len word) = 1*

⟨proof⟩

**lemma** *unat-1*: *unat (1::'a::len word) = 1*

⟨proof⟩

**lemma** *ucast-1*: *ucast (1::'a::len word) = 1*

⟨proof⟩

**instantiation** *word :: (len) size*

**begin**

**lift-definition** *size-word :: ⟨'a word ⇒ nat⟩*

**is** ⟨λ-. *LENGTH('a)⟩* ⟨proof⟩

**instance** ⟨proof⟩

**end**

**lemma** *word-size [code]*:

⟨size *w = LENGTH('a)⟩* **for** *w :: ⟨'a::len word⟩*

⟨proof⟩

**lemma** *word-size-gt-0 [iff]*: *0 < size w*

**for** *w :: 'a::len word*

⟨proof⟩

**lemmas** *lens-gt-0 = word-size-gt-0 len-gt-0*

**lemma** *lens-not-0 [iff]*:

⟨size *w ≠ 0⟩* **for** *w :: ⟨'a::len word⟩*

⟨proof⟩

**lift-definition** *source-size :: ⟨('a::len word ⇒ 'b) ⇒ nat⟩*

**is** ⟨λ-. *LENGTH('a)⟩* ⟨proof⟩

**lift-definition** *target-size :: ⟨('a ⇒ 'b::len word) ⇒ nat⟩*

**is** ⟨λ-. *LENGTH('b)⟩* ⟨proof⟩

**lift-definition** *is-up* ::  $\langle ('a::\text{len word} \Rightarrow 'b::\text{len word}) \Rightarrow \text{bool} \rangle$   
**is**  $\langle \lambda-. \text{LENGTH}('a) \leq \text{LENGTH}('b) \rangle \langle \text{proof} \rangle$

**lift-definition** *is-down* ::  $\langle ('a::\text{len word} \Rightarrow 'b::\text{len word}) \Rightarrow \text{bool} \rangle$   
**is**  $\langle \lambda-. \text{LENGTH}('a) \geq \text{LENGTH}('b) \rangle \langle \text{proof} \rangle$

**lemma** *is-up-eq*:  
 $\langle \text{is-up } f \longleftrightarrow \text{source-size } f \leq \text{target-size } f \rangle$   
**for**  $f :: \langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *is-down-eq*:  
 $\langle \text{is-down } f \longleftrightarrow \text{target-size } f \leq \text{source-size } f \rangle$   
**for**  $f :: \langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lift-definition** *word-int-case* ::  $\langle (\text{int} \Rightarrow 'b) \Rightarrow 'a::\text{len word} \Rightarrow 'b \rangle$   
**is**  $\langle \lambda f. f \circ \text{take-bit LENGTH}('a) \rangle \langle \text{proof} \rangle$

**lemma** *word-int-case-eq-uint* [code]:  
 $\langle \text{word-int-case } f w = f (\text{uint } w) \rangle$   
 $\langle \text{proof} \rangle$

### translations

*case*  $x$  of *XCONST of-int*  $y \Rightarrow b \Rightarrow \text{CONST word-int-case } (\lambda y. b) x$   
*case*  $x$  of (*XCONST of-int* ::  $'a$ )  $y \Rightarrow b \rightarrow \text{CONST word-int-case } (\lambda y. b) x$

## 106.6 Arithmetic operations

**lemma** *div-word-self*:  
 $\langle w \text{ div } w = 1 \rangle$  **if**  $\langle w \neq 0 \rangle$  **for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *mod-word-self* [simp]:  
 $\langle w \text{ mod } w = 0 \rangle$  **for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *div-word-less*:  
 $\langle w \text{ div } v = 0 \rangle$  **if**  $\langle w < v \rangle$  **for**  $w v :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *mod-word-less*:  
 $\langle w \text{ mod } v = w \rangle$  **if**  $\langle w < v \rangle$  **for**  $w v :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *div-word-one* [simp]:  
 $\langle 1 \text{ div } w = \text{of-bool } (w = 1) \rangle$  **for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *mod-word-one* [*simp*]:  
 $\langle 1 \text{ mod } w = 1 - w * \text{of-bool } (w = 1) \rangle$  **for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *div-word-by-minus-1-eq* [*simp*]:  
 $\langle w \text{ div } - 1 = \text{of-bool } (w = - 1) \rangle$  **for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *mod-word-by-minus-1-eq* [*simp*]:  
 $\langle w \text{ mod } - 1 = w * \text{of-bool } (w < - 1) \rangle$  **for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

Legacy theorems:

**lemma** *word-add-def* [*code*]:  
 $a + b = \text{word-of-int } (\text{uint } a + \text{uint } b)$   
 $\langle \text{proof} \rangle$

**lemma** *word-sub-wi* [*code*]:  
 $a - b = \text{word-of-int } (\text{uint } a - \text{uint } b)$   
 $\langle \text{proof} \rangle$

**lemma** *word-mult-def* [*code*]:  
 $a * b = \text{word-of-int } (\text{uint } a * \text{uint } b)$   
 $\langle \text{proof} \rangle$

**lemma** *word-minus-def* [*code*]:  
 $- a = \text{word-of-int } (- \text{uint } a)$   
 $\langle \text{proof} \rangle$

**lemma** *word-0-wi*:  
 $0 = \text{word-of-int } 0$   
 $\langle \text{proof} \rangle$

**lemma** *word-1-wi*:  
 $1 = \text{word-of-int } 1$   
 $\langle \text{proof} \rangle$

**lift-definition** *word-succ* ::  $'a::\text{len word} \Rightarrow 'a \text{ word}$  **is**  $\lambda x. x + 1$   
 $\langle \text{proof} \rangle$

**lift-definition** *word-pred* ::  $'a::\text{len word} \Rightarrow 'a \text{ word}$  **is**  $\lambda x. x - 1$   
 $\langle \text{proof} \rangle$

**lemma** *word-succ-alt* [*code*]:  
 $\text{word-succ } a = \text{word-of-int } (\text{uint } a + 1)$   
 $\langle \text{proof} \rangle$

**lemma** *word-pred-alt* [*code*]:

*word-pred*  $a = \text{word-of-int } (\text{uint } a - 1)$   
 ⟨proof⟩

**lemmas** *word-arith-wis* =  
*word-add-def word-sub-wi word-mult-def*  
*word-minus-def word-succ-alt word-pred-alt*  
*word-0-wi word-1-wi*

**lemma** *wi-homs*:

**shows** *wi-hom-add*:  $\text{word-of-int } a + \text{word-of-int } b = \text{word-of-int } (a + b)$   
**and** *wi-hom-sub*:  $\text{word-of-int } a - \text{word-of-int } b = \text{word-of-int } (a - b)$   
**and** *wi-hom-mult*:  $\text{word-of-int } a * \text{word-of-int } b = \text{word-of-int } (a * b)$   
**and** *wi-hom-neg*:  $-\text{word-of-int } a = \text{word-of-int } (-a)$   
**and** *wi-hom-succ*:  $\text{word-succ } (\text{word-of-int } a) = \text{word-of-int } (a + 1)$   
**and** *wi-hom-pred*:  $\text{word-pred } (\text{word-of-int } a) = \text{word-of-int } (a - 1)$   
 ⟨proof⟩

**lemmas** *wi-hom-syms* = *wi-homs* [*symmetric*]

**lemmas** *word-of-int-homs* = *wi-homs word-0-wi word-1-wi*

**lemmas** *word-of-int-hom-syms* = *word-of-int-homs* [*symmetric*]

**lemma** *double-eq-zero-iff*:

⟨ $2 * a = 0 \longleftrightarrow a = 0 \vee a = 2 \wedge (\text{LENGTH}'a) - \text{Suc } 0$ ⟩  
**for**  $a :: \langle 'a :: \text{len word} \rangle$   
 ⟨proof⟩

## 106.7 Ordering

**lift-definition** *word-sle* ::  $\langle 'a :: \text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool} \rangle$

**is**  $\langle \lambda k l. \text{signed-take-bit } (\text{LENGTH}'a) - \text{Suc } 0 \ k \leq \text{signed-take-bit } (\text{LENGTH}'a) - \text{Suc } 0 \ l \rangle$   
 ⟨proof⟩

**lift-definition** *word-sless* ::  $\langle 'a :: \text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool} \rangle$

**is**  $\langle \lambda k l. \text{signed-take-bit } (\text{LENGTH}'a) - \text{Suc } 0 \ k < \text{signed-take-bit } (\text{LENGTH}'a) - \text{Suc } 0 \ l \rangle$   
 ⟨proof⟩

**notation**

*word-sle*     $\langle \langle '(\leq s') \rangle \rangle$  **and**  
*word-sle*     $\langle \langle \langle \text{notation} = \langle \text{infix } \leq s \rangle \rangle - / \leq s - \rangle \rangle$  [51, 51] 50) **and**  
*word-sless*     $\langle \langle '(< s') \rangle \rangle$  **and**  
*word-sless*     $\langle \langle \langle \text{notation} = \langle \text{infix } < s \rangle \rangle - / < s - \rangle \rangle$  [51, 51] 50)

**notation** (*input*)

*word-sle*     $\langle \langle \langle \text{notation} = \langle \text{infix } \leq s \rangle \rangle - / \leq s - \rangle \rangle$  [51, 51] 50)



**lemma** *word-sle-eq* [*code*]:  
 $\langle a \leq_s b \iff \text{sint } a \leq \text{sint } b \rangle$   
 $\langle \text{proof} \rangle$

**lemma** [*code*]:  
 $\langle a <_s b \iff \text{sint } a < \text{sint } b \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-ordering*:  $\langle \text{ordering word-sle word-sless} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-linorder*:  $\langle \text{class.linorder word-sle word-sless} \rangle$   
 $\langle \text{proof} \rangle$

**interpretation** *signed*: *linorder word-sle word-sless*  
 $\langle \text{proof} \rangle$

**lemma** *word-sless-eq*:  
 $\langle x <_s y \iff x \leq_s y \wedge x \neq y \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-less-alt*:  $a < b \iff \text{uint } a < \text{uint } b$   
 $\langle \text{proof} \rangle$

**lemma** *word-zero-le* [*simp*]:  $0 \leq y$   
**for**  $y :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *word-m1-ge* [*simp*]:  $\text{word-pred } 0 \geq y$   
 $\langle \text{proof} \rangle$

**lemma** *word-n1-ge* [*simp*]:  $y \leq -1$   
**for**  $y :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemmas** *word-not-simps* [*simp*] =  
*word-zero-le* [THEN *leD*] *word-m1-ge* [THEN *leD*] *word-n1-ge* [THEN *leD*]

**lemma** *word-gt-0*:  $0 < y \iff 0 \neq y$   
**for**  $y :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemmas** *word-gt-0-no* [*simp*] = *word-gt-0* [of numeral  $y$ ] **for**  $y$

**lemma** *word-sless-alt*:  $a <_s b \iff \text{sint } a < \text{sint } b$   
 $\langle \text{proof} \rangle$

**lemma** *word-le-nat-alt*:  $a \leq b \iff \text{unat } a \leq \text{unat } b$   
 $\langle \text{proof} \rangle$

**lemma** *word-less-nat-alt*:  $a < b \longleftrightarrow \text{unat } a < \text{unat } b$   
 ⟨*proof*⟩

**lemmas** *unat-mono* = *word-less-nat-alt* [THEN *iffD1*]

**instance** *word* :: (*len*) *wellorder*  
 ⟨*proof*⟩

**lemma** *wi-less*:  
 ( $\text{word-of-int } n < (\text{word-of-int } m :: 'a::\text{len word})$ ) =  
 ( $n \bmod 2 \wedge \text{LENGTH}('a) < m \bmod 2 \wedge \text{LENGTH}('a)$ )  
 ⟨*proof*⟩

**lemma** *wi-le*:  
 ( $\text{word-of-int } n \leq (\text{word-of-int } m :: 'a::\text{len word})$ ) =  
 ( $n \bmod 2 \wedge \text{LENGTH}('a) \leq m \bmod 2 \wedge \text{LENGTH}('a)$ )  
 ⟨*proof*⟩

## 106.8 Bit-wise operations

**context**

**includes** *bit-operations-syntax*

**begin**

**lemma** *take-bit-word-eq-self*:  
 ⟨*take-bit*  $n \ w = w$ ⟩ **if** ⟨ $\text{LENGTH}('a) \leq n$ ⟩ **for**  $w :: \langle 'a::\text{len word} \rangle$   
 ⟨*proof*⟩

**lemma** *take-bit-length-eq* [*simp*]:  
 ⟨*take-bit*  $\text{LENGTH}('a) \ w = w$ ⟩ **for**  $w :: \langle 'a::\text{len word} \rangle$   
 ⟨*proof*⟩

**lemma** *bit-word-of-int-iff*:  
 ⟨*bit* ( $\text{word-of-int } k :: 'a::\text{len word}$ )  $n \longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } k \ n$ ⟩  
 ⟨*proof*⟩

**lemma** *bit-uint-iff*:  
 ⟨*bit* ( $\text{uint } w$ )  $n \longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } w \ n$ ⟩  
**for**  $w :: \langle 'a::\text{len word} \rangle$   
 ⟨*proof*⟩

**lemma** *bit-sint-iff*:  
 ⟨*bit* ( $\text{sint } w$ )  $n \longleftrightarrow n \geq \text{LENGTH}('a) \wedge \text{bit } w \ (\text{LENGTH}('a) - 1) \vee \text{bit } w \ n$ ⟩  
**for**  $w :: \langle 'a::\text{len word} \rangle$   
 ⟨*proof*⟩

**lemma** *bit-word-ucast-iff*:  
 ⟨*bit* ( $\text{ucast } w :: 'b::\text{len word}$ )  $n \longleftrightarrow n < \text{LENGTH}('a) \wedge n < \text{LENGTH}('b) \wedge$

*bit w n*  
**for** *w :: 'a::len word*  
 ⟨*proof*⟩

**lemma** *bit-word-scst-iff*:  
 ⟨*bit (scst w :: 'b::len word) n*  $\longleftrightarrow$   
 $n < \text{LENGTH}('b) \wedge (\text{bit } w \ n \vee \text{LENGTH}('a) \leq n \wedge \text{bit } w \ (\text{LENGTH}('a) - \text{Suc } 0))$ ⟩  
**for** *w :: 'a::len word*  
 ⟨*proof*⟩

**lemma** *bit-word-iff-drop-bit-and* [*code*]:  
 ⟨*bit a n*  $\longleftrightarrow$  *drop-bit n a AND 1 = 1*⟩ **for** *a :: 'a::len word*  
 ⟨*proof*⟩

**lemma**  
*word-not-def*: *NOT (a::'a::len word) = word-of-int (NOT (uint a))*  
**and** *word-and-def*: *(a::'a word) AND b = word-of-int (uint a AND uint b)*  
**and** *word-or-def*: *(a::'a word) OR b = word-of-int (uint a OR uint b)*  
**and** *word-xor-def*: *(a::'a word) XOR b = word-of-int (uint a XOR uint b)*  
 ⟨*proof*⟩

**definition** *even-word* :: *'a::len word  $\Rightarrow$  bool*  
**where** [*code-abbrev*]: *even-word = even*⟩

**lemma** *even-word-iff* [*code*]:  
 ⟨*even-word a*  $\longleftrightarrow$  *a AND 1 = 0*⟩  
 ⟨*proof*⟩

**lemma** *map-bit-range-eq-if-take-bit-eq*:  
 ⟨*map (bit k) [0..*n*] = map (bit l) [0..*n*]*⟩  
**if** *take-bit n k = take-bit n l* **for** *k l :: int*  
 ⟨*proof*⟩

**lemma**  
*take-bit-word-Bit0-eq* [*simp*]: *take-bit (numeral n) (numeral (num.Bit0 m)) :: 'a::len word*  
 =  $2 * \text{take-bit (pred-numeral n) (numeral m)}$  (is ?P)  
**and** *take-bit-word-Bit1-eq* [*simp*]: *take-bit (numeral n) (numeral (num.Bit1 m)) :: 'a::len word*  
 =  $1 + 2 * \text{take-bit (pred-numeral n) (numeral m)}$  (is ?Q)  
**and** *take-bit-word-minus-Bit0-eq* [*simp*]: *take-bit (numeral n) (- numeral (num.Bit0 m)) :: 'a::len word*  
 =  $2 * \text{take-bit (pred-numeral n) (- numeral m)}$  (is ?R)  
**and** *take-bit-word-minus-Bit1-eq* [*simp*]: *take-bit (numeral n) (- numeral (num.Bit1 m)) :: 'a::len word*  
 =  $1 + 2 * \text{take-bit (pred-numeral n) (- numeral (Num.inc m))}$  (is ?S)  
 ⟨*proof*⟩

### 106.9 More shift operations

**lift-definition** *signed-drop-bit* ::  $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a::\text{len word} \rangle$   
**is**  $\langle \lambda n. \text{drop-bit } n \circ \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *bit-signed-drop-bit-iff* [*bit-simps*]:  
 $\langle \text{bit } (\text{signed-drop-bit } m \ w) \ n \longleftrightarrow \text{bit } w \ (\text{if } \text{LENGTH}('a) - m \leq n \wedge n < \text{LENGTH}('a) \text{ then } \text{LENGTH}('a) - 1 \text{ else } m + n) \rangle$   
**for**  $w :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** [*code*]:  
 $\langle \text{Word.the-int } (\text{signed-drop-bit } n \ w) = \text{take-bit } \text{LENGTH}('a) \ (\text{drop-bit } n \ (\text{Word.the-signed-int } w)) \rangle$   
**for**  $w :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *signed-drop-bit-of-0* [*simp*]:  
 $\langle \text{signed-drop-bit } n \ 0 = 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-drop-bit-of-minus-1* [*simp*]:  
 $\langle \text{signed-drop-bit } n \ (-1) = -1 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-drop-bit-signed-drop-bit* [*simp*]:  
 $\langle \text{signed-drop-bit } m \ (\text{signed-drop-bit } n \ w) = \text{signed-drop-bit } (m + n) \ w \rangle$   
**for**  $w :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *signed-drop-bit-0* [*simp*]:  
 $\langle \text{signed-drop-bit } 0 \ w = w \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *sint-signed-drop-bit-eq*:  
 $\langle \text{sint } (\text{signed-drop-bit } n \ w) = \text{drop-bit } n \ (\text{sint } w) \rangle$   
 $\langle \text{proof} \rangle$

### 106.10 Single-bit operations

**lemma** *set-bit-eq-idem-iff*:  
 $\langle \text{Bit-Operations.set-bit } n \ w = w \longleftrightarrow \text{bit } w \ n \vee n \geq \text{LENGTH}('a) \rangle$   
**for**  $w :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *unset-bit-eq-idem-iff*:  
 $\langle \text{unset-bit } n \ w = w \longleftrightarrow \text{bit } w \ n \longrightarrow n \geq \text{LENGTH}('a) \rangle$   
**for**  $w :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *flip-bit-eq-idem-iff*:  
 $\langle \text{flip-bit } n \ w = w \longleftrightarrow n \geq \text{LENGTH}('a) \rangle$   
**for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

## 106.11 Rotation

**lift-definition** *word-rotr* ::  $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'a::\text{len word} \rangle$   
**is**  $\langle \lambda n \ k. \text{concat-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a))$   
 $(\text{drop-bit } (n \bmod \text{LENGTH}('a)) (\text{take-bit } \text{LENGTH}('a) \ k))$   
 $(\text{take-bit } (n \bmod \text{LENGTH}('a)) \ k) \rangle$   
 $\langle \text{proof} \rangle$

**lift-definition** *word-rotl* ::  $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'a::\text{len word} \rangle$   
**is**  $\langle \lambda n \ k. \text{concat-bit } (n \bmod \text{LENGTH}('a))$   
 $(\text{drop-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) (\text{take-bit } \text{LENGTH}('a) \ k))$   
 $(\text{take-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) \ k) \rangle$   
 $\langle \text{proof} \rangle$

**lift-definition** *word-roti* ::  $\langle \text{int} \Rightarrow 'a::\text{len word} \Rightarrow 'a::\text{len word} \rangle$   
**is**  $\langle \lambda r \ k. \text{concat-bit } (\text{LENGTH}('a) - \text{nat } (r \bmod \text{int } \text{LENGTH}('a)))$   
 $(\text{drop-bit } (\text{nat } (r \bmod \text{int } \text{LENGTH}('a))) (\text{take-bit } \text{LENGTH}('a) \ k))$   
 $(\text{take-bit } (\text{nat } (r \bmod \text{int } \text{LENGTH}('a))) \ k) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-rotl-eq-word-rotr* [code]:  
 $\langle \text{word-rotl } n = (\text{word-rotr } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) :: 'a::\text{len word}$   
 $\Rightarrow 'a \ \text{word}) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-roti-eq-word-rotr-word-rotl* [code]:  
 $\langle \text{word-roti } i \ w =$   
 $(\text{if } i \geq 0 \ \text{then } \text{word-rotr } (\text{nat } i) \ w \ \text{else } \text{word-rotl } (\text{nat } (- i)) \ w) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *bit-word-rotr-iff* [bit-simps]:  
 $\langle \text{bit } (\text{word-rotr } m \ w) \ n \longleftrightarrow$   
 $n < \text{LENGTH}('a) \wedge \text{bit } w \ ((n + m) \bmod \text{LENGTH}('a)) \rangle$   
**for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *bit-word-rotl-iff* [bit-simps]:  
 $\langle \text{bit } (\text{word-rotl } m \ w) \ n \longleftrightarrow$   
 $n < \text{LENGTH}('a) \wedge \text{bit } w \ ((n + (\text{LENGTH}('a) - m \bmod \text{LENGTH}('a))) \bmod$   
 $\text{LENGTH}('a)) \rangle$   
**for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *bit-word-roti-iff* [*bit-simps*]:

⟨*bit* (*word-roti* *k w*) *n*  $\longleftrightarrow$   
 $n < \text{LENGTH}('a) \wedge \text{bit } w \text{ (nat ((int } n + k) \text{ mod int LENGTH}('a))}$ ⟩  
**for** *w* :: ⟨*a*::*len word*⟩  
 ⟨*proof*⟩

**lemma** *uint-word-rotr-eq*:

⟨*uint* (*word-rotr* *n w*) = *concat-bit* (*LENGTH*('a) - *n mod LENGTH*('a))  
 (*drop-bit* (*n mod LENGTH*('a)) (*uint w*))  
 (*uint* (*take-bit* (*n mod LENGTH*('a)) *w*))⟩  
**for** *w* :: ⟨*a*::*len word*⟩  
 ⟨*proof*⟩

**lemma** [*code*]:

⟨*Word.the-int* (*word-rotr* *n w*) = *concat-bit* (*LENGTH*('a) - *n mod LENGTH*('a))  
 (*drop-bit* (*n mod LENGTH*('a)) (*Word.the-int w*))  
 (*Word.the-int* (*take-bit* (*n mod LENGTH*('a)) *w*))⟩  
**for** *w* :: ⟨*a*::*len word*⟩  
 ⟨*proof*⟩

## 106.12 Split and cat operations

**lift-definition** *word-cat* :: ⟨*a*::*len word*  $\Rightarrow$  ⟨*b*::*len word*  $\Rightarrow$  ⟨*c*::*len word*⟩

**is** ⟨ $\lambda k l. \text{concat-bit LENGTH}('b) l \text{ (take-bit LENGTH}('a) k)$ ⟩  
 ⟨*proof*⟩

**lemma** *word-cat-eq*:

⟨(*word-cat* *v w* :: ⟨*c*::*len word*) = *push-bit* *LENGTH*('b) (*ucast v*) + *ucast w*⟩  
**for** *v* :: ⟨*a*::*len word*⟩ **and** *w* :: ⟨*b*::*len word*⟩  
 ⟨*proof*⟩

**lemma** *word-cat-eq'* [*code*]:

⟨*word-cat* *a b* = *word-of-int* (*concat-bit* *LENGTH*('b) (*uint b*) (*uint a*))⟩  
**for** *a* :: ⟨*a*::*len word*⟩ **and** *b* :: ⟨*b*::*len word*⟩  
 ⟨*proof*⟩

**lemma** *bit-word-cat-iff* [*bit-simps*]:

⟨*bit* (*word-cat* *v w* :: ⟨*c*::*len word*) *n*  $\longleftrightarrow$   $n < \text{LENGTH}('c) \wedge (\text{if } n < \text{LENGTH}('b)$   
*then bit w n else bit v (n - LENGTH('b))*)⟩  
**for** *v* :: ⟨*a*::*len word*⟩ **and** *w* :: ⟨*b*::*len word*⟩  
 ⟨*proof*⟩

**definition** *word-split* :: ⟨*a*::*len word*  $\Rightarrow$  ⟨*b*::*len word*  $\times$  ⟨*c*::*len word*⟩

**where** ⟨*word-split* *w* =

(*ucast* (*drop-bit* *LENGTH*('c) *w*) :: ⟨*b*::*len word*, *ucast w* :: ⟨*c*::*len word*)⟩

**definition** *word-rcat* :: ⟨*a*::*len word list*  $\Rightarrow$  ⟨*b*::*len word*⟩

**where** ⟨*word-rcat* = *word-of-int*  $\circ$  *horner-sum uint* ( $2 \wedge \text{LENGTH}('a)$ )  $\circ$  *rev*)⟩

**106.13 More on conversions****lemma** *int-word-sint*:
$$\langle \text{sint} (\text{word-of-int } x :: 'a::\text{len word}) = (x + 2^{\wedge} (\text{LENGTH}('a) - 1)) \text{ mod } 2^{\wedge} \text{LENGTH}('a) - 2^{\wedge} (\text{LENGTH}('a) - 1) \rangle$$

*<proof>*

**lemma** *sint-sbintrunc'*:  $\text{sint} (\text{word-of-int } \text{bin} :: 'a \text{ word}) = \text{signed-take-bit} (\text{LENGTH}('a::\text{len}) - 1) \text{ bin}$ *<proof>***lemma** *uint-sint*:  $\text{uint } w = \text{take-bit } \text{LENGTH}('a) (\text{sint } w)$ **for**  $w :: 'a::\text{len word}$ *<proof>***lemma** *bintr-uint*:  $\text{LENGTH}('a) \leq n \implies \text{take-bit } n (\text{uint } w) = \text{uint } w$ **for**  $w :: 'a::\text{len word}$ *<proof>***lemma** *wi-bintr*: $\text{LENGTH}('a::\text{len}) \leq n \implies$  $\text{word-of-int} (\text{take-bit } n w) = (\text{word-of-int } w :: 'a \text{ word})$ *<proof>***lemma** *word-numeral-alt*:  $\text{numeral } b = \text{word-of-int} (\text{numeral } b)$ *<proof>***declare** *word-numeral-alt* [*symmetric, code-abbrev*]**lemma** *word-neg-numeral-alt*:  $-\text{numeral } b = \text{word-of-int} (-\text{numeral } b)$ *<proof>***declare** *word-neg-numeral-alt* [*symmetric, code-abbrev*]**lemma** *uint-bintrunc* [*simp*]:
$$\text{uint} (\text{numeral } \text{bin} :: 'a \text{ word}) = \text{take-bit} (\text{LENGTH}('a::\text{len})) (\text{numeral } \text{bin})$$
*<proof>***lemma** *uint-bintrunc-neg* [*simp*]:
$$\text{uint} (-\text{numeral } \text{bin} :: 'a \text{ word}) = \text{take-bit} (\text{LENGTH}('a::\text{len})) (-\text{numeral } \text{bin})$$
*<proof>***lemma** *sint-sbintrunc* [*simp*]:
$$\text{sint} (\text{numeral } \text{bin} :: 'a \text{ word}) = \text{signed-take-bit} (\text{LENGTH}('a::\text{len}) - 1) (\text{numeral } \text{bin})$$
*<proof>***lemma** *sint-sbintrunc-neg* [*simp*]:
$$\text{sint} (-\text{numeral } \text{bin} :: 'a \text{ word}) = \text{signed-take-bit} (\text{LENGTH}('a::\text{len}) - 1) (-\text{numeral } \text{bin})$$

*numeral bin*)  
 ⟨*proof*⟩

**lemma** *unat-bintrunc* [*simp*]:  
 $unat (numeral\ bin :: 'a::len\ word) = nat (take-bit (LENGTH('a)) (numeral\ bin))$   
 ⟨*proof*⟩

**lemma** *unat-bintrunc-neg* [*simp*]:  
 $unat (-\ numeral\ bin :: 'a::len\ word) = nat (take-bit (LENGTH('a)) (-\ numeral\ bin))$   
 ⟨*proof*⟩

**lemma** *size-0-eq*:  $size\ w = 0 \implies v = w$   
**for**  $v\ w :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *uint-ge-0* [*iff*]:  $0 \leq uint\ x$   
 ⟨*proof*⟩

**lemma** *uint-lt2p* [*iff*]:  $uint\ x < 2 \wedge LENGTH('a)$   
**for**  $x :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *sint-ge*:  $-(2 \wedge (LENGTH('a) - 1)) \leq sint\ x$   
**for**  $x :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *sint-lt*:  $sint\ x < 2 \wedge (LENGTH('a) - 1)$   
**for**  $x :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *uint-m2p-neg*:  $uint\ x - 2 \wedge LENGTH('a) < 0$   
**for**  $x :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *uint-m2p-not-non-neg*:  $\neg 0 \leq uint\ x - 2 \wedge LENGTH('a)$   
**for**  $x :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *lt2p-lem*:  $LENGTH('a) \leq n \implies uint\ w < 2 \wedge n$   
**for**  $w :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *uint-le-0-iff* [*simp*]:  $uint\ x \leq 0 \iff uint\ x = 0$   
 ⟨*proof*⟩

**lemma** *uint-nat*:  $uint\ w = int (unat\ w)$   
 ⟨*proof*⟩



**lemma** *uint-numeral*:  $uint \text{ (numeral } b :: 'a::len \text{ word)} = numeral \ b \text{ mod } 2^{\wedge} LENGTH('a)$   
 ⟨proof⟩

**lemma** *uint-neg-numeral*:  $uint \text{ (- numeral } b :: 'a::len \text{ word)} = - numeral \ b \text{ mod } 2^{\wedge} LENGTH('a)$   
 ⟨proof⟩

**lemma** *unat-numeral*:  $unat \text{ (numeral } b :: 'a::len \text{ word)} = numeral \ b \text{ mod } 2^{\wedge} LENGTH('a)$   
 ⟨proof⟩

**lemma** *sint-numeral*:  
 $sint \text{ (numeral } b :: 'a::len \text{ word)} =$   
 $(numeral \ b + 2^{\wedge}(LENGTH('a) - 1)) \text{ mod } 2^{\wedge} LENGTH('a) - 2^{\wedge}(LENGTH('a) - 1)$   
 - 1)  
 ⟨proof⟩

**lemma** *word-of-int-0* [*simp*, *code-post*]:  $word\text{-of-int } 0 = 0$   
 ⟨proof⟩

**lemma** *word-of-int-1* [*simp*, *code-post*]:  $word\text{-of-int } 1 = 1$   
 ⟨proof⟩

**lemma** *word-of-int-neg-1* [*simp*]:  $word\text{-of-int } (- 1) = - 1$   
 ⟨proof⟩

**lemma** *word-of-int-numeral* [*simp*]:  $(word\text{-of-int (numeral } bin) :: 'a::len \text{ word)} = numeral \ bin$   
 ⟨proof⟩

**lemma** *word-int-case-wi*:  
 $word\text{-int-case } f \text{ (word-of-int } i :: 'b \text{ word)} = f \text{ (} i \text{ mod } 2^{\wedge} LENGTH('b::len)$   
 ⟨proof⟩

**lemma** *word-int-split*:  
 $P \text{ (word-int-case } f \ x) =$   
 $(\forall i. x = (word\text{-of-int } i :: 'b::len \text{ word)} \wedge 0 \leq i \wedge i < 2^{\wedge} LENGTH('b) \longrightarrow P$   
 $(f \ i))$   
 ⟨proof⟩

**lemma** *word-int-split-asm*:  
 $P \text{ (word-int-case } f \ x) =$   
 $(\exists n. x = (word\text{-of-int } n :: 'b::len \text{ word)} \wedge 0 \leq n \wedge n < 2^{\wedge} LENGTH('b::len)$   
 $\wedge \neg P \text{ (} f \ n))$   
 ⟨proof⟩

**lemma** *uint-range-size*:  $0 \leq uint \ w \wedge uint \ w < 2^{\wedge} size \ w$   
 ⟨proof⟩

**lemma** *sint-range-size*:  $-(2 \wedge (\text{size } w - \text{Suc } 0)) \leq \text{sint } w \wedge \text{sint } w < 2 \wedge (\text{size } w - \text{Suc } 0)$   
 ⟨proof⟩

**lemma** *sint-above-size*:  $2 \wedge (\text{size } w - 1) \leq x \implies \text{sint } w < x$   
**for**  $w :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *sint-below-size*:  $x \leq -(2 \wedge (\text{size } w - 1)) \implies x \leq \text{sint } w$   
**for**  $w :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-unat-eq-iff*:  
 $\langle v = w \longleftrightarrow \text{unat } v = \text{unat } w \rangle$   
**for**  $v w :: \langle 'a::\text{len word} \rangle$   
 ⟨proof⟩

## 106.14 Testing bits

**lemma** *bin-nth-uint-imp*:  $\text{bit } (\text{uint } w) n \implies n < \text{LENGTH}('a)$   
**for**  $w :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *bin-nth-sint*:  
 $\text{LENGTH}('a) \leq n \implies$   
 $\text{bit } (\text{sint } w) n = \text{bit } (\text{sint } w) (\text{LENGTH}('a) - 1)$   
**for**  $w :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *num-of-bintr'*:  
 $\text{take-bit } (\text{LENGTH}('a::\text{len})) (\text{numeral } a :: \text{int}) = (\text{numeral } b) \implies$   
 $\text{numeral } a = (\text{numeral } b :: 'a \text{ word})$   
 ⟨proof⟩

**lemma** *num-of-sbintr'*:  
 $\text{signed-take-bit } (\text{LENGTH}('a::\text{len}) - 1) (\text{numeral } a :: \text{int}) = (\text{numeral } b) \implies$   
 $\text{numeral } a = (\text{numeral } b :: 'a \text{ word})$   
 ⟨proof⟩

**lemma** *num-abs-bintr*:  
 $(\text{numeral } x :: 'a \text{ word}) =$   
 $\text{word-of-int } (\text{take-bit } (\text{LENGTH}('a::\text{len})) (\text{numeral } x))$   
 ⟨proof⟩

**lemma** *num-abs-sbintr*:  
 $(\text{numeral } x :: 'a \text{ word}) =$   
 $\text{word-of-int } (\text{signed-take-bit } (\text{LENGTH}('a::\text{len}) - 1) (\text{numeral } x))$   
 ⟨proof⟩

*cast* – note, no arg for new length, as it’s determined by type of result, thus in *cast w = w*, the type means cast to length of *w*!

**lemma** *bit-ucast-iff*:

⟨*bit (ucast a :: 'a::len word) n* ⟷ *n < LENGTH('a::len) ∧ bit a n*⟩  
 ⟨*proof*⟩

**lemma** *ucast-id [simp]*: *ucast w = w*

⟨*proof*⟩

**lemma** *scast-id [simp]*: *scast w = w*

⟨*proof*⟩

**lemma** *ucast-mask-eq*:

⟨*ucast (mask n :: 'b word) = mask (min LENGTH('b::len) n)*⟩  
 ⟨*proof*⟩

**lemma** *ucast-bintr [simp]*:

*ucast (numeral w :: 'a::len word) =*  
*word-of-int (take-bit (LENGTH('a)) (numeral w))*  
 ⟨*proof*⟩

**lemma** *scast-sbintr [simp]*:

*scast (numeral w :: 'a::len word) =*  
*word-of-int (signed-take-bit (LENGTH('a) - Suc 0) (numeral w))*  
 ⟨*proof*⟩

**lemma** *source-size*: *source-size (c::'a::len word ⇒ -) = LENGTH('a)*

⟨*proof*⟩

**lemma** *target-size*: *target-size (c::- ⇒ 'b::len word) = LENGTH('b)*

⟨*proof*⟩

**lemma** *is-down*: *is-down c* ⟷ *LENGTH('b) ≤ LENGTH('a)*

**for** *c :: 'a::len word ⇒ 'b::len word*  
 ⟨*proof*⟩

**lemma** *is-up*: *is-up c* ⟷ *LENGTH('a) ≤ LENGTH('b)*

**for** *c :: 'a::len word ⇒ 'b::len word*  
 ⟨*proof*⟩

**lemma** *is-up-down*:

⟨*is-up c* ⟷ *is-down d*⟩  
**for** *c :: 'a::len word ⇒ 'b::len word*  
**and** *d :: 'b::len word ⇒ 'a::len word*  
 ⟨*proof*⟩

**context**

**fixes** *dummy-types :: 'a::len × 'b::len*⟩

**begin**

**private abbreviation** (*input*) *UCAST* ::  $\langle 'a::len\ word \Rightarrow 'b::len\ word \rangle$   
**where**  $\langle UCAST == ucast \rangle$

**private abbreviation** (*input*) *SCAST* ::  $\langle 'a::len\ word \Rightarrow 'b::len\ word \rangle$   
**where**  $\langle SCAST == scast \rangle$

**lemma** *down-cast-same*:  
 $\langle UCAST = scast \rangle$  **if**  $\langle is-down\ UCAST \rangle$   
 $\langle proof \rangle$

**lemma** *sint-up-scast*:  
 $\langle sint\ (SCAST\ w) = sint\ w \rangle$  **if**  $\langle is-up\ SCAST \rangle$   
 $\langle proof \rangle$

**lemma** *uint-up-ucast*:  
 $\langle uint\ (UCAST\ w) = uint\ w \rangle$  **if**  $\langle is-up\ UCAST \rangle$   
 $\langle proof \rangle$

**lemma** *ucast-up-ucast*:  
 $\langle ucast\ (UCAST\ w) = ucast\ w \rangle$  **if**  $\langle is-up\ UCAST \rangle$   
 $\langle proof \rangle$

**lemma** *ucast-up-ucast-id*:  
 $\langle ucast\ (UCAST\ w) = w \rangle$  **if**  $\langle is-up\ UCAST \rangle$   
 $\langle proof \rangle$

**lemma** *scast-up-scast*:  
 $\langle scast\ (SCAST\ w) = scast\ w \rangle$  **if**  $\langle is-up\ SCAST \rangle$   
 $\langle proof \rangle$

**lemma** *scast-up-scast-id*:  
 $\langle scast\ (SCAST\ w) = w \rangle$  **if**  $\langle is-up\ SCAST \rangle$   
 $\langle proof \rangle$

**lemma** *isduu*:  
 $\langle is-up\ UCAST \rangle$  **if**  $\langle is-down\ d \rangle$   
**for**  $d :: \langle 'b\ word \Rightarrow 'a\ word \rangle$   
 $\langle proof \rangle$

**lemma** *isdus*:  
 $\langle is-up\ SCAST \rangle$  **if**  $\langle is-down\ d \rangle$   
**for**  $d :: \langle 'b\ word \Rightarrow 'a\ word \rangle$   
 $\langle proof \rangle$

**lemmas** *ucast-down-ucast-id = isduu* [THEN *ucast-up-ucast-id*]

**lemmas** *scast-down-scast-id = isdus* [THEN *scast-up-scast-id*]

**lemma** *up-ucast-surj*:

$\langle \text{surj } (\text{ucast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle$  **if**  $\langle \text{is-up } \text{UCAST} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *up-scast-surj*:

$\langle \text{surj } (\text{scast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle$  **if**  $\langle \text{is-up } \text{SCAST} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *down-ucast-inj*:

$\langle \text{inj-on } \text{UCAST } A \rangle$  **if**  $\langle \text{is-down } (\text{ucast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *down-scast-inj*:

$\langle \text{inj-on } \text{SCAST } A \rangle$  **if**  $\langle \text{is-down } (\text{scast} :: 'b \text{ word} \Rightarrow 'a \text{ word}) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *ucast-down-wi*:

$\langle \text{UCAST } (\text{word-of-int } x) = \text{word-of-int } x \rangle$  **if**  $\langle \text{is-down } \text{UCAST} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *ucast-down-no*:

$\langle \text{UCAST } (\text{numeral } \text{bin}) = \text{numeral } \text{bin} \rangle$  **if**  $\langle \text{is-down } \text{UCAST} \rangle$   
 $\langle \text{proof} \rangle$

**end**

**lemmas** *word-log-defs* = *word-and-def word-or-def word-xor-def word-not-def*

**lemma** *bit-last-iff*:

$\langle \text{bit } w (\text{LENGTH}('a) - \text{Suc } 0) \longleftrightarrow \text{sint } w < 0 \rangle$  (**is**  $\langle ?P \longleftrightarrow ?Q \rangle$ )  
**for**  $w :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *drop-bit-eq-zero-iff-not-bit-last*:

$\langle \text{drop-bit } (\text{LENGTH}('a) - \text{Suc } 0) w = 0 \longleftrightarrow \neg \text{bit } w (\text{LENGTH}('a) - \text{Suc } 0) \rangle$   
**for**  $w :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *unat-div*:

$\langle \text{unat } (x \text{ div } y) = \text{unat } x \text{ div } \text{unat } y \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *unat-mod*:

$\langle \text{unat } (x \text{ mod } y) = \text{unat } x \text{ mod } \text{unat } y \rangle$   
 $\langle \text{proof} \rangle$

## 106.15 Word Arithmetic

**lemmas** *less-eq-word-numeral-numeral* [*simp*] =

*word-le-def* [of  $\langle \text{numeral } a \rangle \langle \text{numeral } b \rangle$ , simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$   
**lemmas** *less-word-numeral-numeral* [simp] =  
*word-less-def* [of  $\langle \text{numeral } a \rangle \langle \text{numeral } b \rangle$ , simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$   
**lemmas** *less-eq-word-minus-numeral-numeral* [simp] =  
*word-le-def* [of  $\langle - \text{numeral } a \rangle \langle \text{numeral } b \rangle$ , simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$   
**lemmas** *less-word-minus-numeral-numeral* [simp] =  
*word-less-def* [of  $\langle - \text{numeral } a \rangle \langle \text{numeral } b \rangle$ , simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$   
**lemmas** *less-eq-word-numeral-minus-numeral* [simp] =  
*word-le-def* [of  $\langle \text{numeral } a \rangle \langle - \text{numeral } b \rangle$ , simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$   
**lemmas** *less-word-numeral-minus-numeral* [simp] =  
*word-less-def* [of  $\langle \text{numeral } a \rangle \langle - \text{numeral } b \rangle$ , simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$   
**lemmas** *less-eq-word-minus-numeral-minus-numeral* [simp] =  
*word-le-def* [of  $\langle - \text{numeral } a \rangle \langle - \text{numeral } b \rangle$ , simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$   
**lemmas** *less-word-minus-numeral-minus-numeral* [simp] =  
*word-less-def* [of  $\langle - \text{numeral } a \rangle \langle - \text{numeral } b \rangle$ , simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$   
**lemmas** *less-word-numeral-minus-1* [simp] =  
*word-less-def* [of  $\langle \text{numeral } a \rangle \langle - 1 \rangle$ , simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$   
**lemmas** *less-word-minus-numeral-minus-1* [simp] =  
*word-less-def* [of  $\langle - \text{numeral } a \rangle \langle - 1 \rangle$ , simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$   
  
**lemmas** *sless-eq-word-numeral-numeral* [simp] =  
*word-sle-eq* [of  $\langle \text{numeral } a \rangle \langle \text{numeral } b \rangle$ , simplified *sint-sbintrunc sint-sbintrunc-neg*]  
**for**  $a\ b$   
**lemmas** *sless-word-numeral-numeral* [simp] =  
*word-sless-alt* [of  $\langle \text{numeral } a \rangle \langle \text{numeral } b \rangle$ , simplified *sint-sbintrunc sint-sbintrunc-neg*]  
**for**  $a\ b$   
**lemmas** *sless-eq-word-minus-numeral-numeral* [simp] =  
*word-sle-eq* [of  $\langle - \text{numeral } a \rangle \langle \text{numeral } b \rangle$ , simplified *sint-sbintrunc sint-sbintrunc-neg*]  
**for**  $a\ b$

**lemmas** *sless-word-minus-numeral-numeral* [*simp*] =  
*word-sless-alt* [of  $\langle - \text{ numeral } a \rangle \langle \text{ numeral } b \rangle$ , *simplified sint-sbintrunc sint-sbintrunc-neg*]  
**for** *a b*

**lemmas** *sless-eq-word-numeral-minus-numeral* [*simp*] =  
*word-sle-eq* [of  $\langle \text{ numeral } a \rangle \langle - \text{ numeral } b \rangle$ , *simplified sint-sbintrunc sint-sbintrunc-neg*]  
**for** *a b*

**lemmas** *sless-word-numeral-minus-numeral* [*simp*] =  
*word-sless-alt* [of  $\langle \text{ numeral } a \rangle \langle - \text{ numeral } b \rangle$ , *simplified sint-sbintrunc sint-sbintrunc-neg*]  
**for** *a b*

**lemmas** *sless-eq-word-minus-numeral-minus-numeral* [*simp*] =  
*word-sle-eq* [of  $\langle - \text{ numeral } a \rangle \langle - \text{ numeral } b \rangle$ , *simplified sint-sbintrunc sint-sbintrunc-neg*]  
**for** *a b*

**lemmas** *sless-word-minus-numeral-minus-numeral* [*simp*] =  
*word-sless-alt* [of  $\langle - \text{ numeral } a \rangle \langle - \text{ numeral } b \rangle$ , *simplified sint-sbintrunc sint-sbintrunc-neg*]  
**for** *a b*

**lemmas** *div-word-numeral-numeral* [*simp*] =  
*word-div-def* [of  $\langle \text{ numeral } a \rangle \langle \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for** *a b*

**lemmas** *div-word-minus-numeral-numeral* [*simp*] =  
*word-div-def* [of  $\langle - \text{ numeral } a \rangle \langle \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for** *a b*

**lemmas** *div-word-numeral-minus-numeral* [*simp*] =  
*word-div-def* [of  $\langle \text{ numeral } a \rangle \langle - \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for** *a b*

**lemmas** *div-word-minus-numeral-minus-numeral* [*simp*] =  
*word-div-def* [of  $\langle - \text{ numeral } a \rangle \langle - \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for** *a b*

**lemmas** *div-word-minus-1-numeral* [*simp*] =  
*word-div-def* [of  $\langle - 1 \rangle \langle \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for** *a b*

**lemmas** *div-word-minus-1-minus-numeral* [*simp*] =  
*word-div-def* [of  $\langle - 1 \rangle \langle - \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for** *a b*

**lemmas** *mod-word-numeral-numeral* [*simp*] =  
*word-mod-def* [of  $\langle \text{ numeral } a \rangle \langle \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for** *a b*

**lemmas** *mod-word-minus-numeral-numeral* [*simp*] =  
*word-mod-def* [of  $\langle - \text{ numeral } a \rangle \langle \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for** *a b*

**lemmas** *mod-word-numeral-minus-numeral* [simp] =  
*word-mod-def* [of  $\langle \text{numeral } a \rangle \langle - \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$

**lemmas** *mod-word-minus-numeral-minus-numeral* [simp] =  
*word-mod-def* [of  $\langle - \text{ numeral } a \rangle \langle - \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$

**lemmas** *mod-word-minus-1-numeral* [simp] =  
*word-mod-def* [of  $\langle - 1 \rangle \langle \text{numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$

**lemmas** *mod-word-minus-1-minus-numeral* [simp] =  
*word-mod-def* [of  $\langle - 1 \rangle \langle - \text{ numeral } b \rangle$ , *simplified uint-bintrunc uint-bintrunc-neg*  
*unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]  
**for**  $a\ b$

**lemma** *signed-drop-bit-of-1* [simp]:  
 $\langle \text{signed-drop-bit } n\ (1 :: 'a::\text{len word}) \rangle = \text{of\_bool } (\text{LENGTH } ('a) = 1 \vee n = 0)$   
*<proof>*

**lemma** *take-bit-word-beyond-length-eq*:  
 $\langle \text{take-bit } n\ w = w \rangle$  **if**  $\langle \text{LENGTH } ('a) \leq n \rangle$  **for**  $w :: 'a::\text{len word}$   
*<proof>*

**lemmas** *word-div-no* [simp] = *word-div-def* [of numeral  $a$  numeral  $b$ ] **for**  $a\ b$   
**lemmas** *word-mod-no* [simp] = *word-mod-def* [of numeral  $a$  numeral  $b$ ] **for**  $a\ b$   
**lemmas** *word-less-no* [simp] = *word-less-def* [of numeral  $a$  numeral  $b$ ] **for**  $a\ b$   
**lemmas** *word-le-no* [simp] = *word-le-def* [of numeral  $a$  numeral  $b$ ] **for**  $a\ b$   
**lemmas** *word-sless-no* [simp] = *word-sless-eq* [of numeral  $a$  numeral  $b$ ] **for**  $a\ b$   
**lemmas** *word-sle-no* [simp] = *word-sle-eq* [of numeral  $a$  numeral  $b$ ] **for**  $a\ b$

**lemma** *size-0-same'*:  $\text{size } w = 0 \implies w = v$   
**for**  $v\ w :: 'a::\text{len word}$   
*<proof>*

**lemmas** *size-0-same* = *size-0-same'* [unfolded word-size]

**lemmas** *unat-eq-0* = *unat-0-iff*  
**lemmas** *unat-eq-zero* = *unat-0-iff*

**lemma** *mask-1*:  $\text{mask } 1 = 1$   
*<proof>*

**lemma** *mask-Suc-0*:  $\text{mask } (\text{Suc } 0) = 1$   
*<proof>*

**lemma** *bin-last-bintrunc*:  $\text{odd } (\text{take-bit } l\ n) \iff l > 0 \wedge \text{odd } n$   
*<proof>*



**lemma** *push-bit-word-beyond* [*simp*]:  
 $\langle \text{push-bit } n \ w = 0 \rangle$  **if**  $\langle \text{LENGTH}('a) \leq n \rangle$  **for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *drop-bit-word-beyond* [*simp*]:  
 $\langle \text{drop-bit } n \ w = 0 \rangle$  **if**  $\langle \text{LENGTH}('a) \leq n \rangle$  **for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *signed-drop-bit-beyond*:  
 $\langle \text{signed-drop-bit } n \ w = (\text{if bit } w \ (\text{LENGTH}('a) - \text{Suc } 0) \ \text{then } - 1 \ \text{else } 0) \rangle$   
**if**  $\langle \text{LENGTH}('a) \leq n \rangle$  **for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *take-bit-numeral-minus-numeral-word* [*simp*]:  
 $\langle \text{take-bit} \ (\text{numeral } m) \ (- \ \text{numeral } n :: 'a::\text{len word}) =$   
 $(\text{case take-bit-num} \ (\text{numeral } m) \ n \ \text{of None} \Rightarrow 0 \mid \text{Some } q \Rightarrow \text{take-bit} \ (\text{numeral}$   
 $m) \ (2 \wedge \text{numeral } m - \text{numeral } q)) \rangle$  (**is**  $\langle ?\text{lhs} = ?\text{rhs} \rangle$ )  
 $\langle \text{proof} \rangle$

**lemma** *of-nat-inverse*:  
 $\langle \text{word-of-nat } r = a \implies r < 2 \wedge \text{LENGTH}('a) \implies \text{unat } a = r \rangle$   
**for**  $a :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

## 106.16 Transferring goals from words to ints

**lemma** *word-ths*:  
**shows** *word-succ-p1*:  $\text{word-succ } a = a + 1$   
**and** *word-pred-m1*:  $\text{word-pred } a = a - 1$   
**and** *word-pred-succ*:  $\text{word-pred} \ (\text{word-succ } a) = a$   
**and** *word-succ-pred*:  $\text{word-succ} \ (\text{word-pred } a) = a$   
**and** *word-mult-succ*:  $\text{word-succ } a * b = b + a * b$   
 $\langle \text{proof} \rangle$

**lemma** *uint-cong*:  $x = y \implies \text{uint } x = \text{uint } y$   
 $\langle \text{proof} \rangle$

**lemma** *uint-word-ariths*:  
**fixes**  $a \ b :: 'a::\text{len word}$   
**shows**  $\text{uint} \ (a + b) = (\text{uint } a + \text{uint } b) \ \text{mod } 2 \wedge \text{LENGTH}('a::\text{len})$   
**and**  $\text{uint} \ (a - b) = (\text{uint } a - \text{uint } b) \ \text{mod } 2 \wedge \text{LENGTH}('a)$   
**and**  $\text{uint} \ (a * b) = \text{uint } a * \text{uint } b \ \text{mod } 2 \wedge \text{LENGTH}('a)$   
**and**  $\text{uint} \ (- a) = - \text{uint } a \ \text{mod } 2 \wedge \text{LENGTH}('a)$   
**and**  $\text{uint} \ (\text{word-succ } a) = (\text{uint } a + 1) \ \text{mod } 2 \wedge \text{LENGTH}('a)$   
**and**  $\text{uint} \ (\text{word-pred } a) = (\text{uint } a - 1) \ \text{mod } 2 \wedge \text{LENGTH}('a)$   
**and**  $\text{uint} \ (0 :: 'a \ \text{word}) = 0 \ \text{mod } 2 \wedge \text{LENGTH}('a)$   
**and**  $\text{uint} \ (1 :: 'a \ \text{word}) = 1 \ \text{mod } 2 \wedge \text{LENGTH}('a)$   
 $\langle \text{proof} \rangle$

**lemma** *uint-word-arith-bintrs*:

**fixes**  $a\ b :: 'a::len\ word$

**shows**  $uint\ (a + b) = take-bit\ (LENGTH('a))\ (uint\ a + uint\ b)$

**and**  $uint\ (a - b) = take-bit\ (LENGTH('a))\ (uint\ a - uint\ b)$

**and**  $uint\ (a * b) = take-bit\ (LENGTH('a))\ (uint\ a * uint\ b)$

**and**  $uint\ (- a) = take-bit\ (LENGTH('a))\ (- uint\ a)$

**and**  $uint\ (word-succ\ a) = take-bit\ (LENGTH('a))\ (uint\ a + 1)$

**and**  $uint\ (word-pred\ a) = take-bit\ (LENGTH('a))\ (uint\ a - 1)$

**and**  $uint\ (0 :: 'a\ word) = take-bit\ (LENGTH('a))\ 0$

**and**  $uint\ (1 :: 'a\ word) = take-bit\ (LENGTH('a))\ 1$

*<proof>*

**context**

**fixes**  $a\ b :: 'a::len\ word$

**begin**

**lemma** *sint-word-add*:  $sint\ (a + b) = signed-take-bit\ (LENGTH('a) - 1)\ (sint\ a + sint\ b)$

*<proof>*

**lemma** *sint-word-diff*:  $sint\ (a - b) = signed-take-bit\ (LENGTH('a) - 1)\ (sint\ a - sint\ b)$

*<proof>*

**lemma** *sint-word-mult*:  $sint\ (a * b) = signed-take-bit\ (LENGTH('a) - 1)\ (sint\ a * sint\ b)$

*<proof>*

**lemma** *sint-word-minus*:  $sint\ (- a) = signed-take-bit\ (LENGTH('a) - 1)\ (- sint\ a)$

*<proof>*

**lemma** *sint-word-succ*:  $sint\ (word-succ\ a) = signed-take-bit\ (LENGTH('a) - 1)\ (sint\ a + 1)$

*<proof>*

**lemma** *sint-word-pred*:  $sint\ (word-pred\ a) = signed-take-bit\ (LENGTH('a) - 1)\ (sint\ a - 1)$

*<proof>*

**lemma** *sint-word-01*:

$sint\ (0 :: 'a\ word) = signed-take-bit\ (LENGTH('a) - 1)\ 0$

$sint\ (1 :: 'a\ word) = signed-take-bit\ (LENGTH('a) - 1)\ 1$

*<proof>*

**end**

**lemmas** *sint-word-ariths* =  
*sint-word-add sint-word-diff sint-word-mult sint-word-minus*  
*sint-word-succ sint-word-pred sint-word-01*

**lemma** *word-pred-0-n1*:  $\text{word-pred } 0 = \text{word-of-int } (- 1)$   
 ⟨*proof*⟩

**lemma** *succ-pred-no* [*simp*]:  
 $\text{word-succ } (\text{numeral } w) = \text{numeral } w + 1$   
 $\text{word-pred } (\text{numeral } w) = \text{numeral } w - 1$   
 $\text{word-succ } (- \text{numeral } w) = - \text{numeral } w + 1$   
 $\text{word-pred } (- \text{numeral } w) = - \text{numeral } w - 1$   
 ⟨*proof*⟩

**lemma** *word-sp-01* [*simp*]:  
 $\text{word-succ } (- 1) = 0 \wedge \text{word-succ } 0 = 1 \wedge \text{word-pred } 0 = - 1 \wedge \text{word-pred } 1 = 0$   
 ⟨*proof*⟩

**lemma** *word-of-int-Ex*:  $\exists y. x = \text{word-of-int } y$   
 ⟨*proof*⟩

## 106.17 Order on fixed-length words

**lift-definition** *udvd* ::  $\langle 'a::\text{len word} \Rightarrow 'a::\text{len word} \Rightarrow \text{bool} \rangle$  (**infixl**  $\langle \text{udvd} \rangle$  50)  
 is  $\langle \lambda k l. \text{take-bit } \text{LENGTH}('a) k \text{ dvd take-bit } \text{LENGTH}('a) l \rangle$  ⟨*proof*⟩

**lemma** *udvd-iff-dvd*:  
 $\langle x \text{ udvd } y \longleftrightarrow \text{unat } x \text{ dvd unat } y \rangle$   
 ⟨*proof*⟩

**lemma** *udvd-iff-dvd-int*:  
 $\langle v \text{ udvd } w \longleftrightarrow \text{uint } v \text{ dvd uint } w \rangle$   
 ⟨*proof*⟩

**lemma** *udvdI* [*intro*]:  
 $\langle v \text{ udvd } w \rangle$  **if**  $\langle \text{unat } w = \text{unat } v * \text{unat } u \rangle$   
 ⟨*proof*⟩

**lemma** *udvdE* [*elim*]:  
**fixes**  $v w$  ::  $\langle 'a::\text{len word} \rangle$   
**assumes**  $\langle v \text{ udvd } w \rangle$   
**obtains**  $u$  ::  $\langle 'a \text{ word} \rangle$  **where**  $\langle \text{unat } w = \text{unat } v * \text{unat } u \rangle$   
 ⟨*proof*⟩

**lemma** *udvd-imp-mod-eq-0*:  
 $\langle w \text{ mod } v = 0 \rangle$  **if**  $\langle v \text{ udvd } w \rangle$   
 ⟨*proof*⟩

**lemma** *mod-eq-0-imp-udvd* [*intro?*]:

$\langle v \text{ udvd } w \rangle$  **if**  $\langle w \text{ mod } v = 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *udvd-imp-dvd*:

$\langle v \text{ dvd } w \rangle$  **if**  $\langle v \text{ udvd } w \rangle$  **for**  $v \ w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *exp-dvd-iff-exp-udvd*:

$\langle 2 \wedge^n \text{ dvd } w \rangle \longleftrightarrow \langle 2 \wedge^n \text{ udvd } w \rangle$  **for**  $v \ w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *udvd-nat-alt*:

$\langle a \text{ udvd } b \rangle \longleftrightarrow (\exists n. \text{ unat } b = n * \text{ unat } a)$   
 $\langle \text{proof} \rangle$

**lemma** *udvd-unfold-int*:

$\langle a \text{ udvd } b \rangle \longleftrightarrow (\exists n \geq 0. \text{ uint } b = n * \text{ uint } a)$   
 $\langle \text{proof} \rangle$

**lemma** *unat-minus-one*:

$\langle \text{unat } (w - 1) = \text{unat } w - 1 \rangle$  **if**  $\langle w \neq 0 \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *measure-unat*:  $p \neq 0 \implies \text{unat } (p - 1) < \text{unat } p$

$\langle \text{proof} \rangle$

**lemmas** *uint-add-ge0* [simp] = *add-nonneg-nonneg* [OF *uint-ge-0 uint-ge-0*]

**lemmas** *uint-mult-ge0* [simp] = *mult-nonneg-nonneg* [OF *uint-ge-0 uint-ge-0*]

**lemma** *uint-sub-lt2p* [simp]:  $\text{uint } x - \text{uint } y < 2 \wedge \text{LENGTH}('a)$

**for**  $x :: 'a::\text{len word}$  **and**  $y :: 'b::\text{len word}$

$\langle \text{proof} \rangle$

## 106.18 Conditions for the addition (etc) of two words to overflow

**lemma** *uint-add-lem*:

$(\text{uint } x + \text{uint } y < 2 \wedge \text{LENGTH}('a)) =$

$(\text{uint } (x + y) = \text{uint } x + \text{uint } y)$

**for**  $x \ y :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

**lemma** *uint-mult-lem*:

$(\text{uint } x * \text{uint } y < 2 \wedge \text{LENGTH}('a)) =$

$(\text{uint } (x * y) = \text{uint } x * \text{uint } y)$

**for**  $x \ y :: 'a::\text{len word}$

$\langle \text{proof} \rangle$

**lemma** *uint-sub-lem*:  $\text{uint } x \geq \text{uint } y \longleftrightarrow \text{uint } (x - y) = \text{uint } x - \text{uint } y$

*<proof>*

**lemma** *uint-add-le*:  $\text{uint } (x + y) \leq \text{uint } x + \text{uint } y$   
*<proof>*

**lemma** *uint-sub-ge*:  $\text{uint } (x - y) \geq \text{uint } x - \text{uint } y$   
*<proof>*

**lemma** *int-mod-ge*:  $\langle a \leq a \text{ mod } n \rangle$  **if**  $\langle a < n \rangle \langle 0 < n \rangle$   
**for**  $a \ n :: \text{int}$   
*<proof>*

**lemma** *mod-add-if-z*:  
 $\llbracket x < z; y < z; 0 \leq y; 0 \leq x; 0 \leq z \rrbracket \implies$   
 $(x + y) \text{ mod } z = (\text{if } x + y < z \text{ then } x + y \text{ else } x + y - z)$   
**for**  $x \ y \ z :: \text{int}$   
*<proof>*

**lemma** *uint-plus-if'*:  
 $\text{uint } (a + b) =$   
 $(\text{if } \text{uint } a + \text{uint } b < 2 \wedge \text{LENGTH}('a) \text{ then } \text{uint } a + \text{uint } b$   
 $\text{ else } \text{uint } a + \text{uint } b - 2 \wedge \text{LENGTH}('a))$   
**for**  $a \ b :: 'a::\text{len word}$   
*<proof>*

**lemma** *mod-sub-if-z*:  
 $\llbracket x < z; y < z; 0 \leq y; 0 \leq x; 0 \leq z \rrbracket \implies$   
 $(x - y) \text{ mod } z = (\text{if } y \leq x \text{ then } x - y \text{ else } x - y + z)$   
**for**  $x \ y \ z :: \text{int}$   
*<proof>*

**lemma** *uint-sub-if'*:  
 $\text{uint } (a - b) =$   
 $(\text{if } \text{uint } b \leq \text{uint } a \text{ then } \text{uint } a - \text{uint } b$   
 $\text{ else } \text{uint } a - \text{uint } b + 2 \wedge \text{LENGTH}('a))$   
**for**  $a \ b :: 'a::\text{len word}$   
*<proof>*

**lemma** *word-of-int-inverse*:  
 $\text{word-of-int } r = a \implies 0 \leq r \implies r < 2 \wedge \text{LENGTH}('a) \implies \text{uint } a = r$   
**for**  $a :: 'a::\text{len word}$   
*<proof>*

**lemma** *unat-split*:  $P (\text{unat } x) \longleftrightarrow (\forall n. \text{of-nat } n = x \wedge n < 2 \wedge \text{LENGTH}('a) \longrightarrow P \ n)$   
**for**  $x :: 'a::\text{len word}$   
*<proof>*

**lemma** *unat-split-asm*:  $P (\text{unat } x) \longleftrightarrow (\exists n. \text{of-nat } n = x \wedge n < 2 \wedge \text{LENGTH}('a))$

$\wedge \neg P n$   
**for**  $x :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *un-ui-le*:  
 ⟨ $unat\ a \leq unat\ b \longleftrightarrow uint\ a \leq uint\ b$ ⟩  
 ⟨proof⟩

**lemma** *unat-plus-if'*:  
 ⟨ $unat\ (a + b) =$   
   ⟨if  $unat\ a + unat\ b < 2^{\wedge} LENGTH('a)$   
   then  $unat\ a + unat\ b$   
   else  $unat\ a + unat\ b - 2^{\wedge} LENGTH('a)$ ⟩ **for**  $a\ b :: 'a::len\ word$ ⟩  
 ⟨proof⟩

**lemma** *unat-sub-if-size*:  
 $unat\ (x - y) =$   
   ⟨if  $unat\ y \leq unat\ x$   
   then  $unat\ x - unat\ y$   
   else  $unat\ x + 2^{\wedge} size\ x - unat\ y$ ⟩  
 ⟨proof⟩

**lemmas** *unat-sub-if' = unat-sub-if-size [unfolded word-size]*

**lemma** *uint-split*:  
 $P\ (uint\ x) = (\forall i. word-of-int\ i = x \wedge 0 \leq i \wedge i < 2^{\wedge} LENGTH('a) \longrightarrow P\ i)$   
**for**  $x :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *uint-split-asm*:  
 $P\ (uint\ x) = (\exists i. word-of-int\ i = x \wedge 0 \leq i \wedge i < 2^{\wedge} LENGTH('a) \wedge \neg P\ i)$   
**for**  $x :: 'a::len\ word$   
 ⟨proof⟩

## 106.19 Some proof tool support

**lemma** *power-False-cong*:  $False \implies a^{\wedge} b = c^{\wedge} d$   
 ⟨proof⟩

**lemmas** *unat-splits = unat-split unat-split-asm*

**lemmas** *unat-arith-simps =*  
*word-le-nat-alt word-less-nat-alt*  
*word-unat-eq-iff*  
*unat-sub-if' unat-plus-if' unat-div unat-mod*

**lemmas** *uint-splits = uint-split uint-split-asm*

**lemmas** *uint-arith-simps =*

*word-le-def word-less-alt*  
*word-uint-eq-iff*  
*uint-sub-if' uint-plus-if'*

— *unat-arith-tac*: tactic to reduce word arithmetic to *nat*, try to solve via *arith*  
 ⟨*ML*⟩

## 106.20 More on overflows and monotonicity

**lemma** *no-plus-overflow-uint-size*:  $x \leq x + y \longleftrightarrow \text{uint } x + \text{uint } y < 2^{\wedge \text{size } x}$   
 for  $x \ y :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemmas** *no-olen-add* = *no-plus-overflow-uint-size* [*unfolded word-size*]

**lemma** *no-olen-sub*:  $x \geq x - y \longleftrightarrow \text{uint } y \leq \text{uint } x$   
 for  $x \ y :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemma** *no-olen-add'*:  $x \leq y + x \longleftrightarrow \text{uint } y + \text{uint } x < 2^{\wedge \text{LENGTH}('a)}$   
 for  $x \ y :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemmas** *olen-add-equiv* = *trans* [*OF no-olen-add no-olen-add'* [*symmetric*]]

**lemmas** *uint-plus-simple-iff* = *trans* [*OF no-olen-add uint-add-lem*]

**lemmas** *uint-plus-simple* = *uint-plus-simple-iff* [*THEN iffD1*]

**lemmas** *uint-minus-simple-iff* = *trans* [*OF no-olen-sub uint-sub-lem*]

**lemmas** *uint-minus-simple-alt* = *uint-sub-lem* [*folded word-le-def*]

**lemmas** *word-sub-le-iff* = *no-olen-sub* [*folded word-le-def*]

**lemmas** *word-sub-le* = *word-sub-le-iff* [*THEN iffD2*]

**lemma** *word-less-sub1*:  $x \neq 0 \implies 1 < x \longleftrightarrow 0 < x - 1$   
 for  $x :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemma** *word-le-sub1*:  $x \neq 0 \implies 1 \leq x \longleftrightarrow 0 \leq x - 1$   
 for  $x :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemma** *sub-wrap-lt*:  $x < x - z \longleftrightarrow x < z$   
 for  $x \ z :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemma** *sub-wrap*:  $x \leq x - z \longleftrightarrow z = 0 \vee x < z$   
 for  $x \ z :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemma** *plus-minus-not-NULL-ab*:  $x \leq ab - c \implies c \leq ab \implies c \neq 0 \implies x + c$

$\neq 0$

**for**  $x\ ab\ c :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *plus-minus-no-overflow-ab*:  $x \leq ab - c \implies c \leq ab \implies x \leq x + c$   
**for**  $x\ ab\ c :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *le-minus'*:  $a + c \leq b \implies a \leq a + c \implies c \leq b - a$   
**for**  $a\ b\ c :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *le-plus'*:  $a \leq b \implies c \leq b - a \implies a + c \leq b$   
**for**  $a\ b\ c :: 'a::len\ word$   
 ⟨proof⟩

**lemmas** *le-plus = le-plus'* [rotated]

**lemmas** *le-minus = leD* [THEN *thin-rl*, THEN *le-minus'*]

**lemma** *word-plus-mono-right*:  $y \leq z \implies x \leq x + z \implies x + y \leq x + z$   
**for**  $x\ y\ z :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *word-less-minus-cancel*:  $y - x < z - x \implies x \leq z \implies y < z$   
**for**  $x\ y\ z :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *word-less-minus-mono-left*:  $y < z \implies x \leq y \implies y - x < z - x$   
**for**  $x\ y\ z :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *word-less-minus-mono*:  $a < c \implies d < b \implies a - b < a \implies c - d < c$   
 $\implies a - b < c - d$   
**for**  $a\ b\ c\ d :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *word-le-minus-cancel*:  $y - x \leq z - x \implies x \leq z \implies y \leq z$   
**for**  $x\ y\ z :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *word-le-minus-mono-left*:  $y \leq z \implies x \leq y \implies y - x \leq z - x$   
**for**  $x\ y\ z :: 'a::len\ word$   
 ⟨proof⟩

**lemma** *word-le-minus-mono*:  
 $a \leq c \implies d \leq b \implies a - b \leq a \implies c - d \leq c \implies a - b \leq c - d$   
**for**  $a\ b\ c\ d :: 'a::len\ word$   
 ⟨proof⟩



**lemma** *plus-le-left-cancel-wrap*:  $x + y' < x \implies x + y < x \implies x + y' < x + y$   
 $\iff y' < y$   
**for**  $x\ y\ y' :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *plus-le-left-cancel-nowrap*:  $x \leq x + y' \implies x \leq x + y \implies x + y' < x + y$   
 $\iff y' < y$   
**for**  $x\ y\ y' :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-plus-mono-right2*:  $a \leq a + b \implies c \leq b \implies a \leq a + c$   
**for**  $a\ b\ c :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-less-add-right*:  $x < y - z \implies z \leq y \implies x + z < y$   
**for**  $x\ y\ z :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-less-sub-right*:  $x < y + z \implies y \leq x \implies x - y < z$   
**for**  $x\ y\ z :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-le-plus-either*:  $x \leq y \vee x \leq z \implies y \leq y + z \implies x \leq y + z$   
**for**  $x\ y\ z :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-less-nowrapI*:  $x < z - k \implies k \leq z \implies 0 < k \implies x < x + k$   
**for**  $x\ z\ k :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *inc-le*:  $i < m \implies i + 1 \leq m$   
**for**  $i\ m :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *inc-i*:  $1 \leq i \implies i < m \implies 1 \leq i + 1 \wedge i + 1 \leq m$   
**for**  $i\ m :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *udvd-incr-lem*:  
 $\llbracket up < uq; up = ua + n * \text{uint } K; uq = ua + n' * \text{uint } K \rrbracket$   
 $\implies up + \text{uint } K \leq uq$   
 ⟨proof⟩

**lemma** *udvd-incr'*:  
 $p < q \implies \text{uint } p = ua + n * \text{uint } K \implies$   
 $\text{uint } q = ua + n' * \text{uint } K \implies p + K \leq q$   
 ⟨proof⟩

**lemma** *udvd-decr'*:

**assumes**  $p < q$  *uint*  $p = ua + n * \text{uint } K$  *uint*  $q = ua + n' * \text{uint } K$

**shows** *uint*  $q = ua + n' * \text{uint } K \implies p \leq q - K$

*<proof>*

**lemmas** *udvd-incr-lem0* = *udvd-incr-lem* [**where**  $ua=0$ , *unfolded add-0-left*]

**lemmas** *udvd-incr0* = *udvd-incr'* [**where**  $ua=0$ , *unfolded add-0-left*]

**lemmas** *udvd-decr0* = *udvd-decr'* [**where**  $ua=0$ , *unfolded add-0-left*]

**lemma** *udvd-minus-le'*:  $xy < k \implies z \text{ udvd } xy \implies z \text{ udvd } k \implies xy \leq k - z$

*<proof>*

**lemma** *udvd-incr2-K*:

$p < a + s \implies a \leq a + s \implies K \text{ udvd } s \implies K \text{ udvd } p - a \implies a \leq p \implies$

$0 < K \implies p \leq p + K \wedge p + K \leq a + s$

*<proof>*

## 106.21 Arithmetic type class instantiations

**lemmas** *word-le-0-iff* [*simp*] =

*word-zero-le* [*THEN leD*, *THEN antisym-conv1*]

**lemma** *word-of-int-nat*:  $0 \leq x \implies \text{word-of-int } x = \text{of-nat } (\text{nat } x)$

*<proof>*

note that *iszero-def* is only for class *comm-semiring-1-cancel*, which requires word length  $\geq 1$ , ie *'a::len word*

**lemma** *iszero-word-no* [*simp*]:

*iszero* (*numeral bin* :: *'a::len word*) =

*iszero* (*take-bit LENGTH('a)* (*numeral bin* :: *int*))

*<proof>*

Use *iszero* to simplify equalities between word numerals.

**lemmas** *word-eq-numeral-iff-iszero* [*simp*] =

*eq-numeral-iff-iszero* [**where**  $'a='a::len \text{ word}$ ]

**lemma** *word-less-eq-imp-half-less-eq*:

$\langle v \text{ div } 2 \leq w \text{ div } 2 \rangle$  **if**  $\langle v \leq w \rangle$  **for**  $v w :: \langle 'a::len \text{ word} \rangle$

*<proof>*

**lemma** *word-half-less-imp-less-eq*:

$\langle v \leq w \rangle$  **if**  $\langle v \text{ div } 2 < w \text{ div } 2 \rangle$  **for**  $v w :: \langle 'a::len \text{ word} \rangle$

*<proof>*

## 106.22 Word and nat

**lemma** *word-nchotomy*:  $\forall w :: 'a::len \text{ word}. \exists n. w = \text{of-nat } n \wedge n < 2^{\wedge \text{LENGTH}('a)}$

*<proof>*

**lemma** *of-nat-eq*:  $of\text{-}nat\ n = w \longleftrightarrow (\exists q. n = unat\ w + q * 2 \wedge LENGTH('a))$   
**for**  $w :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *of-nat-eq-size*:  $of\text{-}nat\ n = w \longleftrightarrow (\exists q. n = unat\ w + q * 2 \wedge size\ w)$   
 ⟨*proof*⟩

**lemma** *of-nat-0*:  $of\text{-}nat\ m = (0 :: 'a::len\ word) \longleftrightarrow (\exists q. m = q * 2 \wedge LENGTH('a))$   
 ⟨*proof*⟩

**lemma** *of-nat-2p [simp]*:  $of\text{-}nat\ (2 \wedge LENGTH('a)) = (0 :: 'a::len\ word)$   
 ⟨*proof*⟩

**lemma** *of-nat-gt-0*:  $of\text{-}nat\ k \neq 0 \implies 0 < k$   
 ⟨*proof*⟩

**lemma** *of-nat-neq-0*:  $0 < k \implies k < 2 \wedge LENGTH('a::len) \implies of\text{-}nat\ k \neq (0 :: 'a\ word)$   
 ⟨*proof*⟩

**lemma** *Abs-fnat-hom-add*:  $of\text{-}nat\ a + of\text{-}nat\ b = of\text{-}nat\ (a + b)$   
 ⟨*proof*⟩

**lemma** *Abs-fnat-hom-mult*:  $of\text{-}nat\ a * of\text{-}nat\ b = (of\text{-}nat\ (a * b) :: 'a::len\ word)$   
 ⟨*proof*⟩

**lemma** *Abs-fnat-hom-Suc*:  $word\text{-}succ\ (of\text{-}nat\ a) = of\text{-}nat\ (Suc\ a)$   
 ⟨*proof*⟩

**lemma** *Abs-fnat-hom-0*:  $(0 :: 'a::len\ word) = of\text{-}nat\ 0$   
 ⟨*proof*⟩

**lemma** *Abs-fnat-hom-1*:  $(1 :: 'a::len\ word) = of\text{-}nat\ (Suc\ 0)$   
 ⟨*proof*⟩

**lemmas** *Abs-fnat-homs =*  
*Abs-fnat-hom-add Abs-fnat-hom-mult Abs-fnat-hom-Suc*  
*Abs-fnat-hom-0 Abs-fnat-hom-1*

**lemma** *word-arith-nat-add*:  $a + b = of\text{-}nat\ (unat\ a + unat\ b)$   
 ⟨*proof*⟩

**lemma** *word-arith-nat-mult*:  $a * b = of\text{-}nat\ (unat\ a * unat\ b)$   
 ⟨*proof*⟩

**lemma** *word-arith-nat-Suc*:  $word\text{-}succ\ a = of\text{-}nat\ (Suc\ (unat\ a))$   
 ⟨*proof*⟩

**lemma** *word-arith-nat-div*:  $a\ div\ b = of\text{-}nat\ (unat\ a\ div\ unat\ b)$

⟨proof⟩

**lemma** *word-arith-nat-mod*:  $a \bmod b = \text{of-nat } (\text{unat } a \bmod \text{unat } b)$   
 ⟨proof⟩

**lemmas** *word-arith-nat-defs* =  
*word-arith-nat-add word-arith-nat-mult*  
*word-arith-nat-Suc Abs-fnat-hom-0*  
*Abs-fnat-hom-1 word-arith-nat-div*  
*word-arith-nat-mod*

**lemma** *unat-of-nat*:  
 ⟨ $\text{unat } (\text{word-of-nat } x :: 'a::\text{len word}) = x \bmod 2^{\text{LENGTH}('a)}$ ⟩  
 ⟨proof⟩

**lemma** *unat-cong*:  $x = y \implies \text{unat } x = \text{unat } y$   
 ⟨proof⟩

**lemmas** *unat-word-ariths* = *word-arith-nat-defs*  
 [THEN trans [OF unat-cong unat-of-nat]]

**lemmas** *word-sub-less-iff* = *word-sub-le-iff*  
 [unfolded linorder-not-less [symmetric] Not-eq-iff]

**lemma** *unat-add-lem*:  
 $\text{unat } x + \text{unat } y < 2^{\text{LENGTH}('a)} \iff \text{unat } (x + y) = \text{unat } x + \text{unat } y$   
**for**  $x \ y :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *unat-mult-lem*:  
 $\text{unat } x * \text{unat } y < 2^{\text{LENGTH}('a)} \iff \text{unat } (x * y) = \text{unat } x * \text{unat } y$   
**for**  $x \ y :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *le-no-overflow*:  $x \leq b \implies a \leq a + b \implies x \leq a + b$   
**for**  $a \ b \ x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *uint-div*:  
 ⟨ $\text{uint } (x \text{ div } y) = \text{uint } x \text{ div } \text{uint } y$ ⟩  
 ⟨proof⟩

**lemma** *uint-mod*:  
 ⟨ $\text{uint } (x \bmod y) = \text{uint } x \bmod \text{uint } y$ ⟩  
 ⟨proof⟩

**lemma** *no-plus-overflow-unat-size*:  $x \leq x + y \iff \text{unat } x + \text{unat } y < 2^{\text{size } x}$   
**for**  $x \ y :: 'a::\text{len word}$   
 ⟨proof⟩

**lemmas** *no-olen-add-nat* =  
*no-plus-overflow-unat-size* [*unfolded word-size*]

**lemmas** *unat-plus-simple* =  
*trans* [*OF no-olen-add-nat unat-add-lem*]

**lemma** *word-div-mult*:  $\llbracket 0 < y; \text{unat } x * \text{unat } y < 2 \wedge \text{LENGTH}('a) \rrbracket \implies x * y$   
*div y = x*  
**for**  $x y :: 'a::\text{len word}$   
*<proof>*

**lemma** *div-lt'*:  $i \leq k \text{ div } x \implies \text{unat } i * \text{unat } x < 2 \wedge \text{LENGTH}('a)$   
**for**  $i k x :: 'a::\text{len word}$   
*<proof>*

**lemmas** *div-lt''* = *order-less-imp-le* [*THEN div-lt'*]

**lemma** *div-lt-mult*:  $\llbracket i < k \text{ div } x; 0 < x \rrbracket \implies i * x < k$   
**for**  $i k x :: 'a::\text{len word}$   
*<proof>*

**lemma** *div-le-mult*:  $\llbracket i \leq k \text{ div } x; 0 < x \rrbracket \implies i * x \leq k$   
**for**  $i k x :: 'a::\text{len word}$   
*<proof>*

**lemma** *div-lt-uint'*:  $i \leq k \text{ div } x \implies \text{uint } i * \text{uint } x < 2 \wedge \text{LENGTH}('a)$   
**for**  $i k x :: 'a::\text{len word}$   
*<proof>*

**lemmas** *div-lt-uint''* = *order-less-imp-le* [*THEN div-lt-uint'*]

**lemma** *word-le-exists'*:  $x \leq y \implies \exists z. y = x + z \wedge \text{uint } x + \text{uint } z < 2 \wedge$   
 $\text{LENGTH}('a)$   
**for**  $x y z :: 'a::\text{len word}$   
*<proof>*

**lemmas** *plus-minus-not-NULL* = *order-less-imp-le* [*THEN plus-minus-not-NULL-ab*]

**lemmas** *plus-minus-no-overflow* =  
*order-less-imp-le* [*THEN plus-minus-no-overflow-ab*]

**lemmas** *mcs* = *word-less-minus-cancel word-less-minus-mono-left*  
*word-le-minus-cancel word-le-minus-mono-left*

**lemmas** *word-l-diffs* = *mcs* [**where**  $y = w + x$ , *unfolded add-diff-cancel*] **for**  $w x$   
**lemmas** *word-diff-ls* = *mcs* [**where**  $z = w + x$ , *unfolded add-diff-cancel*] **for**  $w x$   
**lemmas** *word-plus-mcs* = *word-diff-ls* [**where**  $y = v + x$ , *unfolded add-diff-cancel*]  
**for**  $v x$

**lemma** *le-unat-woi*:

$\langle y \leq \text{unat } z \implies \text{unat } (\text{word-of-nat } y :: 'a \text{ word}) = y \rangle$   
**for**  $z :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemmas** *thd = times-div-less-eq-dividend*

**lemmas** *uno-simps [THEN le-unat-woi] = mod-le-divisor div-le-dividend*

**lemma** *word-mod-div-equality*:  $(n \text{ div } b) * b + (n \text{ mod } b) = n$   
**for**  $n b :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *word-div-mult-le*:  $a \text{ div } b * b \leq a$   
**for**  $a b :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *word-mod-less-divisor*:  $0 < n \implies m \text{ mod } n < n$   
**for**  $m n :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *word-of-int-power-hom*:  $\text{word-of-int } a \wedge n = (\text{word-of-int } (a \wedge n) :: 'a::\text{len word})$   
 $\langle \text{proof} \rangle$

**lemma** *word-arith-power-alt*:  $a \wedge n = (\text{word-of-int } (\text{uint } a \wedge n) :: 'a::\text{len word})$   
 $\langle \text{proof} \rangle$

**lemma** *unatSuc*:  $1 + n \neq 0 \implies \text{unat } (1 + n) = \text{Suc } (\text{unat } n)$   
**for**  $n :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

### 106.23 Cardinality, finiteness of set of words

**lemma** *inj-on-word-of-int*:  $\langle \text{inj-on } (\text{word-of-int } :: \text{int} \Rightarrow 'a \text{ word}) \{0..<2 \wedge \text{LENGTH}('a::\text{len})\} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *range-uint*:  $\langle \text{range } (\text{uint } :: 'a \text{ word} \Rightarrow \text{int}) = \{0..<2 \wedge \text{LENGTH}('a::\text{len})\} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *UNIV-eq*:  $\langle (\text{UNIV } :: 'a \text{ word set}) = \text{word-of-int } \{0..<2 \wedge \text{LENGTH}('a::\text{len})\} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *card-word*:  $\text{CARD}('a \text{ word}) = 2 \wedge \text{LENGTH}('a::\text{len})$   
 $\langle \text{proof} \rangle$

**lemma** *card-word-size*:  $\text{CARD}('a \text{ word}) = 2 \wedge \text{size } x$   
**for**  $x :: 'a::\text{len word}$

*<proof>*

**end**

**instance** *word* :: (*len*) *finite*

*<proof>*

## 106.24 Bitwise Operations on Words

**context**

**includes** *bit-operations-syntax*

**begin**

**lemma** *word-wi-log-defs*:

*NOT* (*word-of-int* *a*) = *word-of-int* (*NOT* *a*)  
*word-of-int* *a* *AND* *word-of-int* *b* = *word-of-int* (*a* *AND* *b*)  
*word-of-int* *a* *OR* *word-of-int* *b* = *word-of-int* (*a* *OR* *b*)  
*word-of-int* *a* *XOR* *word-of-int* *b* = *word-of-int* (*a* *XOR* *b*)  
*<proof>*

**lemma** *word-no-log-defs* [*simp*]:

*NOT* (*numeral* *a*) = *word-of-int* (*NOT* (*numeral* *a*))  
*NOT* ( $\neg$  *numeral* *a*) = *word-of-int* (*NOT* ( $\neg$  *numeral* *a*))  
*numeral* *a* *AND* *numeral* *b* = *word-of-int* (*numeral* *a* *AND* *numeral* *b*)  
*numeral* *a* *AND*  $\neg$  *numeral* *b* = *word-of-int* (*numeral* *a* *AND*  $\neg$  *numeral* *b*)  
 $\neg$  *numeral* *a* *AND* *numeral* *b* = *word-of-int* ( $\neg$  *numeral* *a* *AND* *numeral* *b*)  
 $\neg$  *numeral* *a* *AND*  $\neg$  *numeral* *b* = *word-of-int* ( $\neg$  *numeral* *a* *AND*  $\neg$  *numeral* *b*)  
*numeral* *a* *OR* *numeral* *b* = *word-of-int* (*numeral* *a* *OR* *numeral* *b*)  
*numeral* *a* *OR*  $\neg$  *numeral* *b* = *word-of-int* (*numeral* *a* *OR*  $\neg$  *numeral* *b*)  
 $\neg$  *numeral* *a* *OR* *numeral* *b* = *word-of-int* ( $\neg$  *numeral* *a* *OR* *numeral* *b*)  
 $\neg$  *numeral* *a* *OR*  $\neg$  *numeral* *b* = *word-of-int* ( $\neg$  *numeral* *a* *OR*  $\neg$  *numeral* *b*)  
*numeral* *a* *XOR* *numeral* *b* = *word-of-int* (*numeral* *a* *XOR* *numeral* *b*)  
*numeral* *a* *XOR*  $\neg$  *numeral* *b* = *word-of-int* (*numeral* *a* *XOR*  $\neg$  *numeral* *b*)  
 $\neg$  *numeral* *a* *XOR* *numeral* *b* = *word-of-int* ( $\neg$  *numeral* *a* *XOR* *numeral* *b*)  
 $\neg$  *numeral* *a* *XOR*  $\neg$  *numeral* *b* = *word-of-int* ( $\neg$  *numeral* *a* *XOR*  $\neg$  *numeral* *b*)  
*<proof>*

Special cases for when one of the arguments equals 1.

**lemma** *word-bitwise-1-simps* [*simp*]:

*NOT* ( $1::'a::len$  *word*) =  $\neg 2$   
 $1$  *AND* *numeral* *b* = *word-of-int* ( $1$  *AND* *numeral* *b*)  
 $1$  *AND*  $\neg$  *numeral* *b* = *word-of-int* ( $1$  *AND*  $\neg$  *numeral* *b*)  
*numeral* *a* *AND*  $1$  = *word-of-int* (*numeral* *a* *AND*  $1$ )  
 $\neg$  *numeral* *a* *AND*  $1$  = *word-of-int* ( $\neg$  *numeral* *a* *AND*  $1$ )  
 $1$  *OR* *numeral* *b* = *word-of-int* ( $1$  *OR* *numeral* *b*)  
 $1$  *OR*  $\neg$  *numeral* *b* = *word-of-int* ( $1$  *OR*  $\neg$  *numeral* *b*)  
*numeral* *a* *OR*  $1$  = *word-of-int* (*numeral* *a* *OR*  $1$ )  
 $\neg$  *numeral* *a* *OR*  $1$  = *word-of-int* ( $\neg$  *numeral* *a* *OR*  $1$ )  
 $1$  *XOR* *numeral* *b* = *word-of-int* ( $1$  *XOR* *numeral* *b*)  
 $1$  *XOR*  $\neg$  *numeral* *b* = *word-of-int* ( $1$  *XOR*  $\neg$  *numeral* *b*)

$\text{numeral } a \text{ XOR } 1 = \text{word-of-int } (\text{numeral } a \text{ XOR } 1)$   
 $-\text{ numeral } a \text{ XOR } 1 = \text{word-of-int } (-\text{ numeral } a \text{ XOR } 1)$   
 ⟨proof⟩

Special cases for when one of the arguments equals -1.

**lemma** *word-bitwise-m1-simps* [simp]:

$\text{NOT } (-1::'a::\text{len word}) = 0$   
 $(-1::'a::\text{len word}) \text{ AND } x = x$   
 $x \text{ AND } (-1::'a::\text{len word}) = x$   
 $(-1::'a::\text{len word}) \text{ OR } x = -1$   
 $x \text{ OR } (-1::'a::\text{len word}) = -1$   
 $(-1::'a::\text{len word}) \text{ XOR } x = \text{NOT } x$   
 $x \text{ XOR } (-1::'a::\text{len word}) = \text{NOT } x$   
 ⟨proof⟩

**lemma** *word-of-int-not-numeral-eq* [simp]:

$\langle \text{word-of-int } (\text{NOT } (\text{numeral bin})) :: 'a::\text{len word} \rangle = -\text{ numeral bin } - 1$   
 ⟨proof⟩

**lemma** *uint-and*:

$\langle \text{uint } (x \text{ AND } y) \rangle = \text{uint } x \text{ AND } \text{uint } y$   
 ⟨proof⟩

**lemma** *uint-or*:

$\langle \text{uint } (x \text{ OR } y) \rangle = \text{uint } x \text{ OR } \text{uint } y$   
 ⟨proof⟩

**lemma** *uint-xor*:

$\langle \text{uint } (x \text{ XOR } y) \rangle = \text{uint } x \text{ XOR } \text{uint } y$   
 ⟨proof⟩

**lemmas** *bwsimps* =

*wi-hom-add*  
*word-wi-log-defs*

**lemma** *word-bw-assocs*:

$(x \text{ AND } y) \text{ AND } z = x \text{ AND } y \text{ AND } z$   
 $(x \text{ OR } y) \text{ OR } z = x \text{ OR } y \text{ OR } z$   
 $(x \text{ XOR } y) \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$   
**for**  $x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-bw-comms*:

$x \text{ AND } y = y \text{ AND } x$   
 $x \text{ OR } y = y \text{ OR } x$   
 $x \text{ XOR } y = y \text{ XOR } x$   
**for**  $x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-bw-lcs*:



$y \text{ AND } x \text{ AND } z = x \text{ AND } y \text{ AND } z$   
 $y \text{ OR } x \text{ OR } z = x \text{ OR } y \text{ OR } z$   
 $y \text{ XOR } x \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$   
**for**  $x :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemma** *word-log-esimps*:

$x \text{ AND } 0 = 0$   
 $x \text{ AND } -1 = x$   
 $x \text{ OR } 0 = x$   
 $x \text{ OR } -1 = -1$   
 $x \text{ XOR } 0 = x$   
 $x \text{ XOR } -1 = \text{NOT } x$   
 $0 \text{ AND } x = 0$   
 $-1 \text{ AND } x = x$   
 $0 \text{ OR } x = x$   
 $-1 \text{ OR } x = -1$   
 $0 \text{ XOR } x = x$   
 $-1 \text{ XOR } x = \text{NOT } x$   
**for**  $x :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemma** *word-not-dist*:

$\text{NOT } (x \text{ OR } y) = \text{NOT } x \text{ AND } \text{NOT } y$   
 $\text{NOT } (x \text{ AND } y) = \text{NOT } x \text{ OR } \text{NOT } y$   
**for**  $x :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemma** *word-bw-same*:

$x \text{ AND } x = x$   
 $x \text{ OR } x = x$   
 $x \text{ XOR } x = 0$   
**for**  $x :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemma** *word-ao-absorbs* [*simp*]:

$x \text{ AND } (y \text{ OR } x) = x$   
 $x \text{ OR } y \text{ AND } x = x$   
 $x \text{ AND } (x \text{ OR } y) = x$   
 $y \text{ AND } x \text{ OR } x = x$   
 $(y \text{ OR } x) \text{ AND } x = x$   
 $x \text{ OR } x \text{ AND } y = x$   
 $(x \text{ OR } y) \text{ AND } x = x$   
 $x \text{ AND } y \text{ OR } x = x$   
**for**  $x :: 'a::\text{len word}$   
 ⟨*proof*⟩

**lemma** *word-not-not* [*simp*]:  $\text{NOT } (\text{NOT } x) = x$

**for**  $x :: 'a::\text{len word}$

⟨proof⟩

**lemma** *word-ao-dist*:  $(x \text{ OR } y) \text{ AND } z = x \text{ AND } z \text{ OR } y \text{ AND } z$   
**for**  $x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-oa-dist*:  $x \text{ AND } y \text{ OR } z = (x \text{ OR } z) \text{ AND } (y \text{ OR } z)$   
**for**  $x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-add-not* [*simp*]:  $x + \text{NOT } x = -1$   
**for**  $x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-plus-and-or* [*simp*]:  $(x \text{ AND } y) + (x \text{ OR } y) = x + y$   
**for**  $x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *leoa*:  $w = x \text{ OR } y \implies y = w \text{ AND } y$   
**for**  $x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *leao*:  $w' = x' \text{ AND } y' \implies x' = x' \text{ OR } w'$   
**for**  $x' :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-ao-equiv*:  $w = w \text{ OR } w' \longleftrightarrow w' = w \text{ AND } w'$   
**for**  $w w' :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *le-word-or2*:  $x \leq x \text{ OR } y$   
**for**  $x y :: 'a::\text{len word}$   
 ⟨proof⟩

**lemmas** *le-word-or1* = *xtrans*(3) [*OF word-bw-comms* (2) *le-word-or2*]

**lemmas** *word-and-le1* = *xtrans*(3) [*OF word-ao-absorbs* (4) [*symmetric*] *le-word-or2*]

**lemmas** *word-and-le2* = *xtrans*(3) [*OF word-ao-absorbs* (8) [*symmetric*] *le-word-or2*]

**lemma** *bit-horner-sum-bit-word-iff* [*bit-simps*]:  
 ⟨*bit* (*horner-sum of-bool* (2 :: 'a::len word) *bs*) *n*  
 $\longleftrightarrow n < \min \text{LENGTH}('a) (\text{length } bs) \wedge bs ! n$ ⟩  
 ⟨proof⟩

**definition** *word-reverse* :: ⟨'a::len word  $\Rightarrow$  'a word⟩  
**where** ⟨*word-reverse*  $w = \text{horner-sum of-bool } 2 (\text{rev } (\text{map } (\text{bit } w) [0..<\text{LENGTH}('a)]))$ ⟩

**lemma** *bit-word-reverse-iff* [*bit-simps*]:  
 ⟨*bit* (*word-reverse*  $w$ )  $n \longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } w (\text{LENGTH}('a) - \text{Suc } n)$ ⟩  
**for**  $w :: 'a::\text{len word}$ ⟩

⟨proof⟩

**lemma** *word-rev-rev* [simp] : *word-reverse* (*word-reverse* *w*) = *w*  
 ⟨proof⟩

**lemma** *word-rev-gal*: *word-reverse* *w* = *u*  $\implies$  *word-reverse* *u* = *w*  
 ⟨proof⟩

**lemma** *word-rev-gal'*: *u* = *word-reverse* *w*  $\implies$  *w* = *word-reverse* *u*  
 ⟨proof⟩

**lemma** *word-eq-reverseI*:  
 ⟨*v* = *w*⟩ **if** ⟨*word-reverse* *v* = *word-reverse* *w*⟩  
 ⟨proof⟩

**lemma** *uint-2p*: (0::*a*::len *word*) < 2 <sup>*n*</sup>  $\implies$  *uint* (2 <sup>*n*</sup>::*a*::len *word*) = 2 <sup>*n*</sup>  
 ⟨proof⟩

**lemma** *word-of-int-2p*: (*word-of-int* (2 <sup>*n*</sup>)::*a*::len *word*) = 2 <sup>*n*</sup>  
 ⟨proof⟩

#### 106.24.1 shift functions in terms of lists of bools

**lemma** *drop-bit-word-numeral* [simp]:  
 ⟨*drop-bit* (*numeral* *n*) (*numeral* *k*) =

$$\langle \text{word-of-int } (\text{drop-bit } (\text{numeral } n) (\text{take-bit } \text{LENGTH}('a) (\text{numeral } k))) :: 'a::\text{len } \text{word} \rangle$$

⟨proof⟩

**lemma** *drop-bit-word-Suc-numeral* [simp]:  
 ⟨*drop-bit* (*Suc* *n*) (*numeral* *k*) =

$$\langle \text{word-of-int } (\text{drop-bit } (\text{Suc } n) (\text{take-bit } \text{LENGTH}('a) (\text{numeral } k))) :: 'a::\text{len } \text{word} \rangle$$

⟨proof⟩

**lemma** *drop-bit-word-minus-numeral* [simp]:  
 ⟨*drop-bit* (*numeral* *n*) (− *numeral* *k*) =

$$\langle \text{word-of-int } (\text{drop-bit } (\text{numeral } n) (\text{take-bit } \text{LENGTH}('a) (− \text{numeral } k))) :: 'a::\text{len } \text{word} \rangle$$

⟨proof⟩

**lemma** *drop-bit-word-Suc-minus-numeral* [simp]:  
 ⟨*drop-bit* (*Suc* *n*) (− *numeral* *k*) =

$$\langle \text{word-of-int } (\text{drop-bit } (\text{Suc } n) (\text{take-bit } \text{LENGTH}('a) (− \text{numeral } k))) :: 'a::\text{len } \text{word} \rangle$$

⟨proof⟩

**lemma** *signed-drop-bit-word-numeral* [simp]:  
 ⟨*signed-drop-bit* (*numeral* *n*) (*numeral* *k*) =

(word-of-int (drop-bit (numeral n) (signed-take-bit (LENGTH('a) - 1) (numeral k))) :: 'a::len word)›  
 ⟨proof⟩

**lemma** signed-drop-bit-word-Suc-numeral [simp]:

⟨signed-drop-bit (Suc n) (numeral k) =  
 (word-of-int (drop-bit (Suc n) (signed-take-bit (LENGTH('a) - 1) (numeral k))) :: 'a::len word)›  
 ⟨proof⟩

**lemma** signed-drop-bit-word-minus-numeral [simp]:

⟨signed-drop-bit (numeral n) (- numeral k) =  
 (word-of-int (drop-bit (numeral n) (signed-take-bit (LENGTH('a) - 1) (- numeral k))) :: 'a::len word)›  
 ⟨proof⟩

**lemma** signed-drop-bit-word-Suc-minus-numeral [simp]:

⟨signed-drop-bit (Suc n) (- numeral k) =  
 (word-of-int (drop-bit (Suc n) (signed-take-bit (LENGTH('a) - 1) (- numeral k))) :: 'a::len word)›  
 ⟨proof⟩

**lemma** take-bit-word-numeral [simp]:

⟨take-bit (numeral n) (numeral k) =  
 (word-of-int (take-bit (min LENGTH('a) (numeral n)) (numeral k)) :: 'a::len word)›  
 ⟨proof⟩

**lemma** take-bit-word-Suc-numeral [simp]:

⟨take-bit (Suc n) (numeral k) =  
 (word-of-int (take-bit (min LENGTH('a) (Suc n)) (numeral k)) :: 'a::len word)›  
 ⟨proof⟩

**lemma** take-bit-word-minus-numeral [simp]:

⟨take-bit (numeral n) (- numeral k) =  
 (word-of-int (take-bit (min LENGTH('a) (numeral n)) (- numeral k)) :: 'a::len word)›  
 ⟨proof⟩

**lemma** take-bit-word-Suc-minus-numeral [simp]:

⟨take-bit (Suc n) (- numeral k) =  
 (word-of-int (take-bit (min LENGTH('a) (Suc n)) (- numeral k)) :: 'a::len word)›  
 ⟨proof⟩

**lemma** signed-take-bit-word-numeral [simp]:

⟨signed-take-bit (numeral n) (numeral k) =  
 (word-of-int (signed-take-bit (numeral n) (take-bit LENGTH('a) (numeral k))) :: 'a::len word)›

⟨proof⟩

**lemma** *signed-take-bit-word-Suc-numeral* [simp]:

⟨signed-take-bit (Suc n) (numeral k) =  
 (word-of-int (signed-take-bit (Suc n) (take-bit LENGTH('a) (numeral k))) ::  
 'a::len word)⟩  
 ⟨proof⟩

**lemma** *signed-take-bit-word-minus-numeral* [simp]:

⟨signed-take-bit (numeral n) (− numeral k) =  
 (word-of-int (signed-take-bit (numeral n) (take-bit LENGTH('a) (− numeral  
 k))) :: 'a::len word)⟩  
 ⟨proof⟩

**lemma** *signed-take-bit-word-Suc-minus-numeral* [simp]:

⟨signed-take-bit (Suc n) (− numeral k) =  
 (word-of-int (signed-take-bit (Suc n) (take-bit LENGTH('a) (− numeral k))) ::  
 'a::len word)⟩  
 ⟨proof⟩

**lemma** *False-map2-or*:  $\llbracket \text{set } xs \subseteq \{False\}; \text{length } ys = \text{length } xs \rrbracket \implies \text{map2 } (\vee) \text{ } xs$   
 $ys = ys$   
 ⟨proof⟩

**lemma** *align-lem-or*:

**assumes**  $\text{length } xs = n + m$   $\text{length } ys = n + m$   
**and**  $\text{drop } m \text{ } xs = \text{replicate } n \text{ } False$   $\text{take } m \text{ } ys = \text{replicate } m \text{ } False$   
**shows**  $\text{map2 } (\vee) \text{ } xs \text{ } ys = \text{take } m \text{ } xs \text{ } @ \text{drop } m \text{ } ys$   
 ⟨proof⟩

**lemma** *False-map2-and*:  $\llbracket \text{set } xs \subseteq \{False\}; \text{length } ys = \text{length } xs \rrbracket \implies \text{map2 } (\wedge)$   
 $xs \text{ } ys = xs$   
 ⟨proof⟩

**lemma** *align-lem-and*:

**assumes**  $\text{length } xs = n + m$   $\text{length } ys = n + m$   
**and**  $\text{drop } m \text{ } xs = \text{replicate } n \text{ } False$   $\text{take } m \text{ } ys = \text{replicate } m \text{ } False$   
**shows**  $\text{map2 } (\wedge) \text{ } xs \text{ } ys = \text{replicate } (n + m) \text{ } False$   
 ⟨proof⟩

## 106.24.2 Mask

**lemma** *minus-1-eq-mask*:

⟨− 1 = (mask LENGTH('a) :: 'a::len word)⟩  
 ⟨proof⟩

**lemma** *mask-eq-decr-exp*:

⟨mask n =  $2^{\wedge} n - 1$  :: 'a::len word)⟩  
 ⟨proof⟩

**lemma** *mask-Suc-rec*:

$\langle \text{mask } (\text{Suc } n) = 2 * \text{mask } n + (1 :: 'a::\text{len word}) \rangle$   
 $\langle \text{proof} \rangle$

**context**

**begin**

**qualified lemma** *bit-mask-iff* [*bit-simps*]:

$\langle \text{bit } (\text{mask } m :: 'a::\text{len word}) n \longleftrightarrow n < \text{min LENGTH}('a) m \rangle$   
 $\langle \text{proof} \rangle$

**end**

**lemma** *mask-bin*:  $\text{mask } n = \text{word-of-int } (\text{take-bit } n (- 1))$

$\langle \text{proof} \rangle$

**lemma** *and-mask-bintr*:  $w \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n (\text{uint } w))$

$\langle \text{proof} \rangle$

**lemma** *and-mask-wi*:  $\text{word-of-int } i \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n i)$

$\langle \text{proof} \rangle$

**lemma** *and-mask-wi'*:

$\text{word-of-int } i \text{ AND } \text{mask } n = (\text{word-of-int } (\text{take-bit } (\text{min LENGTH}('a) n) i) :: 'a::\text{len word})$

$\langle \text{proof} \rangle$

**lemma** *and-mask-no*:  $\text{numeral } i \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n (\text{numeral } i))$

$\langle \text{proof} \rangle$

**lemma** *and-mask-mod-2p*:  $w \text{ AND } \text{mask } n = \text{word-of-int } (\text{uint } w \text{ mod } 2 \wedge n)$

$\langle \text{proof} \rangle$

**lemma** *uint-mask-eq*:

$\langle \text{uint } (\text{mask } n :: 'a::\text{len word}) = \text{mask } (\text{min LENGTH}('a) n) \rangle$

$\langle \text{proof} \rangle$

**lemma** *and-mask-lt-2p*:  $\text{uint } (w \text{ AND } \text{mask } n) < 2 \wedge n$

$\langle \text{proof} \rangle$

**lemma** *mask-eq-iff*:  $w \text{ AND } \text{mask } n = w \longleftrightarrow \text{uint } w < 2 \wedge n$

$\langle \text{proof} \rangle$

**lemma** *and-mask-dvd*:  $2 \wedge n \text{ dvd } \text{uint } w \longleftrightarrow w \text{ AND } \text{mask } n = 0$

$\langle \text{proof} \rangle$

**lemma** *and-mask-dvd-nat*:  $2 \wedge n \text{ dvd } \text{unat } w \longleftrightarrow w \text{ AND } \text{mask } n = 0$

⟨proof⟩

**lemma** *word-2p-lem*:  $n < \text{size } w \implies w < 2 \wedge n = (\text{uint } w < 2 \wedge n)$   
**for**  $w :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *less-mask-eq*:  
**fixes**  $x :: 'a::\text{len word}$   
**assumes**  $x < 2 \wedge n$  **shows**  $x \text{ AND mask } n = x$   
 ⟨proof⟩

**lemmas** *mask-eq-iff-w2p* = *trans* [OF *mask-eq-iff word-2p-lem* [*symmetric*]]

**lemmas** *and-mask-less'* = *iffD2* [OF *word-2p-lem and-mask-lt-2p*, *simplified word-size*]

**lemma** *and-mask-less-size*:  $n < \text{size } x \implies x \text{ AND mask } n < 2 \wedge n$   
**for**  $x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *word-mod-2p-is-mask* [OF *refl*]:  $c = 2 \wedge n \implies c > 0 \implies x \text{ mod } c = x$   
 AND *mask n*  
**for**  $c x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *mask-eqs*:  
 $(a \text{ AND mask } n) + b \text{ AND mask } n = a + b \text{ AND mask } n$   
 $a + (b \text{ AND mask } n) \text{ AND mask } n = a + b \text{ AND mask } n$   
 $(a \text{ AND mask } n) - b \text{ AND mask } n = a - b \text{ AND mask } n$   
 $a - (b \text{ AND mask } n) \text{ AND mask } n = a - b \text{ AND mask } n$   
 $a * (b \text{ AND mask } n) \text{ AND mask } n = a * b \text{ AND mask } n$   
 $(b \text{ AND mask } n) * a \text{ AND mask } n = b * a \text{ AND mask } n$   
 $(a \text{ AND mask } n) + (b \text{ AND mask } n) \text{ AND mask } n = a + b \text{ AND mask } n$   
 $(a \text{ AND mask } n) - (b \text{ AND mask } n) \text{ AND mask } n = a - b \text{ AND mask } n$   
 $(a \text{ AND mask } n) * (b \text{ AND mask } n) \text{ AND mask } n = a * b \text{ AND mask } n$   
 $-(a \text{ AND mask } n) \text{ AND mask } n = -a \text{ AND mask } n$   
 $\text{word-succ } (a \text{ AND mask } n) \text{ AND mask } n = \text{word-succ } a \text{ AND mask } n$   
 $\text{word-pred } (a \text{ AND mask } n) \text{ AND mask } n = \text{word-pred } a \text{ AND mask } n$   
 ⟨proof⟩

**lemma** *mask-power-eq*:  $(x \text{ AND mask } n) \wedge k \text{ AND mask } n = x \wedge k \text{ AND mask } n$   
**for**  $x :: 'a::\text{len word}$   
 ⟨proof⟩

**lemma** *mask-full* [*simp*]:  $\text{mask LENGTH } ('a) = (- 1 :: 'a::\text{len word})$   
 ⟨proof⟩

### 106.24.3 Slices

**definition** *slice1* ::  $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$

**where**  $\langle \text{slice1 } n \ w = (\text{if } n < \text{LENGTH}('a)$   
 $\text{then } \text{ucast } (\text{drop-bit } (\text{LENGTH}('a) - n) \ w)$   
 $\text{else } \text{push-bit } (n - \text{LENGTH}('a)) \ (\text{ucast } w)) \rangle$

**lemma** *bit-slice1-iff* [*bit-simps*]:

$\langle \text{bit } (\text{slice1 } m \ w :: 'b::\text{len word}) \ n \longleftrightarrow m - \text{LENGTH}('a) \leq n \wedge n < \min$   
 $\text{LENGTH}('b) \ m$   
 $\wedge \text{bit } w \ (n + (\text{LENGTH}('a) - m) - (m - \text{LENGTH}('a))) \rangle$   
**for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**definition** *slice* ::  $\langle \text{nat} \Rightarrow 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$

**where**  $\langle \text{slice } n = \text{slice1 } (\text{LENGTH}('a) - n) \rangle$

**lemma** *bit-slice-iff* [*bit-simps*]:

$\langle \text{bit } (\text{slice } m \ w :: 'b::\text{len word}) \ n \longleftrightarrow n < \min \text{LENGTH}('b) \ (\text{LENGTH}('a) -$   
 $m) \wedge \text{bit } w \ (n + \text{LENGTH}('a) - (\text{LENGTH}('a) - m)) \rangle$   
**for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *slice1-0* [*simp*] :  $\text{slice1 } n \ 0 = 0$

$\langle \text{proof} \rangle$

**lemma** *slice-0* [*simp*] :  $\text{slice } n \ 0 = 0$

$\langle \text{proof} \rangle$

**lemma** *ucast-slice1*:  $\text{ucast } w = \text{slice1 } (\text{size } w) \ w$

$\langle \text{proof} \rangle$

**lemma** *ucast-slice*:  $\text{ucast } w = \text{slice } 0 \ w$

$\langle \text{proof} \rangle$

**lemma** *slice-id*:  $\text{slice } 0 \ t = t$

$\langle \text{proof} \rangle$

**lemma** *rev-slice1*:

$\langle \text{slice1 } n \ (\text{word-reverse } w :: 'b::\text{len word}) = \text{word-reverse } (\text{slice1 } k \ w :: 'a::\text{len}$   
 $\text{word}) \rangle$   
**if**  $\langle n + k = \text{LENGTH}('a) + \text{LENGTH}('b) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *rev-slice*:

$n + k + \text{LENGTH}('a::\text{len}) = \text{LENGTH}('b::\text{len}) \implies$   
 $\text{slice } n \ (\text{word-reverse } (w::'b \ \text{word})) = \text{word-reverse } (\text{slice } k \ w :: 'a \ \text{word})$   
 $\langle \text{proof} \rangle$

#### 106.24.4 Revcast

**definition** *revcast* ::  $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$



**where**  $\langle \text{revcast} = \text{slice1 LENGTH}('b) \rangle$

**lemma** *bit-revcast-iff* [*bit-simps*]:

$\langle \text{bit} (\text{revcast } w :: 'b::\text{len word}) n \longleftrightarrow \text{LENGTH}('b) - \text{LENGTH}('a) \leq n \wedge n < \text{LENGTH}('b) \wedge \text{bit } w (n + (\text{LENGTH}('a) - \text{LENGTH}('b)) - (\text{LENGTH}('b) - \text{LENGTH}('a))) \rangle$   
**for**  $w :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *revcast-slice1* [*OF refl*]:  $rc = \text{revcast } w \implies \text{slice1} (\text{size } rc) w = rc$

$\langle \text{proof} \rangle$

**lemma** *revcast-rev-ucast* [*OF refl refl refl*]:

$cs = [rc, uc] \implies rc = \text{revcast} (\text{word-reverse } w) \implies uc = \text{ucast } w \implies rc = \text{word-reverse } uc$   
 $\langle \text{proof} \rangle$

**lemma** *revcast-ucast*:  $\text{revcast } w = \text{word-reverse} (\text{ucast} (\text{word-reverse } w))$

$\langle \text{proof} \rangle$

**lemma** *ucast-revcast*:  $\text{ucast } w = \text{word-reverse} (\text{revcast} (\text{word-reverse } w))$

$\langle \text{proof} \rangle$

**lemma** *ucast-rev-revcast*:  $\text{ucast} (\text{word-reverse } w) = \text{word-reverse} (\text{revcast } w)$

$\langle \text{proof} \rangle$

linking revcast and cast via shift

**lemmas** *wsst-TYs* = *source-size target-size word-size*

**lemmas** *sym-notr* =

*not-iff* [*THEN iffD2*, *THEN not-sym*, *THEN not-iff* [*THEN iffD1*]]

## 106.25 Split and cat

**lemmas** *word-split-bin'* = *word-split-def*

**lemmas** *word-cat-bin'* = *word-cat-eq*

— this odd result is analogous to *ucast-id*, result to the length given by the result type

**lemma** *word-cat-id*:  $\text{word-cat } a b = b$

$\langle \text{proof} \rangle$

**lemma** *word-cat-split-alt*:  $\llbracket \text{size } w \leq \text{size } u + \text{size } v; \text{word-split } w = (u,v) \rrbracket \implies \text{word-cat } u v = w$

$\langle \text{proof} \rangle$

**lemmas** *word-cat-split-size* = *sym* [*THEN* [2] *word-cat-split-alt* [*symmetric*]]

**106.25.1 Split and slice****lemma** *split-slices*:**assumes** *word-split*  $w = (u, v)$ **shows**  $u = \text{slice } (\text{size } v) w \wedge v = \text{slice } 0 w$  $\langle \text{proof} \rangle$ **lemma** *slice-cat1* [*OF refl*]: $\llbracket wc = \text{word-cat } a b; \text{size } a + \text{size } b \leq \text{size } wc \rrbracket \implies \text{slice } (\text{size } b) wc = a$  $\langle \text{proof} \rangle$ **lemmas** *slice-cat2 = trans* [*OF slice-id word-cat-id*]**lemma** *cat-slices*: $\llbracket a = \text{slice } n c; b = \text{slice } 0 c; n = \text{size } b; \text{size } c \leq \text{size } a + \text{size } b \rrbracket \implies \text{word-cat } a b = c$  $\langle \text{proof} \rangle$ **lemma** *word-split-cat-alt*:**assumes**  $w = \text{word-cat } u v$  **and** *size*:  $\text{size } u + \text{size } v \leq \text{size } w$ **shows** *word-split*  $w = (u, v)$  $\langle \text{proof} \rangle$ **lemma** *horner-sum-uint-exp-Cons-eg*: $\langle \text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) (w \# ws) =$  $\text{concat-bit LENGTH}('a) (\text{uint } w) (\text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) ws) \rangle$ **for**  $ws :: \langle 'a::\text{len word list} \rangle$  $\langle \text{proof} \rangle$ **lemma** *bit-horner-sum-uint-exp-iff*: $\langle \text{bit } (\text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) ws) n \longleftrightarrow$  $n \text{ div LENGTH}('a) < \text{length } ws \wedge \text{bit } (ws ! (n \text{ div LENGTH}('a))) (n \text{ mod LENGTH}('a)) \rangle$ **for**  $ws :: \langle 'a::\text{len word list} \rangle$  $\langle \text{proof} \rangle$ **106.26 Rotation****lemma** *word-rotr-word-rotr-eg*:  $\langle \text{word-rotr } m (\text{word-rotr } n w) = \text{word-rotr } (m + n) w \rangle$  $\langle \text{proof} \rangle$ **lemma** *word-rot-lem*:  $\llbracket l + k = d + k \text{ mod } l; n < l \rrbracket \implies ((d + n) \text{ mod } l) = n$  **for** $l::\text{nat}$  $\langle \text{proof} \rangle$ **lemma** *word-rot-rl* [*simp*]:  $\langle \text{word-rotl } k (\text{word-rotr } k v) = v \rangle$  $\langle \text{proof} \rangle$

**lemma** *word-rot-lr* [*simp*]:  $\langle \text{word-rotr } k (\text{word-rotl } k v) = v \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-rot-gal*:  
 $\langle \text{word-rotr } n v = w \longleftrightarrow \text{word-rotl } n w = v \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-rot-gal'*:  
 $\langle w = \text{word-rotr } n v \longleftrightarrow v = \text{word-rotl } n w \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-reverse-word-rotl*:  
 $\langle \text{word-reverse } (\text{word-rotl } n w) = \text{word-rotr } n (\text{word-reverse } w) \rangle$  (**is**  $\langle ?lhs = ?rhs \rangle$ )  
 $\langle \text{proof} \rangle$

**lemma** *word-reverse-word-rotr*:  
 $\langle \text{word-reverse } (\text{word-rotr } n w) = \text{word-rotl } n (\text{word-reverse } w) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-rotl-rev*:  
 $\langle \text{word-rotl } n w = \text{word-reverse } (\text{word-rotr } n (\text{word-reverse } w)) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-rotr-rev*:  
 $\langle \text{word-rotr } n w = \text{word-reverse } (\text{word-rotl } n (\text{word-reverse } w)) \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-roti-0* [*simp*]:  $\text{word-roti } 0 w = w$   
 $\langle \text{proof} \rangle$

**lemma** *word-roti-add*:  $\text{word-roti } (m + n) w = \text{word-roti } m (\text{word-roti } n w)$   
 $\langle \text{proof} \rangle$

**lemma** *word-roti-conv-mod'*:  
 $\text{word-roti } n w = \text{word-roti } (n \bmod \text{int } (\text{size } w)) w$   
 $\langle \text{proof} \rangle$

**lemmas** *word-roti-conv-mod = word-roti-conv-mod'* [*unfolded word-size*]

**end**

### 106.26.1 "Word rotation commutes with bit-wise operations

**locale** *word-rotate*

**begin**

**context**

**includes** *bit-operations-syntax*

**begin**

**lemma** *word-rot-logs*:

$\text{word-rotl } n \text{ (NOT } v) = \text{NOT (word-rotl } n \text{ } v)$   
 $\text{word-rotr } n \text{ (NOT } v) = \text{NOT (word-rotr } n \text{ } v)$   
 $\text{word-rotl } n \text{ (} x \text{ AND } y) = \text{word-rotl } n \text{ } x \text{ AND word-rotl } n \text{ } y$   
 $\text{word-rotr } n \text{ (} x \text{ AND } y) = \text{word-rotr } n \text{ } x \text{ AND word-rotr } n \text{ } y$   
 $\text{word-rotl } n \text{ (} x \text{ OR } y) = \text{word-rotl } n \text{ } x \text{ OR word-rotl } n \text{ } y$   
 $\text{word-rotr } n \text{ (} x \text{ OR } y) = \text{word-rotr } n \text{ } x \text{ OR word-rotr } n \text{ } y$   
 $\text{word-rotl } n \text{ (} x \text{ XOR } y) = \text{word-rotl } n \text{ } x \text{ XOR word-rotl } n \text{ } y$   
 $\text{word-rotr } n \text{ (} x \text{ XOR } y) = \text{word-rotr } n \text{ } x \text{ XOR word-rotr } n \text{ } y$   
 ⟨*proof*⟩

**end**

**end**

**lemmas** *word-rot-logs* = *word-rotate.word-rot-logs*

**lemma** *word-rotx-0* [*simp*] :  $\text{word-rotr } i \text{ } 0 = 0 \wedge \text{word-rotl } i \text{ } 0 = 0$   
 ⟨*proof*⟩

**lemma** *word-roti-0'* [*simp*] :  $\text{word-roti } n \text{ } 0 = 0$   
 ⟨*proof*⟩

**declare** *word-roti-eq-word-rotr-word-rotl* [*simp*]

## 106.27 Maximum machine word

**context**

**includes** *bit-operations-syntax*

**begin**

**lemma** *word-int-cases*:

**fixes**  $x :: 'a::\text{len word}$

**obtains**  $n$  **where**  $x = \text{word-of-int } n$  **and**  $0 \leq n$  **and**  $n < 2^{\text{LENGTH('a)}}$   
 ⟨*proof*⟩

**lemma** *word-nat-cases* [*cases type: word*]:

**fixes**  $x :: 'a::\text{len word}$

**obtains**  $n$  **where**  $x = \text{of-nat } n$  **and**  $n < 2^{\text{LENGTH('a)}}$   
 ⟨*proof*⟩

**lemma** *max-word-max* [*intro!*]:

⟨ $n \leq - 1$ ⟩ **for**  $n :: \langle 'a::\text{len word} \rangle$

⟨*proof*⟩

**lemma** *word-of-int-2p-len*:  $\text{word-of-int } (2^{\text{LENGTH('a)}}) = (0 :: 'a::\text{len word})$   
 ⟨*proof*⟩

**lemma** *word-pow-0*:  $(2::'a::len\ word) \wedge LENGTH('a) = 0$   
 ⟨*proof*⟩

**lemma** *max-word-wrap*:  
 $\langle x + 1 = 0 \implies x = - 1 \rangle$  **for**  $x :: \langle 'a::len\ word \rangle$   
 ⟨*proof*⟩

**lemma** *word-and-max*:  
 $\langle x\ AND\ - 1 = x \rangle$  **for**  $x :: \langle 'a::len\ word \rangle$   
 ⟨*proof*⟩

**lemma** *word-or-max*:  
 $\langle x\ OR\ - 1 = - 1 \rangle$  **for**  $x :: \langle 'a::len\ word \rangle$   
 ⟨*proof*⟩

**lemma** *word-ao-dist2*:  $x\ AND\ (y\ OR\ z) = x\ AND\ y\ OR\ x\ AND\ z$   
**for**  $x\ y\ z :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *word-oa-dist2*:  $x\ OR\ y\ AND\ z = (x\ OR\ y)\ AND\ (x\ OR\ z)$   
**for**  $x\ y\ z :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *word-and-not* [*simp*]:  $x\ AND\ NOT\ x = 0$   
**for**  $x :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *word-or-not* [*simp*]:  
 $\langle x\ OR\ NOT\ x = - 1 \rangle$  **for**  $x :: \langle 'a::len\ word \rangle$   
 ⟨*proof*⟩

**lemma** *word-xor-and-or*:  $x\ XOR\ y = x\ AND\ NOT\ y\ OR\ NOT\ x\ AND\ y$   
**for**  $x\ y :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *uint-lt-0* [*simp*]:  $uint\ x < 0 = False$   
 ⟨*proof*⟩

**lemma** *word-less-1* [*simp*]:  $x < 1 \iff x = 0$   
**for**  $x :: 'a::len\ word$   
 ⟨*proof*⟩

**lemma** *uint-plus-if-size*:  
 $uint\ (x + y) =$   
 (if  $uint\ x + uint\ y < 2^{\widehat{size}\ x}$   
 then  $uint\ x + uint\ y$   
 else  $uint\ x + uint\ y - 2^{\widehat{size}\ x}$ )  
 ⟨*proof*⟩

**lemma** *unat-plus-if-size*:

```
unat (x + y) =
  (if unat x + unat y < 2^size x
   then unat x + unat y
   else unat x + unat y - 2^size x)
for x y :: 'a::len word
⟨proof⟩
```

**lemma** *word-neq-0-conv*:  $w \neq 0 \longleftrightarrow 0 < w$

```
for w :: 'a::len word
⟨proof⟩
```

**lemma** *max-lt*:  $\text{unat} (\max a b \text{ div } c) = \text{unat} (\max a b) \text{ div } \text{unat } c$

```
for c :: 'a::len word
⟨proof⟩
```

**lemma** *uint-sub-if-size*:

```
uint (x - y) =
  (if uint y ≤ uint x
   then uint x - uint y
   else uint x - uint y + 2^size x)
⟨proof⟩
```

**lemma** *unat-sub*:

```
⟨unat (a - b) = unat a - unat b⟩
if ⟨b ≤ a⟩
⟨proof⟩
```

**lemmas** *word-less-sub1-numberof* [simp] = *word-less-sub1* [of numeral w] **for** w

**lemmas** *word-le-sub1-numberof* [simp] = *word-le-sub1* [of numeral w] **for** w

**lemma** *word-of-int-minus*:  $\text{word-of-int} (2^{\text{LENGTH('a)}} - i) = (\text{word-of-int } (-i))::'a::\text{len word}$

```
⟨proof⟩
```

**lemma** *word-of-int-inj*:

```
⟨(word-of-int x :: 'a::len word) = word-of-int y ⟷ x = y⟩
if ⟨0 ≤ x ∧ x < 2^LENGTH('a)⟩ ⟨0 ≤ y ∧ y < 2^LENGTH('a)⟩
⟨proof⟩
```

**lemma** *word-le-less-eq*:  $x \leq y \longleftrightarrow x = y \vee x < y$

```
for x y :: 'z::len word
⟨proof⟩
```

**lemma** *mod-plus-cong*:

```
fixes b b' :: int
assumes 1: b = b'
and 2: x mod b' = x' mod b'
and 3: y mod b' = y' mod b'
```

**and**  $4: x' + y' = z'$   
**shows**  $(x + y) \bmod b = z' \bmod b'$   
 $\langle \text{proof} \rangle$

**lemma** *mod-minus-cong*:  
**fixes**  $b b' :: \text{int}$   
**assumes**  $b = b'$   
**and**  $x \bmod b' = x' \bmod b'$   
**and**  $y \bmod b' = y' \bmod b'$   
**and**  $x' - y' = z'$   
**shows**  $(x - y) \bmod b = z' \bmod b'$   
 $\langle \text{proof} \rangle$

**lemma** *word-induct-less* [*case-names zero less*]:  
 $\langle P m \rangle$  **if** *zero*:  $\langle P 0 \rangle$  **and** *less*:  $\langle \bigwedge n. n < m \implies P n \implies P (1 + n) \rangle$   
**for**  $m :: \langle 'a::\text{len word} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *word-induct*:  $P 0 \implies (\bigwedge n. P n \implies P (1 + n)) \implies P m$   
**for**  $P :: 'a::\text{len word} \Rightarrow \text{bool}$   
 $\langle \text{proof} \rangle$

**lemma** *word-induct2* [*case-names zero suc, induct type*]:  $P 0 \implies (\bigwedge n. 1 + n \neq 0 \implies P n \implies P (1 + n)) \implies P n$   
**for**  $P :: 'b::\text{len word} \Rightarrow \text{bool}$   
 $\langle \text{proof} \rangle$

## 106.28 Recursion combinator for words

**definition** *word-rec* ::  $'a \Rightarrow ('b::\text{len word} \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b \text{ word} \Rightarrow 'a$   
**where** *word-rec* *forZero forSuc*  $n = \text{rec-nat forZero (forSuc } \circ \text{ of-nat)} (\text{unat } n)$

**lemma** *word-rec-0* [*simp*]:  $\text{word-rec } z s 0 = z$   
 $\langle \text{proof} \rangle$

**lemma** *word-rec-Suc* [*simp*]:  $1 + n \neq 0 \implies \text{word-rec } z s (1 + n) = s n (\text{word-rec } z s n)$   
**for**  $n :: 'a::\text{len word}$   
 $\langle \text{proof} \rangle$

**lemma** *word-rec-Pred*:  $n \neq 0 \implies \text{word-rec } z s n = s (n - 1) (\text{word-rec } z s (n - 1))$   
 $\langle \text{proof} \rangle$

**lemma** *word-rec-in*:  $f (\text{word-rec } z (\lambda-. f) n) = \text{word-rec } (f z) (\lambda-. f) n$   
 $\langle \text{proof} \rangle$

**lemma** *word-rec-in2*:  $f n (\text{word-rec } z f n) = \text{word-rec } (f 0 z) (f \circ (+) 1) n$   
 $\langle \text{proof} \rangle$

**lemma** *word-rec-twice*:

$m \leq n \implies \text{word-rec } z \ f \ n = \text{word-rec } (\text{word-rec } z \ f \ (n - m)) \ (f \circ (+) \ (n - m))$   
 $m$   
 ⟨*proof*⟩

**lemma** *word-rec-id*:  $\text{word-rec } z \ (\lambda-. \text{id}) \ n = z$

⟨*proof*⟩

**lemma** *word-rec-id-eq*:  $(\bigwedge m. m < n \implies f \ m = \text{id}) \implies \text{word-rec } z \ f \ n = z$

⟨*proof*⟩

**lemma** *word-rec-max*:

**assumes**  $\forall m \geq n. m \neq -1 \longrightarrow f \ m = \text{id}$

**shows**  $\text{word-rec } z \ f \ (-1) = \text{word-rec } z \ f \ n$

⟨*proof*⟩

**end**

## 106.29 Tool support

⟨*ML*⟩

**end**

## 107 The Field of Integers mod 2

**theory** *Z2*

**imports** *Main*

**begin**

Note that in most cases *bool* is appropriate when a binary type is needed; the type provided here, for historical reasons named *bit*, is only needed if proper field operations are required.

**typedef** *bit* = ⟨*UNIV* :: *bool set*⟩ ⟨*proof*⟩

**instantiation** *bit* :: *zero-neq-one*

**begin**

**definition** *zero-bit* :: *bit*

**where** ⟨*0* = *Abs-bit False*⟩

**definition** *one-bit* :: *bit*

**where** ⟨*1* = *Abs-bit True*⟩

**instance**

⟨*proof*⟩

**end**



**free-constructors** *case-bit* **for**  $\langle 0::bit \rangle \mid \langle 1::bit \rangle$   
 $\langle proof \rangle$

**lemma** *bit-not-zero-iff* [*simp*]:  
 $\langle a \neq 0 \longleftrightarrow a = 1 \rangle$  **for**  $a :: bit$   
 $\langle proof \rangle$

**lemma** *bit-not-one-iff* [*simp*]:  
 $\langle a \neq 1 \longleftrightarrow a = 0 \rangle$  **for**  $a :: bit$   
 $\langle proof \rangle$

**instantiation** *bit* :: *semidom-modulo*  
**begin**

**definition** *plus-bit* ::  $\langle bit \Rightarrow bit \Rightarrow bit \rangle$   
**where**  $\langle a + b = Abs-bit (Rep-bit a \neq Rep-bit b) \rangle$

**definition** *minus-bit* ::  $\langle bit \Rightarrow bit \Rightarrow bit \rangle$   
**where** [*simp*]:  $\langle minus-bit = plus \rangle$

**definition** *times-bit* ::  $\langle bit \Rightarrow bit \Rightarrow bit \rangle$   
**where**  $\langle a * b = Abs-bit (Rep-bit a \wedge Rep-bit b) \rangle$

**definition** *divide-bit* ::  $\langle bit \Rightarrow bit \Rightarrow bit \rangle$   
**where** [*simp*]:  $\langle divide-bit = times \rangle$

**definition** *modulo-bit* ::  $\langle bit \Rightarrow bit \Rightarrow bit \rangle$   
**where**  $\langle a \bmod b = Abs-bit (Rep-bit a \wedge \neg Rep-bit b) \rangle$

**instance**  
 $\langle proof \rangle$

**end**

**lemma** *bit-2-eq-0* [*simp*]:  
 $\langle 2 = (0::bit) \rangle$   
 $\langle proof \rangle$

**instance** *bit* :: *semiring-parity*  
 $\langle proof \rangle$

**lemma** *Abs-bit-eq-of-bool* [*code-abbrev*]:  
 $\langle Abs-bit = of-bool \rangle$   
 $\langle proof \rangle$

**lemma** *Rep-bit-eq-odd*:  
 $\langle Rep-bit = odd \rangle$   
 $\langle proof \rangle$

```

lemma Rep-bit-iff-odd [code-abbrev]:
  ⟨Rep-bit b  $\longleftrightarrow$  odd b⟩
  ⟨proof⟩

lemma Not-Rep-bit-iff-even [code-abbrev]:
  ⟨ $\neg$  Rep-bit b  $\longleftrightarrow$  even b⟩
  ⟨proof⟩

lemma Not-Not-Rep-bit [code-unfold]:
  ⟨ $\neg \neg$  Rep-bit b  $\longleftrightarrow$  Rep-bit b⟩
  ⟨proof⟩

code-datatype ⟨0::bit⟩ ⟨1::bit⟩

lemma Abs-bit-code [code]:
  ⟨Abs-bit False = 0⟩
  ⟨Abs-bit True = 1⟩
  ⟨proof⟩

lemma Rep-bit-code [code]:
  ⟨Rep-bit 0  $\longleftrightarrow$  False⟩
  ⟨Rep-bit 1  $\longleftrightarrow$  True⟩
  ⟨proof⟩

context zero-neq-one
begin

abbreviation of-bit :: ⟨bit  $\Rightarrow$  'a'⟩
  where ⟨of-bit b  $\equiv$  of-bool (odd b)⟩

end

context
begin

qualified lemma bit-eq-iff:
  ⟨a = b  $\longleftrightarrow$  (even a  $\longleftrightarrow$  even b)⟩ for a b :: bit
  ⟨proof⟩

end

lemma modulo-bit-unfold [simp, code]:
  ⟨a mod b = of-bool (odd a  $\wedge$  even b)⟩ for a b :: bit
  ⟨proof⟩

lemma power-bit-unfold [simp]:
  ⟨a  $\wedge$  n = of-bool (odd a  $\vee$  n = 0)⟩ for a :: bit
  ⟨proof⟩

```

**instantiation** *bit* :: *field*

**begin**

**definition** *uminus-bit* ::  $\langle \text{bit} \Rightarrow \text{bit} \rangle$

**where** [*simp*]:  $\langle \text{uminus-bit} = \text{id} \rangle$

**definition** *inverse-bit* ::  $\langle \text{bit} \Rightarrow \text{bit} \rangle$

**where** [*simp*]:  $\langle \text{inverse-bit} = \text{id} \rangle$

**instance**

$\langle \text{proof} \rangle$

**end**

**instantiation** *bit* :: *semiring-bits*

**begin**

**definition** *bit-bit* ::  $\langle \text{bit} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$

**where** [*simp*]:  $\langle \text{bit-bit } b \ n \longleftrightarrow \text{odd } b \wedge n = 0 \rangle$

**instance**

$\langle \text{proof} \rangle$

**end**

**instantiation** *bit* :: *ring-bit-operations*

**begin**

**context**

**includes** *bit-operations-syntax*

**begin**

**definition** *not-bit* ::  $\langle \text{bit} \Rightarrow \text{bit} \rangle$

**where** [*simp*]:  $\langle \text{NOT } b = \text{of-bool } (\text{even } b) \rangle$  **for** *b* :: *bit*

**definition** *and-bit* ::  $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$

**where** [*simp*]:  $\langle b \ \text{AND} \ c = \text{of-bool } (\text{odd } b \wedge \text{odd } c) \rangle$  **for** *b c* :: *bit*

**definition** *or-bit* ::  $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$

**where** [*simp*]:  $\langle b \ \text{OR} \ c = \text{of-bool } (\text{odd } b \vee \text{odd } c) \rangle$  **for** *b c* :: *bit*

**definition** *xor-bit* ::  $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$

**where** [*simp*]:  $\langle b \ \text{XOR} \ c = \text{of-bool } (\text{odd } b \neq \text{odd } c) \rangle$  **for** *b c* :: *bit*

**definition** *mask-bit* ::  $\langle \text{nat} \Rightarrow \text{bit} \rangle$

**where** [*simp*]:  $\langle \text{mask } n = (\text{of-bool } (n > 0)) :: \text{bit} \rangle$

**definition** *set-bit-bit* ::  $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$

```

  where [simp]: ⟨set-bit n b = of-bool (n = 0 ∨ odd b)⟩ for b :: bit

definition unset-bit-bit :: ⟨nat ⇒ bit ⇒ bit⟩
  where [simp]: ⟨unset-bit n b = of-bool (n > 0 ∧ odd b)⟩ for b :: bit

definition flip-bit-bit :: ⟨nat ⇒ bit ⇒ bit⟩
  where [simp]: ⟨flip-bit n b = of-bool ((n = 0) ≠ odd b)⟩ for b :: bit

definition push-bit-bit :: ⟨nat ⇒ bit ⇒ bit⟩
  where [simp]: ⟨push-bit n b = of-bool (odd b ∧ n = 0)⟩ for b :: bit

definition drop-bit-bit :: ⟨nat ⇒ bit ⇒ bit⟩
  where [simp]: ⟨drop-bit n b = of-bool (odd b ∧ n = 0)⟩ for b :: bit

definition take-bit-bit :: ⟨nat ⇒ bit ⇒ bit⟩
  where [simp]: ⟨take-bit n b = of-bool (odd b ∧ n > 0)⟩ for b :: bit

end

instance
  ⟨proof⟩

end

lemma add-bit-eq-xor [simp, code]:
  ⟨(+) = (Bit-Operations.xor :: bit ⇒ -)⟩
  ⟨proof⟩

lemma mult-bit-eq-and [simp, code]:
  ⟨(*) = (Bit-Operations.and :: bit ⇒ -)⟩
  ⟨proof⟩

lemma bit-numeral-even [simp]:
  ⟨numeral (Num.Bit0 n) = (0 :: bit)⟩
  ⟨proof⟩

lemma bit-numeral-odd [simp]:
  ⟨numeral (Num.Bit1 n) = (1 :: bit)⟩
  ⟨proof⟩

end

```

## 108 Pointwise order on product types

```

theory Product-Order
imports Product-Plus
begin

```

### 108.1 Pointwise ordering

**instantiation** *prod* :: (*ord*, *ord*) *ord*  
**begin**

**definition**

$$x \leq y \iff \text{fst } x \leq \text{fst } y \wedge \text{snd } x \leq \text{snd } y$$

**definition**

$$(x::'a \times 'b) < y \iff x \leq y \wedge \neg y \leq x$$

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

**end**

**lemma** *fst-mono*:  $x \leq y \implies \text{fst } x \leq \text{fst } y$   
 $\langle \text{proof} \rangle$

**lemma** *snd-mono*:  $x \leq y \implies \text{snd } x \leq \text{snd } y$   
 $\langle \text{proof} \rangle$

**lemma** *Pair-mono*:  $x \leq x' \implies y \leq y' \implies (x, y) \leq (x', y')$   
 $\langle \text{proof} \rangle$

**lemma** *Pair-le* [*simp*]:  $(a, b) \leq (c, d) \iff a \leq c \wedge b \leq d$   
 $\langle \text{proof} \rangle$

**lemma** *atLeastAtMost-prod-eq*:  $\{a..b\} = \{\text{fst } a.. \text{fst } b\} \times \{\text{snd } a.. \text{snd } b\}$   
 $\langle \text{proof} \rangle$

**instance** *prod* :: (*preorder*, *preorder*) *preorder*  
 $\langle \text{proof} \rangle$

**instance** *prod* :: (*order*, *order*) *order*  
 $\langle \text{proof} \rangle$

### 108.2 Binary infimum and supremum

**instantiation** *prod* :: (*inf*, *inf*) *inf*  
**begin**

**definition**  $\text{inf } x \ y = (\text{inf } (\text{fst } x) (\text{fst } y), \text{inf } (\text{snd } x) (\text{snd } y))$

**lemma** *inf-Pair-Pair* [*simp*]:  $\text{inf } (a, b) (c, d) = (\text{inf } a \ c, \text{inf } b \ d)$   
 $\langle \text{proof} \rangle$

**lemma** *fst-inf* [*simp*]:  $\text{fst } (\text{inf } x \ y) = \text{inf } (\text{fst } x) (\text{fst } y)$   
 $\langle \text{proof} \rangle$

**lemma** *snd-inf* [*simp*]:  $\text{snd } (\text{inf } x \ y) = \text{inf } (\text{snd } x) (\text{snd } y)$

```

  <proof>
instance <proof>
end

instance prod :: (semilattice-inf, semilattice-inf) semilattice-inf
  <proof>

instantiation prod :: (sup, sup) sup
begin

definition
  sup x y = (sup (fst x) (fst y), sup (snd x) (snd y))

lemma sup-Pair-Pair [simp]: sup (a, b) (c, d) = (sup a c, sup b d)
  <proof>

lemma fst-sup [simp]: fst (sup x y) = sup (fst x) (fst y)
  <proof>

lemma snd-sup [simp]: snd (sup x y) = sup (snd x) (snd y)
  <proof>

instance <proof>

end

instance prod :: (semilattice-sup, semilattice-sup) semilattice-sup
  <proof>

instance prod :: (lattice, lattice) lattice <proof>

instance prod :: (distrib-lattice, distrib-lattice) distrib-lattice
  <proof>

```

### 108.3 Top and bottom elements

```

instantiation prod :: (top, top) top
begin

definition
  top = (top, top)

instance <proof>

end

```

```

lemma fst-top [simp]: fst top = top
  ⟨proof⟩

lemma snd-top [simp]: snd top = top
  ⟨proof⟩

lemma Pair-top-top: (top, top) = top
  ⟨proof⟩

instance prod :: (order-top, order-top) order-top
  ⟨proof⟩

instantiation prod :: (bot, bot) bot
begin

definition
  bot = (bot, bot)

instance ⟨proof⟩

end

lemma fst-bot [simp]: fst bot = bot
  ⟨proof⟩

lemma snd-bot [simp]: snd bot = bot
  ⟨proof⟩

lemma Pair-bot-bot: (bot, bot) = bot
  ⟨proof⟩

instance prod :: (order-bot, order-bot) order-bot
  ⟨proof⟩

instance prod :: (bounded-lattice, bounded-lattice) bounded-lattice ⟨proof⟩

instance prod :: (boolean-algebra, boolean-algebra) boolean-algebra
  ⟨proof⟩

```

#### 108.4 Complete lattice operations

```

instantiation prod :: (Inf, Inf) Inf
begin

definition Inf A = (INF x∈A. fst x, INF x∈A. snd x)

instance ⟨proof⟩

end

```

**instantiation**  $prod :: (Sup, Sup) Sup$   
**begin**

**definition**  $Sup A = (SUP x \in A. fst x, SUP x \in A. snd x)$

**instance**  $\langle proof \rangle$

**end**

**instance**  $prod :: (conditionally-complete-lattice, conditionally-complete-lattice)$   
 $conditionally-complete-lattice$   
 $\langle proof \rangle$

**instance**  $prod :: (complete-lattice, complete-lattice) complete-lattice$   
 $\langle proof \rangle$

**lemma**  $fst-Inf: fst (Inf A) = (INF x \in A. fst x)$   
 $\langle proof \rangle$

**lemma**  $fst-INF: fst (INF x \in A. f x) = (INF x \in A. fst (f x))$   
 $\langle proof \rangle$

**lemma**  $fst-Sup: fst (Sup A) = (SUP x \in A. fst x)$   
 $\langle proof \rangle$

**lemma**  $fst-SUP: fst (SUP x \in A. f x) = (SUP x \in A. fst (f x))$   
 $\langle proof \rangle$

**lemma**  $snd-Inf: snd (Inf A) = (INF x \in A. snd x)$   
 $\langle proof \rangle$

**lemma**  $snd-INF: snd (INF x \in A. f x) = (INF x \in A. snd (f x))$   
 $\langle proof \rangle$

**lemma**  $snd-Sup: snd (Sup A) = (SUP x \in A. snd x)$   
 $\langle proof \rangle$

**lemma**  $snd-SUP: snd (SUP x \in A. f x) = (SUP x \in A. snd (f x))$   
 $\langle proof \rangle$

**lemma**  $INF-Pair: (INF x \in A. (f x, g x)) = (INF x \in A. f x, INF x \in A. g x)$   
 $\langle proof \rangle$

**lemma**  $SUP-Pair: (SUP x \in A. (f x, g x)) = (SUP x \in A. f x, SUP x \in A. g x)$   
 $\langle proof \rangle$

Alternative formulations for set infima and suprema over the product of two complete lattices:



**lemma** *INF-prod-alt-def*:

$Inf (f \text{ ' } A) = (Inf ((fst \circ f) \text{ ' } A), Inf ((snd \circ f) \text{ ' } A))$   
 $\langle proof \rangle$

**lemma** *SUP-prod-alt-def*:

$Sup (f \text{ ' } A) = (Sup ((fst \circ f) \text{ ' } A), Sup((snd \circ f) \text{ ' } A))$   
 $\langle proof \rangle$

## 108.5 Complete distributive lattices

**instance** *prod* :: (*complete-distrib-lattice*, *complete-distrib-lattice*) *complete-distrib-lattice*

$\langle proof \rangle$

## 108.6 Bekic’s Theorem

Simultaneous fixed points over pairs can be written in terms of separate fixed points. Transliterated from HOLCF.Fix by Peter Gammie

**lemma** *lfp-prod*:

**fixes**  $F :: 'a::complete-lattice \times 'b::complete-lattice \Rightarrow 'a \times 'b$   
**assumes** *mono F*  
**shows**  $lfp F = (lfp (\lambda x. fst (F (x, lfp (\lambda y. snd (F (x, y)))))),$   
 $(lfp (\lambda y. snd (F (lfp (\lambda x. fst (F (x, lfp (\lambda y. snd (F (x, y)))))), y))))$   
**(is**  $lfp F = (?x, ?y)$   
 $\langle proof \rangle$

**lemma** *gfp-prod*:

**fixes**  $F :: 'a::complete-lattice \times 'b::complete-lattice \Rightarrow 'a \times 'b$   
**assumes** *mono F*  
**shows**  $gfp F = (gfp (\lambda x. fst (F (x, gfp (\lambda y. snd (F (x, y)))))),$   
 $(gfp (\lambda y. snd (F (gfp (\lambda x. fst (F (x, gfp (\lambda y. snd (F (x, y)))))), y))))$   
**(is**  $gfp F = (?x, ?y)$   
 $\langle proof \rangle$

**end**

## 109 Finite Lattices

**theory** *Finite-Lattice*  
**imports** *Product-Order*  
**begin**

### 109.1 Finite Complete Lattices

A non-empty finite lattice is a complete lattice. Since types are never empty in Isabelle/HOL, a type of classes *finite* and *lattice* should also have class *complete-lattice*. A type class is defined that extends classes *finite* and *lattice* with the operators *bot*, *top*, *Inf*, and *Sup*, along with assumptions that define

these operators in terms of the ones of classes *finite* and *lattice*. The resulting class is a subclass of *complete-lattice*.

```

class finite-lattice-complete = finite + lattice + bot + top + Inf + Sup +
  assumes bot-def: bot = Inf-fin UNIV
  assumes top-def: top = Sup-fin UNIV
  assumes Inf-def: Inf A = Finite-Set.fold inf top A
  assumes Sup-def: Sup A = Finite-Set.fold sup bot A

```

The definitional assumptions on the operators *bot* and *top* of class *finite-lattice-complete* ensure that they yield bottom and top.

**lemma** *finite-lattice-complete-bot-least*:  $(\text{bot} :: 'a :: \text{finite-lattice-complete}) \leq x$   
 ⟨proof⟩

**instance** *finite-lattice-complete*  $\subseteq$  *order-bot*  
 ⟨proof⟩

**lemma** *finite-lattice-complete-top-greatest*:  $(\text{top} :: 'a :: \text{finite-lattice-complete}) \geq x$   
 ⟨proof⟩

**instance** *finite-lattice-complete*  $\subseteq$  *order-top*  
 ⟨proof⟩

**instance** *finite-lattice-complete*  $\subseteq$  *bounded-lattice* ⟨proof⟩

The definitional assumptions on the operators *Inf* and *Sup* of class *finite-lattice-complete* ensure that they yield infimum and supremum.

**lemma** *finite-lattice-complete-Inf-empty*:  $\text{Inf } \{\} = (\text{top} :: 'a :: \text{finite-lattice-complete})$   
 ⟨proof⟩

**lemma** *finite-lattice-complete-Sup-empty*:  $\text{Sup } \{\} = (\text{bot} :: 'a :: \text{finite-lattice-complete})$   
 ⟨proof⟩

**lemma** *finite-lattice-complete-Inf-insert*:  
**fixes**  $A :: 'a :: \text{finite-lattice-complete}$  *set*  
**shows**  $\text{Inf } (\text{insert } x A) = \text{inf } x (\text{Inf } A)$   
 ⟨proof⟩

**lemma** *finite-lattice-complete-Sup-insert*:  
**fixes**  $A :: 'a :: \text{finite-lattice-complete}$  *set*  
**shows**  $\text{Sup } (\text{insert } x A) = \text{sup } x (\text{Sup } A)$   
 ⟨proof⟩

**lemma** *finite-lattice-complete-Inf-lower*:  
 $(x :: 'a :: \text{finite-lattice-complete}) \in A \implies \text{Inf } A \leq x$   
 ⟨proof⟩

**lemma** *finite-lattice-complete-Inf-greatest*:  
 $\forall x :: 'a :: \text{finite-lattice-complete} \in A. z \leq x \implies z \leq \text{Inf } A$   
 ⟨proof⟩

**lemma** *finite-lattice-complete-Sup-upper*:  
 $(x :: 'a :: \text{finite-lattice-complete}) \in A \implies \text{Sup } A \geq x$   
 ⟨proof⟩

**lemma** *finite-lattice-complete-Sup-least*:  
 $\forall x :: 'a :: \text{finite-lattice-complete} \in A. z \geq x \implies z \geq \text{Sup } A$   
 ⟨proof⟩

**instance** *finite-lattice-complete*  $\subseteq$  *complete-lattice*  
 ⟨proof⟩

The product of two finite lattices is already a finite lattice.

**lemma** *finite-bot-prod*:  
 $(\text{bot} :: ('a :: \text{finite-lattice-complete} \times 'b :: \text{finite-lattice-complete})) =$   
 $\text{Inf-fin UNIV}$   
 ⟨proof⟩

**lemma** *finite-top-prod*:  
 $(\text{top} :: ('a :: \text{finite-lattice-complete} \times 'b :: \text{finite-lattice-complete})) =$   
 $\text{Sup-fin UNIV}$   
 ⟨proof⟩

**lemma** *finite-Inf-prod*:  
 $\text{Inf}(A :: ('a :: \text{finite-lattice-complete} \times 'b :: \text{finite-lattice-complete}) \text{ set}) =$   
 $\text{Finite-Set.fold inf top } A$   
 ⟨proof⟩

**lemma** *finite-Sup-prod*:  
 $\text{Sup}(A :: ('a :: \text{finite-lattice-complete} \times 'b :: \text{finite-lattice-complete}) \text{ set}) =$   
 $\text{Finite-Set.fold sup bot } A$   
 ⟨proof⟩

**instance** *prod* ::  $(\text{finite-lattice-complete}, \text{finite-lattice-complete})$  *finite-lattice-complete*  
 ⟨proof⟩

Functions with a finite domain and with a finite lattice as codomain already form a finite lattice.

**lemma** *finite-bot-fun*:  $(\text{bot} :: ('a :: \text{finite} \Rightarrow 'b :: \text{finite-lattice-complete})) = \text{Inf-fin UNIV}$   
 ⟨proof⟩

**lemma** *finite-top-fun*:  $(\text{top} :: ('a :: \text{finite} \Rightarrow 'b :: \text{finite-lattice-complete})) = \text{Sup-fin UNIV}$   
 ⟨proof⟩

**lemma** *finite-Inf-fun*:  
 $\text{Inf}(A :: ('a :: \text{finite} \Rightarrow 'b :: \text{finite-lattice-complete}) \text{ set}) =$   
 $\text{Finite-Set.fold inf top } A$   
 ⟨proof⟩

**lemma** *finite-Sup-fun*:  
 $Sup (A::('a::finite \Rightarrow 'b::finite-lattice-complete) set) =$   
 $Finite-Set.fold sup bot A$   
 ⟨proof⟩

**instance** *fun* :: (*finite*, *finite-lattice-complete*) *finite-lattice-complete*  
 ⟨proof⟩

## 109.2 Finite Distributive Lattices

A finite distributive lattice is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

**class** *finite-distrib-lattice-complete* =  
*distrib-lattice* + *finite-lattice-complete*

**lemma** *finite-distrib-lattice-complete-sup-Inf*:  
 $sup (x::'a::finite-distrib-lattice-complete) (Inf A) = (INF y \in A. sup x y)$   
 ⟨proof⟩

**lemma** *finite-distrib-lattice-complete-inf-Sup*:  
 $inf (x::'a::finite-distrib-lattice-complete) (Sup A) = (SUP y \in A. inf x y)$   
 ⟨proof⟩

**context** *finite-distrib-lattice-complete*  
**begin**  
**subclass** *finite-distrib-lattice*  
 ⟨proof⟩  
**end**

**instance** *finite-distrib-lattice-complete*  $\subseteq$  *complete-distrib-lattice* ⟨proof⟩

The product of two finite distributive lattices is already a finite distributive lattice.

**instance** *prod* ::  
 (*finite-distrib-lattice-complete*, *finite-distrib-lattice-complete*)  
*finite-distrib-lattice-complete*  
 ⟨proof⟩

Functions with a finite domain and with a finite distributive lattice as codomain already form a finite distributive lattice.

**instance** *fun* ::  
 (*finite*, *finite-distrib-lattice-complete*) *finite-distrib-lattice-complete*  
 ⟨proof⟩

## 109.3 Linear Orders

A linear order is a distributive lattice. A type class is defined that extends class *linorder* with the operators *inf* and *sup*, along with assumptions that

define these operators in terms of the ones of class *linorder*. The resulting class is a subclass of *distrib-lattice*.

```
class linorder-lattice = linorder + inf + sup +
  assumes inf-def: inf x y = (if x ≤ y then x else y)
  assumes sup-def: sup x y = (if x ≥ y then x else y)
```

The definitional assumptions on the operators *inf* and *sup* of class *linorder-lattice* ensure that they yield infimum and supremum and that they distribute over each other.

```
lemma linorder-lattice-inf-le1: inf (x::'a::linorder-lattice) y ≤ x
  <proof>
```

```
lemma linorder-lattice-inf-le2: inf (x::'a::linorder-lattice) y ≤ y
  <proof>
```

```
lemma linorder-lattice-inf-greatest:
  (x::'a::linorder-lattice) ≤ y ⇒ x ≤ z ⇒ x ≤ inf y z
  <proof>
```

```
lemma linorder-lattice-sup-ge1: sup (x::'a::linorder-lattice) y ≥ x
  <proof>
```

```
lemma linorder-lattice-sup-ge2: sup (x::'a::linorder-lattice) y ≥ y
  <proof>
```

```
lemma linorder-lattice-sup-least:
  (x::'a::linorder-lattice) ≥ y ⇒ x ≥ z ⇒ x ≥ sup y z
  <proof>
```

```
lemma linorder-lattice-sup-inf-distrib1:
  sup (x::'a::linorder-lattice) (inf y z) = inf (sup x y) (sup x z)
  <proof>
```

```
instance linorder-lattice ⊆ distrib-lattice
  <proof>
```

## 109.4 Finite Linear Orders

A (non-empty) finite linear order is a complete linear order.

```
class finite-linorder-complete = linorder-lattice + finite-lattice-complete
```

```
instance finite-linorder-complete ⊆ complete-linorder <proof>
```

A (non-empty) finite linear order is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

```
instance finite-linorder-complete ⊆ finite-distrib-lattice-complete <proof>
```

```
end
```

## 110 Lexicographic order on lists

**theory** *List-Lexorder*

**imports** *Main*

**begin**

**instantiation** *list* :: (*ord*) *ord*

**begin**

**definition**

*list-less-def*:  $xs < ys \longleftrightarrow (xs, ys) \in \text{lexord } \{(u, v). u < v\}$

**definition**

*list-le-def*:  $(xs :: \text{- list}) \leq ys \longleftrightarrow xs < ys \vee xs = ys$

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

**end**

**instance** *list* :: (*order*) *order*

$\langle \text{proof} \rangle$

**instance** *list* :: (*linorder*) *linorder*

$\langle \text{proof} \rangle$

**instantiation** *list* :: (*linorder*) *distrib-lattice*

**begin**

**definition** (*inf* :: 'a *list*  $\Rightarrow$  -) = *min*

**definition** (*sup* :: 'a *list*  $\Rightarrow$  -) = *max*

**instance**

$\langle \text{proof} \rangle$

**end**

**lemma** *not-less-Nil* [*simp*]:  $\neg x < []$

$\langle \text{proof} \rangle$

**lemma** *Nil-less-Cons* [*simp*]:  $[] < a \# x$

$\langle \text{proof} \rangle$

**lemma** *Cons-less-Cons* [*simp*]:  $a \# x < b \# y \longleftrightarrow a < b \vee a = b \wedge x < y$

$\langle \text{proof} \rangle$

**lemma** *le-Nil* [*simp*]:  $x \leq [] \longleftrightarrow x = []$

$\langle \text{proof} \rangle$

**lemma** *Nil-le-Cons* [*simp*]:  $[] \leq x$   
 ⟨*proof*⟩

**lemma** *Cons-le-Cons* [*simp*]:  $a \# x \leq b \# y \longleftrightarrow a < b \vee a = b \wedge x \leq y$   
 ⟨*proof*⟩

**instantiation** *list* :: (*order*) *order-bot*  
**begin**

**definition** *bot* = []

**instance**  
 ⟨*proof*⟩

**end**

**lemma** *less-list-code* [*code*]:  
 $xs < ([] :: 'a :: \{equal, order\} list) \longleftrightarrow False$   
 $[] < (xs :: 'a :: \{equal, order\}) \# xs \longleftrightarrow True$   
 $(x :: 'a :: \{equal, order\}) \# xs < y \# ys \longleftrightarrow x < y \vee x = y \wedge xs < ys$   
 ⟨*proof*⟩

**lemma** *less-eq-list-code* [*code*]:  
 $x \# xs \leq ([] :: 'a :: \{equal, order\} list) \longleftrightarrow False$   
 $[] \leq (xs :: 'a :: \{equal, order\} list) \longleftrightarrow True$   
 $(x :: 'a :: \{equal, order\}) \# xs \leq y \# ys \longleftrightarrow x < y \vee x = y \wedge xs \leq ys$   
 ⟨*proof*⟩

**end**

## 111 Lexicographic order on lists

This version prioritises length and can yield wellorderings

**theory** *List-Lenlexorder*  
**imports** *Main*  
**begin**

**instantiation** *list* :: (*ord*) *ord*  
**begin**

**definition**  
*list-less-def*:  $xs < ys \longleftrightarrow (xs, ys) \in \text{lenlex } \{(u, v). u < v\}$

**definition**  
*list-le-def*:  $(xs :: - list) \leq ys \longleftrightarrow xs < ys \vee xs = ys$

**instance** ⟨*proof*⟩

**end**

**instance** *list* :: (*order*) *order*  
 ⟨*proof*⟩

**instance** *list* :: (*linorder*) *linorder*  
 ⟨*proof*⟩

**instance** *list* :: (*wellorder*) *wellorder*  
 ⟨*proof*⟩

**instantiation** *list* :: (*linorder*) *distrib-lattice*  
**begin**

**definition** (*inf* :: 'a *list* ⇒ -) = *min*

**definition** (*sup* :: 'a *list* ⇒ -) = *max*

**instance**  
 ⟨*proof*⟩

**end**

**lemma** *not-less-Nil* [*simp*]:  $\neg x < []$   
 ⟨*proof*⟩

**lemma** *Nil-less-Cons* [*simp*]:  $[] < a \# x$   
 ⟨*proof*⟩

**lemma** *Cons-less-Cons*:  $a \# x < b \# y \longleftrightarrow \text{length } x < \text{length } y \vee \text{length } x = \text{length } y \wedge (a < b \vee a = b \wedge x < y)$   
 ⟨*proof*⟩

**lemma** *le-Nil* [*simp*]:  $x \leq [] \longleftrightarrow x = []$   
 ⟨*proof*⟩

**lemma** *Nil-le-Cons* [*simp*]:  $[] \leq x$   
 ⟨*proof*⟩

**lemma** *Cons-le-Cons*:  $a \# x \leq b \# y \longleftrightarrow \text{length } x < \text{length } y \vee \text{length } x = \text{length } y \wedge (a < b \vee a = b \wedge x \leq y)$   
 ⟨*proof*⟩

**instantiation** *list* :: (*order*) *order-bot*  
**begin**

**definition** *bot* = []



**instance**  
 ⟨*proof*⟩

**end**

**end**

## 112 Prefix order on lists as order class instance

**theory** *Prefix-Order*  
**imports** *Sublist*  
**begin**

**instantiation** *list* :: (*type*) *order*  
**begin**

**definition**  $xs \leq ys \equiv \text{prefix } xs \text{ } ys$  **for**  $xs \text{ } ys :: 'a \text{ list}$

**definition**  $xs < ys \equiv xs \leq ys \wedge \neg (ys \leq xs)$  **for**  $xs \text{ } ys :: 'a \text{ list}$

**instance**  
 ⟨*proof*⟩

**end**

**lemma** *less-list-def'*:  $xs < ys \longleftrightarrow \text{strict-prefix } xs \text{ } ys$  **for**  $xs \text{ } ys :: 'a \text{ list}$   
 ⟨*proof*⟩

**lemmas** *prefixI* [*intro?*] = *prefixI* [*folded less-eq-list-def*]

**lemmas** *prefixE* [*elim?*] = *prefixE* [*folded less-eq-list-def*]

**lemmas** *strict-prefixI'* [*intro?*] = *strict-prefixI'* [*folded less-list-def*']

**lemmas** *strict-prefixE'* [*elim?*] = *strict-prefixE'* [*folded less-list-def*']

**lemmas** *strict-prefixI* [*intro?*] = *strict-prefixI* [*folded less-list-def*']

**lemmas** *strict-prefixE* [*elim?*] = *strict-prefixE* [*folded less-list-def*']

**lemmas** *Nil-prefix* [*iff*] = *Nil-prefix* [*folded less-eq-list-def*]

**lemmas** *prefix-Nil* [*simp*] = *prefix-Nil* [*folded less-eq-list-def*]

**lemmas** *prefix-snoc* [*simp*] = *prefix-snoc* [*folded less-eq-list-def*]

**lemmas** *Cons-prefix-Cons* [*simp*] = *Cons-prefix-Cons* [*folded less-eq-list-def*]

**lemmas** *same-prefix-prefix* [*simp*] = *same-prefix-prefix* [*folded less-eq-list-def*]

**lemmas** *same-prefix-nil* [*iff*] = *same-prefix-nil* [*folded less-eq-list-def*]

**lemmas** *prefix-prefix* [*simp*] = *prefix-prefix* [*folded less-eq-list-def*]

**lemmas** *prefix-Cons* = *prefix-Cons* [*folded less-eq-list-def*]

**lemmas** *prefix-length-le* = *prefix-length-le* [*folded less-eq-list-def*]

**lemmas** *strict-prefix-simps* [*simp*, *code*] = *strict-prefix-simps* [*folded less-list-def*']

**lemmas** *not-prefix-induct* [*consumes 1*, *case-names Nil Neq Eq*] =

*not-prefix-induct* [*folded less-eq-list-def*]

**end**

## 113 Lexicographic order on product types

**theory** *Product-Lexorder*

**imports** *Main*

**begin**

**instantiation** *prod* :: (*ord*, *ord*) *ord*

**begin**

**definition**

$$x \leq y \longleftrightarrow \text{fst } x < \text{fst } y \vee \text{fst } x \leq \text{fst } y \wedge \text{snd } x \leq \text{snd } y$$

**definition**

$$x < y \longleftrightarrow \text{fst } x < \text{fst } y \vee \text{fst } x \leq \text{fst } y \wedge \text{snd } x < \text{snd } y$$

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

**end**

**lemma** *less-eq-prod-simp* [*simp*, *code*]:

$$(x1, y1) \leq (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 \leq y2$$

$\langle \text{proof} \rangle$

**lemma** *less-prod-simp* [*simp*, *code*]:

$$(x1, y1) < (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 < y2$$

$\langle \text{proof} \rangle$

A stronger version for partial orders.

**lemma** *less-prod-def'*:

**fixes** *x y* :: '*a*::*order* × '*b*::*ord*

**shows**  $x < y \longleftrightarrow \text{fst } x < \text{fst } y \vee \text{fst } x = \text{fst } y \wedge \text{snd } x < \text{snd } y$

$\langle \text{proof} \rangle$

**instance** *prod* :: (*preorder*, *preorder*) *preorder*

$\langle \text{proof} \rangle$

**instance** *prod* :: (*order*, *order*) *order*

$\langle \text{proof} \rangle$

**instance** *prod* :: (*linorder*, *linorder*) *linorder*

$\langle \text{proof} \rangle$

**instantiation** *prod* :: (*linorder*, *linorder*) *distrib-lattice*

**begin**

**definition**

$$(\text{inf} :: 'a \times 'b \Rightarrow - \Rightarrow -) = \text{min}$$

**definition**

$$(\text{sup} :: 'a \times 'b \Rightarrow - \Rightarrow -) = \text{max}$$

```

instance
  ⟨proof⟩

end

instantiation prod :: (bot, bot) bot
begin

definition
  bot = (bot, bot)

instance ⟨proof⟩

end

instance prod :: (order-bot, order-bot) order-bot
  ⟨proof⟩

instantiation prod :: (top, top) top
begin

definition
  top = (top, top)

instance ⟨proof⟩

end

instance prod :: (order-top, order-top) order-top
  ⟨proof⟩

instance prod :: (wellorder, wellorder) wellorder
  ⟨proof⟩

  Legacy lemma bindings

lemmas prod-le-def = less-eq-prod-def
lemmas prod-less-def = less-prod-def
lemmas prod-less-eq = less-prod-def'

end

```

## 114 Subsequence Ordering

```

theory Subseq-Order
imports Sublist
begin

```

This theory defines subsequence ordering on lists. A list *ys* is a subse-

quence of a list  $xs$ , iff one obtains  $ys$  by erasing some elements from  $xs$ .

### 114.1 Definitions and basic lemmas

**instantiation**  $list :: (type) ord$   
**begin**

**definition**  $less-eq-list$

**where**  $\langle xs \leq ys \longleftrightarrow subseq\ xs\ ys \rangle$  **for**  $xs\ ys :: \langle 'a\ list \rangle$

**definition**  $less-list$

**where**  $\langle xs < ys \longleftrightarrow xs \leq ys \wedge \neg ys \leq xs \rangle$  **for**  $xs\ ys :: \langle 'a\ list \rangle$

**instance**  $\langle proof \rangle$

**end**

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

**lemmas**  $less-eq-list-induct$  [*consumes 1, case-names empty drop take*] =  
 $list-emb.induct$  [*of (=), folded less-eq-list-def*]

**lemma**  $less-eq-list-empty$  [*code*]:

$\langle [] \leq xs \longleftrightarrow True \rangle$   
 $\langle proof \rangle$

**lemma**  $less-eq-list-below-empty$  [*code*]:

$\langle x \# xs \leq [] \longleftrightarrow False \rangle$   
 $\langle proof \rangle$

**lemma**  $le-list-Cons2-iff$  [*simp, code*]:

$\langle x \# xs \leq y \# ys \longleftrightarrow (if\ x = y\ then\ xs \leq ys\ else\ x \# xs \leq ys) \rangle$   
 $\langle proof \rangle$

**lemma**  $less-list-empty$  [*simp*]:

$\langle [] < xs \longleftrightarrow xs \neq [] \rangle$   
 $\langle proof \rangle$

**lemma**  $less-list-empty-Cons$  [*code*]:

$\langle [] < x \# xs \longleftrightarrow True \rangle$   
 $\langle proof \rangle$

**lemma**  $less-list-below-empty$  [*simp, code*]:

$\langle xs < [] \longleftrightarrow False \rangle$   
 $\langle proof \rangle$

**lemma**  $less-list-Cons2-iff$  [*code*]:

$\langle x \# xs < y \# ys \longleftrightarrow (if\ x = y\ then\ xs < ys\ else\ x \# xs \leq ys) \rangle$

*<proof>*

**lemmas** *less-eq-list-drop* = *list-emb.list-emb-Cons* [of (=), folded *less-eq-list-def*]

**lemmas** *le-list-map* = *subseq-map* [folded *less-eq-list-def*]

**lemmas** *le-list-filter* = *subseq-filter* [folded *less-eq-list-def*]

**lemmas** *le-list-length* = *list-emb-length* [of (=), folded *less-eq-list-def*]

**lemma** *less-list-length*:  $xs < ys \implies \text{length } xs < \text{length } ys$

*<proof>*

**lemma** *less-list-drop*:  $xs < ys \implies xs < x \# ys$

*<proof>*

**lemma** *less-list-take-iff*:  $x \# xs < x \# ys \longleftrightarrow xs < ys$

*<proof>*

**lemma** *less-list-drop-many*:  $xs < ys \implies xs < zs @ ys$

*<proof>*

**lemma** *less-list-take-many-iff*:  $zs @ xs < zs @ ys \longleftrightarrow xs < ys$

*<proof>*

**lemma** *less-list-rev-take*:  $xs @ zs < ys @ zs \longleftrightarrow xs < ys$

*<proof>*

end

## 115 Records based on BNF/datatype machinery

**theory** *Datatype-Records*

**imports** *Main*

**keywords** *datatype-record* :: *thy-defn*

**begin**

This theory provides an alternative, stripped-down implementation of records based on the machinery of the **datatype** package.

It supports:

- similar declaration syntax as records
- record creation and update syntax (using `( ... )` brackets)
- regular datatype features (e.g. dead type variables etc.)
- “after-the-fact” registration of single-free-constructor types as records

Caveats:

- there is no compatibility layer; importing this theory will disrupt existing syntax

- extensible records are not supported

**nonterminal**

*ident* **and**  
*field-type* **and**  
*field-types* **and**  
*field* **and**  
*fields* **and**  
*field-update* **and**  
*field-updates*

**open-bundle** *datatype-record-syntax*  
**begin****unbundle** *no record-syntax***syntax**

*-constify*            :: *id* => *ident*                    (⟨-⟩)  
*-constify*            :: *longid* => *ident*                (⟨-⟩)

*-datatype-field*     :: *ident* => '*a*' => *field*            (⟨(⟨indent=2 notation=⟨infix  
*field value*⟩)- =/ -⟩)

                      :: *field* => *fields*                    (⟨-⟩)

*-datatype-fields*     :: *field* => *fields* => *fields*        (⟨-,/ -⟩)

*-datatype-record*    :: *fields* => '*a*'                    (⟨(⟨indent=3 notation=⟨mixfix  
*datatype record value*⟩)(-))⟩)

*-datatype-field-update* :: *ident* => '*a*' => *field-update*    (⟨(⟨indent=2 nota-  
*tion=⟨infix field update*⟩)- :=/ -⟩)

                      :: *field-update* => *field-updates*    (⟨-⟩)

*-datatype-field-updates* :: *field-update* => *field-updates* => *field-updates* (⟨-,/ -⟩)

*-datatype-record-update* :: '*a*' => *field-updates* => '*b*'    (⟨(⟨open-block nota-  
*tion=⟨mixfix datatype record update*⟩)-/(3(-))⟩ [900, 0] 900)

**syntax** (ASCII)

*-datatype-record*     :: *fields* => '*a*'                    (⟨(⟨indent=3 notation=⟨mixfix  
*datatype record value*⟩'(| - |')⟩)

*-datatype-record-update* :: '*a*' => *field-updates* => '*b*'    (⟨(⟨open-block nota-  
*tion=⟨mixfix datatype record update*⟩)-/(3'(| - |')⟩ [900, 0] 900)

**end****named-theorems** *datatype-record-update*

⟨ML⟩

**end**

## 116 Implementation of mappings with Association Lists

```
theory AList-Mapping
  imports AList Mapping
begin
```

```
lift-definition Mapping :: ('a × 'b) list ⇒ ('a, 'b) mapping is map-of <proof>
```

```
code-datatype Mapping
```

```
lemma lookup-Mapping [simp, code]: Mapping.lookup (Mapping xs) = map-of xs
  <proof>
```

```
lemma keys-Mapping [simp, code]: Mapping.keys (Mapping xs) = set (map fst xs)
  <proof>
```

```
lemma empty-Mapping [code]: Mapping.empty = Mapping []
  <proof>
```

```
lemma is-empty-Mapping [code]: Mapping.is-empty (Mapping xs) ⟷ List.null xs
  <proof>
```

```
lemma update-Mapping [code]: Mapping.update k v (Mapping xs) = Mapping (AList.update
k v xs)
  <proof>
```

```
lemma delete-Mapping [code]: Mapping.delete k (Mapping xs) = Mapping (AList.delete
k xs)
  <proof>
```

```
lemma ordered-keys-Mapping [code]:
  Mapping.ordered-keys (Mapping xs) = sort (remdups (map fst xs))
  <proof>
```

```
lemma entries-Mapping [code]:
  Mapping.entries (Mapping xs) = set (AList.clearjunk xs)
  <proof>
```

```
lemma ordered-entries-Mapping [code]:
  Mapping.ordered-entries (Mapping xs) = sort-key fst (AList.clearjunk xs)
  <proof>
```

```
lemma fold-Mapping [code]:
  Mapping.fold f (Mapping xs) a = List.fold (case-prod f) (sort-key fst (AList.clearjunk
xs)) a
  <proof>
```

```
lemma size-Mapping [code]: Mapping.size (Mapping xs) = length (remdups (map
```

*fst xs*)  
 ⟨proof⟩

**lemma** *tabulate-Mapping* [code]: *Mapping.tabulate ks f = Mapping (map (λk. (k, f k)) ks)*  
 ⟨proof⟩

**lemma** *bulkload-Mapping* [code]:  
*Mapping.bulkload vs = Mapping (map (λn. (n, vs ! n)) [0..<length vs])*  
 ⟨proof⟩

**lemma** *equal-Mapping* [code]:  
*HOL.equal (Mapping xs) (Mapping ys) ⟷*  
*(let ks = map fst xs; ls = map fst ys*  
*in (∀ l ∈ set ls. l ∈ set ks) ∧ (∀ k ∈ set ks. k ∈ set ls ∧ map-of xs k = map-of ys k))*  
 ⟨proof⟩

**lemma** *map-values-Mapping* [code]:  
*Mapping.map-values f (Mapping xs) = Mapping (map (λ(x,y). (x, f x y)) xs)*  
**for** *f :: 'c ⇒ 'a ⇒ 'b and xs :: ('c × 'a) list*  
 ⟨proof⟩

**lemma** *combine-with-key-code* [code]:  
*Mapping.combine-with-key f (Mapping xs) (Mapping ys) =*  
*Mapping.tabulate (remdups (map fst xs @ map fst ys))*  
*(λx. the (combine-options (f x) (map-of xs x) (map-of ys x)))*  
 ⟨proof⟩

**lemma** *combine-code* [code]:  
*Mapping.combine f (Mapping xs) (Mapping ys) =*  
*Mapping.tabulate (remdups (map fst xs @ map fst ys))*  
*(λx. the (combine-options f (map-of xs x) (map-of ys x)))*  
 ⟨proof⟩

**lemma** *map-of-filter-distinct*:  
**assumes** *distinct (map fst xs)*  
**shows** *map-of (filter P xs) x =*  
*(case map-of xs x of*  
*None ⇒ None*  
*| Some y ⇒ if P (x,y) then Some y else None)*  
 ⟨proof⟩

**lemma** *filter-Mapping* [code]:  
*Mapping.filter P (Mapping xs) = Mapping (filter (λ(k,v). P k v) (AList.clearjunk xs))*  
 ⟨proof⟩

**lemma** [code nbe]: *HOL.equal (x :: ('a, 'b) mapping) x ⟷ True*



⟨proof⟩

end

**theory** *Code-Abstract-Char*

**imports**

*Main*

*HOL-Library.Char-ord*

**begin**

**definition** *Chr* :: ⟨integer ⇒ char⟩

**where** [*simp*]: ⟨*Chr* = *char-of*⟩

**lemma** *char-of-integer-of-char* [*code abstype*]:

⟨*Chr* (*integer-of-char* *c*) = *c*⟩

⟨proof⟩

**lemma** *char-of-integer-code* [*code*]:

⟨*integer-of-char* (*char-of-integer* *k*) = (if  $0 \leq k \wedge k < 256$  then *k* else *k mod 256*)⟩

⟨proof⟩

**lemma** *of-char-code* [*code*]:

⟨*of-char* *c* = *of-nat* (*nat-of-integer* (*integer-of-char* *c*))⟩

⟨proof⟩

**definition** *byte* :: ⟨bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ integer⟩

**where** [*simp*]: ⟨*byte* *b0* *b1* *b2* *b3* *b4* *b5* *b6* *b7* = *horner-sum of-bool 2* [*b0*, *b1*, *b2*, *b3*, *b4*, *b5*, *b6*, *b7*]⟩

**lemma** *byte-code* [*code*]:

⟨*byte* *b0* *b1* *b2* *b3* *b4* *b5* *b6* *b7* = (

let

*s0* = if *b0* then 1 else 0;

*s1* = if *b1* then *s0* + 2 else *s0*;

*s2* = if *b2* then *s1* + 4 else *s1*;

*s3* = if *b3* then *s2* + 8 else *s2*;

*s4* = if *b4* then *s3* + 16 else *s3*;

*s5* = if *b5* then *s4* + 32 else *s4*;

*s6* = if *b6* then *s5* + 64 else *s5*;

*s7* = if *b7* then *s6* + 128 else *s6*

in *s7*)⟩

⟨proof⟩

**lemma** *Char-code* [*code*]:

⟨*integer-of-char* (*Char* *b0* *b1* *b2* *b3* *b4* *b5* *b6* *b7*) = *byte* *b0* *b1* *b2* *b3* *b4* *b5* *b6* *b7*⟩

⟨proof⟩

**lemma** *digit-0-code* [code]:

⟨*digit0*  $c \longleftrightarrow \text{bit } (\text{integer-of-char } c) 0$ ⟩  
 ⟨*proof*⟩

**lemma** *digit-1-code* [code]:

⟨*digit1*  $c \longleftrightarrow \text{bit } (\text{integer-of-char } c) 1$ ⟩  
 ⟨*proof*⟩

**lemma** *digit-2-code* [code]:

⟨*digit2*  $c \longleftrightarrow \text{bit } (\text{integer-of-char } c) 2$ ⟩  
 ⟨*proof*⟩

**lemma** *digit-3-code* [code]:

⟨*digit3*  $c \longleftrightarrow \text{bit } (\text{integer-of-char } c) 3$ ⟩  
 ⟨*proof*⟩

**lemma** *digit-4-code* [code]:

⟨*digit4*  $c \longleftrightarrow \text{bit } (\text{integer-of-char } c) 4$ ⟩  
 ⟨*proof*⟩

**lemma** *digit-5-code* [code]:

⟨*digit5*  $c \longleftrightarrow \text{bit } (\text{integer-of-char } c) 5$ ⟩  
 ⟨*proof*⟩

**lemma** *digit-6-code* [code]:

⟨*digit6*  $c \longleftrightarrow \text{bit } (\text{integer-of-char } c) 6$ ⟩  
 ⟨*proof*⟩

**lemma** *digit-7-code* [code]:

⟨*digit7*  $c \longleftrightarrow \text{bit } (\text{integer-of-char } c) 7$ ⟩  
 ⟨*proof*⟩

**lemma** *case-char-code* [code]:

⟨*case-char*  $f c = f (\text{digit0 } c) (\text{digit1 } c) (\text{digit2 } c) (\text{digit3 } c) (\text{digit4 } c) (\text{digit5 } c)$   
 $(\text{digit6 } c) (\text{digit7 } c)$ ⟩  
 ⟨*proof*⟩

**lemma** *rec-char-code* [code]:

⟨*rec-char*  $f c = f (\text{digit0 } c) (\text{digit1 } c) (\text{digit2 } c) (\text{digit3 } c) (\text{digit4 } c) (\text{digit5 } c)$   
 $(\text{digit6 } c) (\text{digit7 } c)$ ⟩  
 ⟨*proof*⟩

**lemma** *char-of-code* [code]:

⟨*integer-of-char* (*char-of*  $a$ ) =  
 $\text{byte } (\text{bit } a 0) (\text{bit } a 1) (\text{bit } a 2) (\text{bit } a 3) (\text{bit } a 4) (\text{bit } a 5) (\text{bit } a 6) (\text{bit } a 7)$ ⟩  
 ⟨*proof*⟩

**lemma** *ascii-of-code* [code]:

```

  ⟨integer-of-char (String.ascii-of c) = (let k = integer-of-char c in if k < 128 then
  k else k - 128)⟩
  ⟨proof⟩

```

```

lemma equal-char-code [code]:
  ⟨HOL.equal c d ⟷ integer-of-char c = integer-of-char d⟩
  ⟨proof⟩

```

```

lemma less-eq-char-code [code]:
  ⟨c ≤ d ⟷ integer-of-char c ≤ integer-of-char d⟩ (is ⟨?P ⟷ ?Q⟩)
  ⟨proof⟩

```

```

lemma less-char-code [code]:
  ⟨c < d ⟷ integer-of-char c < integer-of-char d⟩ (is ⟨?P ⟷ ?Q⟩)
  ⟨proof⟩

```

```

lemma absdef-simps:
  ⟨horner-sum of-bool 2 [] = (0 :: integer)⟩
  ⟨horner-sum of-bool 2 (False # bs) = (0 :: integer) ⟷ horner-sum of-bool 2 bs
  = (0 :: integer)⟩
  ⟨horner-sum of-bool 2 (True # bs) = (1 :: integer) ⟷ horner-sum of-bool 2 bs
  = (0 :: integer)⟩
  ⟨horner-sum of-bool 2 (False # bs) = (numeral (Num.Bit0 n) :: integer) ⟷
  horner-sum of-bool 2 bs = (numeral n :: integer)⟩
  ⟨horner-sum of-bool 2 (True # bs) = (numeral (Num.Bit1 n) :: integer) ⟷
  horner-sum of-bool 2 bs = (numeral n :: integer)⟩
  ⟨proof⟩

```

```

⟨ML⟩

```

```

code-identifier

```

```

code-module Code-Abstract-Char ↯
  (SML) Str and (OCaml) Str and (Haskell) Str and (Scala) Str

```

```

end

```

## 117 Avoidance of pattern matching on natural numbers

```

theory Code-Abstract-Nat
imports Main
begin

```

When natural numbers are implemented in another than the conventional inductive  $0/Suc$  representation, it is necessary to avoid all pattern matching on natural numbers altogether. This is accomplished by this theory (up to a certain extent).

### 117.1 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

**lemma** *[code, code-unfold]*:  
 $case\text{-}nat = (\lambda f\ g\ n.\ if\ n = 0\ then\ f\ else\ g\ (n - 1))$   
*<proof>*

### 117.2 Preprocessors

The term *Suc n* is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a code equation) must be eliminated. This can be accomplished – as far as possible – by applying the following transformation rule:

**lemma** *Suc-if-eq*:  
**assumes**  $\bigwedge n.\ f\ (Suc\ n) \equiv h\ n$   
**assumes**  $f\ 0 \equiv g$   
**shows**  $f\ n \equiv if\ n = 0\ then\ g\ else\ h\ (n - 1)$   
*<proof>*

The rule above is built into a preprocessor that is plugged into the code generator.

*<ML>*

### 117.3 Candidates which need special treatment

**lemma** *drop-bit-int-code* *[code]*:  
 $\langle drop\text{-}bit\ n\ k = k\ div\ 2^{\wedge} n \rangle\ \mathbf{for}\ k :: int$   
*<proof>*

**lemma** *take-bit-num-code* *[code]*:  
 $\langle take\text{-}bit\text{-}num\ n\ Num.One =$   
 $(case\ n\ of\ 0 \Rightarrow None\ | Suc\ n \Rightarrow Some\ Num.One) \rangle$   
 $\langle take\text{-}bit\text{-}num\ n\ (Num.Bit0\ m) =$   
 $(case\ n\ of\ 0 \Rightarrow None\ | Suc\ n \Rightarrow (case\ take\text{-}bit\text{-}num\ n\ m\ of\ None \Rightarrow None\ |$   
 $Some\ q \Rightarrow Some\ (Num.Bit0\ q))) \rangle$   
 $\langle take\text{-}bit\text{-}num\ n\ (Num.Bit1\ m) =$   
 $(case\ n\ of\ 0 \Rightarrow None\ | Suc\ n \Rightarrow Some\ (case\ take\text{-}bit\text{-}num\ n\ m\ of\ None \Rightarrow$   
 $Num.One\ | Some\ q \Rightarrow Num.Bit1\ q)) \rangle$   
*<proof>*

**end**

## 118 Implementation of natural numbers as binary numerals

**theory** *Code-Binary-Nat*

```
imports Code-Abstract-Nat
begin
```

When generating code for functions on natural numbers, the canonical representation using  $0$  and  $Suc$  is unsuitable for computations involving large numbers. This theory refines the representation of natural numbers for code generation to use binary numerals, which do not grow linear in size but logarithmic.

### 118.1 Representation

```
code-datatype 0::nat nat-of-num
```

```
lemma [code]:
  num-of-nat 0 = Num.One
  num-of-nat (nat-of-num k) = k
  ⟨proof⟩
```

```
lemma [code]:
  (1::nat) = Numeral1
  ⟨proof⟩
```

```
lemma [code-abbrev]: Numeral1 = (1::nat)
  ⟨proof⟩
```

```
lemma [code]:
  Suc n = n + 1
  ⟨proof⟩
```

### 118.2 Basic arithmetic

```
context
begin
```

```
declare [[code drop: plus :: nat ⇒ -]]
```

```
lemma plus-nat-code [code]:
  nat-of-num k + nat-of-num l = nat-of-num (k + l)
  m + 0 = (m::nat)
  0 + n = (n::nat)
  ⟨proof⟩
```

Bounded subtraction needs some auxiliary

```
qualified definition dup :: nat ⇒ nat where
  dup n = n + n
```

```
lemma dup-code [code]:
  dup 0 = 0
  dup (nat-of-num k) = nat-of-num (Num.Bit0 k)
```

*<proof>* **definition** *sub* :: *num*  $\Rightarrow$  *num*  $\Rightarrow$  *nat option* **where**  
*sub* *k l* = (if *k*  $\geq$  *l* then *Some* (*numeral k* - *numeral l*) else *None*)

**lemma** *sub-code* [*code*]:  
*sub* *Num.One Num.One* = *Some 0*  
*sub* (*Num.Bit0 m*) *Num.One* = *Some* (*nat-of-num* (*Num.BitM m*))  
*sub* (*Num.Bit1 m*) *Num.One* = *Some* (*nat-of-num* (*Num.Bit0 m*))  
*sub* *Num.One* (*Num.Bit0 n*) = *None*  
*sub* *Num.One* (*Num.Bit1 n*) = *None*  
*sub* (*Num.Bit0 m*) (*Num.Bit0 n*) = *map-option dup* (*sub m n*)  
*sub* (*Num.Bit1 m*) (*Num.Bit1 n*) = *map-option dup* (*sub m n*)  
*sub* (*Num.Bit1 m*) (*Num.Bit0 n*) = *map-option* ( $\lambda q. \text{dup } q + 1$ ) (*sub m n*)  
*sub* (*Num.Bit0 m*) (*Num.Bit1 n*) = (*case sub m n of None*  $\Rightarrow$  *None*  
| *Some q*  $\Rightarrow$  if *q* = 0 then *None* else *Some* (*dup q* - 1))  
*<proof>*

**declare** [[*code drop: minus* :: *nat*  $\Rightarrow$  -]]

**lemma** *minus-nat-code* [*code*]:  
*nat-of-num k* - *nat-of-num l* = (*case sub k l of None*  $\Rightarrow$  0 | *Some j*  $\Rightarrow$  *j*)  
*m* - 0 = (*m::nat*)  
0 - *n* = (*0::nat*)  
*<proof>*

**declare** [[*code drop: times* :: *nat*  $\Rightarrow$  -]]

**lemma** *times-nat-code* [*code*]:  
*nat-of-num k* \* *nat-of-num l* = *nat-of-num* (*k* \* *l*)  
*m* \* 0 = (*0::nat*)  
0 \* *n* = (*0::nat*)  
*<proof>*

**declare** [[*code drop: HOL.equal* :: *nat*  $\Rightarrow$  -]]

**lemma** *equal-nat-code* [*code*]:  
*HOL.equal* 0 (*0::nat*)  $\longleftrightarrow$  *True*  
*HOL.equal* 0 (*nat-of-num l*)  $\longleftrightarrow$  *False*  
*HOL.equal* (*nat-of-num k*) 0  $\longleftrightarrow$  *False*  
*HOL.equal* (*nat-of-num k*) (*nat-of-num l*)  $\longleftrightarrow$  *HOL.equal k l*  
*<proof>*

**lemma** *equal-nat-refl* [*code nbe*]:  
*HOL.equal* (*n::nat*) *n*  $\longleftrightarrow$  *True*  
*<proof>*

**declare** [[*code drop: less-eq* :: *nat*  $\Rightarrow$  -]]

**lemma** *less-eq-nat-code* [*code*]:  
0  $\leq$  (*n::nat*)  $\longleftrightarrow$  *True*

```

nat-of-num k ≤ 0 ↔ False
nat-of-num k ≤ nat-of-num l ↔ k ≤ l
⟨proof⟩

```

```

declare [[code drop: less :: nat ⇒ -]]

```

```

lemma less-nat-code [code]:
  (m::nat) < 0 ↔ False
  0 < nat-of-num l ↔ True
  nat-of-num k < nat-of-num l ↔ k < l
  ⟨proof⟩

```

```

declare [[code drop: Euclidean-Rings.divmod-nat]]

```

```

lemma divmod-nat-code [code]:
  Euclidean-Rings.divmod-nat (nat-of-num k) (nat-of-num l) = divmod k l
  Euclidean-Rings.divmod-nat m 0 = (0, m)
  Euclidean-Rings.divmod-nat 0 n = (0, 0)
  ⟨proof⟩

```

```

end

```

### 118.3 Conversions

```

declare [[code drop: of-nat]]

```

```

lemma of-nat-code [code]:
  of-nat 0 = 0
  of-nat (nat-of-num k) = numeral k
  ⟨proof⟩

```

```

code-identifier

```

```

code-module Code-Binary-Nat ↪
  (SML) Arith and (OCaml) Arith and (Haskell) Arith

```

```

end

```

## 119 Code generation of prolog programs

```

theory Code-Prolog
imports Main
keywords values-prolog :: diag
begin

⟨ML⟩

```

## 120 Setup for Numerals

$\langle ML \rangle$

end

## 121 Implementation of integer numbers by target-language integers

**theory** *Code-Target-Int*

**imports** *Main*

**begin**

**code-datatype** *int-of-integer*

**declare**  $[[code\ drop:\ integer\ of\ int]]$

**context**

**includes** *integer.lifting*

**begin**

**lemma** [*code*]:

$integer\ of\ int\ (int\ of\ integer\ k) = k$

$\langle proof \rangle$

**lemma** [*code*]:

$Int.Pos = int\ of\ integer \circ integer\ of\ num$

$\langle proof \rangle$

**lemma** [*code*]:

$Int.Neg = int\ of\ integer \circ uminus \circ integer\ of\ num$

$\langle proof \rangle$

**lemma** [*code-abbrev*]:

$int\ of\ integer\ (numeral\ k) = Int.Pos\ k$

$\langle proof \rangle$

**lemma** [*code-abbrev*]:

$int\ of\ integer\ (-\ numeral\ k) = Int.Neg\ k$

$\langle proof \rangle$

**context**

**begin**

**qualified definition** *positive* :: *num*  $\Rightarrow$  *int*

**where** [*simp*]: *positive* = *numeral*

**qualified definition** *negative* :: *num*  $\Rightarrow$  *int*



**where** [*simp*]:  $negative = uminus \circ numeral$

**lemma** [*code-computation-unfold*]:

$numeral = positive$

$Int.Pos = positive$

$Int.Neg = negative$

$\langle proof \rangle$

**end**

**lemma** [*code, symmetric, code-post*]:

$0 = int-of-integer\ 0$

$\langle proof \rangle$

**lemma** [*code, symmetric, code-post*]:

$1 = int-of-integer\ 1$

$\langle proof \rangle$

**lemma** [*code-post*]:

$int-of-integer\ (-\ 1) = -\ 1$

$\langle proof \rangle$

**lemma** [*code*]:

$k + l = int-of-integer\ (of-int\ k + of-int\ l)$

$\langle proof \rangle$

**lemma** [*code*]:

$-k = int-of-integer\ (-\ of-int\ k)$

$\langle proof \rangle$

**lemma** [*code*]:

$k - l = int-of-integer\ (of-int\ k - of-int\ l)$

$\langle proof \rangle$

**lemma** [*code*]:

$Int.dup\ k = int-of-integer\ (Code-Numeral.dup\ (of-int\ k))$

$\langle proof \rangle$

**declare** [[*code drop: Int.sub*]]

**lemma** [*code*]:

$k * l = int-of-integer\ (of-int\ k * of-int\ l)$

$\langle proof \rangle$

**lemma** [*code*]:

$k\ div\ l = int-of-integer\ (of-int\ k\ div\ of-int\ l)$

$\langle proof \rangle$

**lemma** [*code*]:

$k \bmod l = \text{int-of-integer } (\text{of-int } k \bmod \text{of-int } l)$   
 ⟨proof⟩

**lemma** [code]:  
 $\text{divmod } m \ n = \text{map-prod } \text{int-of-integer } \text{int-of-integer } (\text{divmod } m \ n)$   
 ⟨proof⟩

**lemma** [code]:  
 $\text{HOL.equal } k \ l = \text{HOL.equal } (\text{of-int } k :: \text{integer}) (\text{of-int } l)$   
 ⟨proof⟩

**lemma** [code]:  
 $k \leq l \iff (\text{of-int } k :: \text{integer}) \leq \text{of-int } l$   
 ⟨proof⟩

**lemma** [code]:  
 $k < l \iff (\text{of-int } k :: \text{integer}) < \text{of-int } l$   
 ⟨proof⟩

**declare** [[code drop: gcd :: int ⇒ - lcm :: int ⇒ -]]

**lemma** gcd-int-of-integer [code]:  
 $\text{gcd } (\text{int-of-integer } x) (\text{int-of-integer } y) = \text{int-of-integer } (\text{gcd } x \ y)$   
 ⟨proof⟩

**lemma** lcm-int-of-integer [code]:  
 $\text{lcm } (\text{int-of-integer } x) (\text{int-of-integer } y) = \text{int-of-integer } (\text{lcm } x \ y)$   
 ⟨proof⟩

**end**

**lemma** (in ring-1) of-int-code-if:  
 $\text{of-int } k = (\text{if } k = 0 \text{ then } 0$   
    $\text{else if } k < 0 \text{ then } - \text{of-int } (- k)$   
    $\text{else let}$   
      $l = 2 * \text{of-int } (k \ \text{div } 2);$   
      $j = k \ \text{mod } 2$   
    $\text{in if } j = 0 \text{ then } l \text{ else } l + 1)$   
 ⟨proof⟩

**declare** of-int-code-if [code]

**lemma** [code]:  
 $\text{nat} = \text{nat-of-integer} \circ \text{of-int}$   
**including** integer.lifting ⟨proof⟩

**definition** char-of-int :: int ⇒ char  
**where** [code-abbrev]: char-of-int = char-of

**definition** *int-of-char* :: *char*  $\Rightarrow$  *int*  
**where** [*code-abbrev*]: *int-of-char* = *of-char*

**lemma** [*code*]:  
*char-of-int* = *char-of-integer*  $\circ$  *integer-of-int*  
**including** *integer.lifting*  $\langle$ *proof* $\rangle$

**lemma** [*code*]:  
*int-of-char* = *int-of-integer*  $\circ$  *integer-of-char*  
**including** *integer.lifting*  $\langle$ *proof* $\rangle$

**context**  
**includes** *integer.lifting* **and** *bit-operations-syntax*  
**begin**

**declare** [[*code drop*:  $\langle$ *bit* :: *int*  $\Rightarrow$   $\rightarrow$   $\langle$ *not* :: *int*  $\Rightarrow$   $\rightarrow$   
 $\langle$ *and* :: *int*  $\Rightarrow$   $\rightarrow$   $\langle$ *or* :: *int*  $\Rightarrow$   $\rightarrow$   $\langle$ *xor* :: *int*  $\Rightarrow$   $\rightarrow$   
 $\langle$ *push-bit* ::  $- \Rightarrow - \Rightarrow$  *int*  $\langle$ *drop-bit* ::  $- \Rightarrow - \Rightarrow$  *int*  $\langle$ *take-bit* ::  $- \Rightarrow - \Rightarrow$  *int* $\rangle$ ]]

**lemma** [*code*]:  
 $\langle$ *bit* (*int-of-integer* *k*) *n*  $\longleftrightarrow$  *bit* *k* *n* $\rangle$   
 $\langle$ *proof* $\rangle$

**lemma** [*code*]:  
 $\langle$ *NOT* (*int-of-integer* *k*) = *int-of-integer* (*NOT* *k*) $\rangle$   
 $\langle$ *proof* $\rangle$

**lemma** [*code*]:  
 $\langle$ *int-of-integer* *k* *AND* *int-of-integer* *l* = *int-of-integer* (*k* *AND* *l*) $\rangle$   
 $\langle$ *proof* $\rangle$

**lemma** [*code*]:  
 $\langle$ *int-of-integer* *k* *OR* *int-of-integer* *l* = *int-of-integer* (*k* *OR* *l*) $\rangle$   
 $\langle$ *proof* $\rangle$

**lemma** [*code*]:  
 $\langle$ *int-of-integer* *k* *XOR* *int-of-integer* *l* = *int-of-integer* (*k* *XOR* *l*) $\rangle$   
 $\langle$ *proof* $\rangle$

**lemma** [*code*]:  
 $\langle$ *push-bit* *n* (*int-of-integer* *k*) = *int-of-integer* (*push-bit* *n* *k*) $\rangle$   
 $\langle$ *proof* $\rangle$

**lemma** [*code*]:  
 $\langle$ *drop-bit* *n* (*int-of-integer* *k*) = *int-of-integer* (*drop-bit* *n* *k*) $\rangle$   
 $\langle$ *proof* $\rangle$

**lemma** [*code*]:  
 $\langle$ *take-bit* *n* (*int-of-integer* *k*) = *int-of-integer* (*take-bit* *n* *k*) $\rangle$

⟨*proof*⟩

**lemma** [*code*]:

⟨*mask n = int-of-integer (mask n)*⟩

⟨*proof*⟩

**lemma** [*code*]:

⟨*set-bit n (int-of-integer k) = int-of-integer (set-bit n k)*⟩

⟨*proof*⟩

**lemma** [*code*]:

⟨*unset-bit n (int-of-integer k) = int-of-integer (unset-bit n k)*⟩

⟨*proof*⟩

**lemma** [*code*]:

⟨*flip-bit n (int-of-integer k) = int-of-integer (flip-bit n k)*⟩

⟨*proof*⟩

**end**

**code-identifier**

**code-module** *Code-Target-Int*  $\rightarrow$

(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

**end**

**theory** *Code-Real-Approx-By-Float*

**imports** *Complex-Main* *Code-Target-Int*

**begin**

**WARNING!** This theory implements mathematical reals by machine reals (floats). This is inconsistent. See the proof of `False` at the end of the theory, where an equality on mathematical reals is (incorrectly) disproved by mapping it to machine reals.

The **value** command cannot display real results yet.

The only legitimate use of this theory is as a tool for code generation purposes.

**context**

**begin**

**qualified definition** *real-of-integer* :: *integer*  $\Rightarrow$  *real*

**where** [*code-abbrev*]: *real-of-integer* = *of-int*  $\circ$  *int-of-integer*

**end**

**code-datatype** *Code-Real-Approx-By-Float.real-of-integer*  $\langle (/) \rangle$  :: *real*  $\Rightarrow$  *real*  $\Rightarrow$  *real*

**lemma** [*code-unfold del*]: *numeral k*  $\equiv$  *real-of-rat (numeral k)*  
 ⟨*proof*⟩

**lemma** [*code-unfold del*]:  $-$  *numeral k*  $\equiv$  *real-of-rat (- numeral k)*  
 ⟨*proof*⟩

**context**  
**begin**

**qualified definition** *real-of-int* :: ⟨*int*  $\Rightarrow$  *real*⟩  
**where** [*code-abbrev*]: ⟨*real-of-int = of-int*⟩

**lemma** [*code*]: *real-of-int = Code-Real-Approx-By-Float.real-of-integer*  $\circ$  *integer-of-int*  
 ⟨*proof*⟩ **definition** *exp-real* :: ⟨*real*  $\Rightarrow$  *real*⟩  
**where** [*code-abbrev, code del*]: ⟨*exp-real = exp*⟩

**qualified definition** *sin-real* :: ⟨*real*  $\Rightarrow$  *real*⟩  
**where** [*code-abbrev, code del*]: ⟨*sin-real = sin*⟩

**qualified definition** *cos-real* :: ⟨*real*  $\Rightarrow$  *real*⟩  
**where** [*code-abbrev, code del*]: ⟨*cos-real = cos*⟩

**qualified definition** *tan-real* :: ⟨*real*  $\Rightarrow$  *real*⟩  
**where** [*code-abbrev, code del*]: ⟨*tan-real = tan*⟩

**end**

**lemma** [*code*]: ⟨*Ratreal r = (case quotient-of r of (p, q)  $\Rightarrow$  real-of-int p / real-of-int q)*⟩  
 ⟨*proof*⟩

**lemma** [*code*]: ⟨*inverse r = 1 / r*⟩ **for** *r* :: *real*  
 ⟨*proof*⟩

**declare** [[*code drop: <HOL.equal :: real  $\Rightarrow$  real  $\Rightarrow$  bool*⟩  
 ⟨*( $\leq$ ) :: real  $\Rightarrow$  real  $\Rightarrow$  bool*⟩  
 ⟨*( $<$ ) :: real  $\Rightarrow$  real  $\Rightarrow$  bool*⟩  
 ⟨*plus :: real  $\Rightarrow$  real  $\Rightarrow$  real*⟩  
 ⟨*times :: real  $\Rightarrow$  real  $\Rightarrow$  real*⟩  
 ⟨*uminus :: real  $\Rightarrow$  real*⟩  
 ⟨*minus :: real  $\Rightarrow$  real  $\Rightarrow$  real*⟩  
 ⟨*divide :: real  $\Rightarrow$  real  $\Rightarrow$  real*⟩  
*sqrt*  
 ⟨*ln :: real  $\Rightarrow$  real*⟩  
*pi*  
*arcsin*  
*arccos*  
*arctan*]]

**code-reserved** (*SML*) *Real*

**code-printing**

```

type-constructor real  $\rightarrow$ 
  (SML) real
  and (OCaml) float
  and (Haskell) Prelude.Double
| constant 0 :: real  $\rightarrow$ 
  (SML) 0.0
  and (OCaml) 0.0
  and (Haskell) 0.0
| constant 1 :: real  $\rightarrow$ 
  (SML) 1.0
  and (OCaml) 1.0
  and (Haskell) 1.0
| constant HOL.equal :: real  $\Rightarrow$  real  $\Rightarrow$  bool  $\rightarrow$ 
  (SML) Real.== ((-), (-))
  and (OCaml) Pervasives.(=)
  and (Haskell) infix 4 ==
| class-instance real :: HOL.equal  $\Rightarrow$  (Haskell)  $-$ 
| constant ( $\leq$ ) :: real  $\Rightarrow$  real  $\Rightarrow$  bool  $\rightarrow$ 
  (SML) Real.<= ((-), (-))
  and (OCaml) Pervasives.(<=)
  and (Haskell) infix 4 <=
| constant ( $<$ ) :: real  $\Rightarrow$  real  $\Rightarrow$  bool  $\rightarrow$ 
  (SML) Real.< ((-), (-))
  and (OCaml) Pervasives.<
  and (Haskell) infix 4 <
| constant ( $+$ ) :: real  $\Rightarrow$  real  $\Rightarrow$  real  $\rightarrow$ 
  (SML) Real.+ ((-), (-))
  and (OCaml) Pervasives.(+.)
  and (Haskell) infixl 6 +
| constant ( $*$ ) :: real  $\Rightarrow$  real  $\Rightarrow$  real  $\rightarrow$ 
  (SML) Real.* ((-), (-))
  and (Haskell) infixl 7 *
| constant uminus :: real  $\Rightarrow$  real  $\rightarrow$ 
  (SML) Real.~
  and (OCaml) Pervasives.(~-.)
  and (Haskell) negate
| constant ( $-$ ) :: real  $\Rightarrow$  real  $\Rightarrow$  real  $\rightarrow$ 
  (SML) Real.- ((-), (-))
  and (OCaml) Pervasives.(-.)
  and (Haskell) infixl 6 -
| constant ( $/$ ) :: real  $\Rightarrow$  real  $\Rightarrow$  real  $\rightarrow$ 
  (SML) Real.'/ ((-), (-))
  and (OCaml) Pervasives.( '/. )
  and (Haskell) infixl 7 /
| constant sqrt :: real  $\Rightarrow$  real  $\rightarrow$ 
  (SML) Math.sqrt

```

```

    and (OCaml) Pervasives.sqrt
    and (Haskell) Prelude.sqrt
| constant Code-Real-Approx-By-Float.exp-real →
  (SML) Math.exp
  and (OCaml) Pervasives.exp
  and (Haskell) Prelude.exp
| constant ln →
  (SML) Math.ln
  and (OCaml) Pervasives.ln
  and (Haskell) Prelude.log
| constant Code-Real-Approx-By-Float.sin-real →
  (SML) Math.sin
  and (OCaml) Pervasives.sin
  and (Haskell) Prelude.sin
| constant Code-Real-Approx-By-Float.cos-real →
  (SML) Math.cos
  and (OCaml) Pervasives.cos
  and (Haskell) Prelude.cos
| constant Code-Real-Approx-By-Float.tan-real →
  (SML) Math.tan
  and (OCaml) Pervasives.tan
  and (Haskell) Prelude.tan
| constant pi →
  (SML) Math.pi

  and (Haskell) Prelude.pi
| constant arcsin →
  (SML) Math.asin
  and (OCaml) Pervasives.asin
  and (Haskell) Prelude.asin
| constant arccos →
  (SML) Math.scos
  and (OCaml) Pervasives.acos
  and (Haskell) Prelude.acos
| constant arctan →
  (SML) Math.atan
  and (OCaml) Pervasives.atan
  and (Haskell) Prelude.atan
| constant Code-Real-Approx-By-Float.real-of-integer →
  (SML) Real.fromInt
  and (OCaml) Pervasives.float/ (Big'-int.to'-int (-))
  and (Haskell) Prelude.fromIntegral (-)

notepad
begin
  ⟨proof⟩
end

end

```

## 122 Implementation of natural numbers by target-language integers

```
theory Code-Target-Nat
imports Code-Abstract-Nat
begin
```

### 122.1 Implementation for *nat*

```
context
includes natural.lifting and integer.lifting
begin
```

```
lift-definition Nat :: integer  $\Rightarrow$  nat
is nat
<proof>
```

```
lemma [code-post]:
  Nat 0 = 0
  Nat 1 = 1
  Nat (numeral k) = numeral k
  <proof>
```

```
lemma [code-abbrev]:
  integer-of-nat = of-nat
  <proof>
```

```
lemma [code-unfold]:
  Int.nat (int-of-integer k) = nat-of-integer k
  <proof>
```

```
lemma [code abstype]:
  Code-Target-Nat.Nat (integer-of-nat n) = n
  <proof>
```

```
lemma [code abstract]:
  integer-of-nat (nat-of-integer k) = max 0 k
  <proof>
```

```
lemma [code-abbrev]:
  nat-of-integer (numeral k) = nat-of-num k
  <proof>
```

```
context
begin
```

```
qualified definition natural :: num  $\Rightarrow$  nat
  where [simp]: natural = nat-of-num
```



```

lemma [code-computation-unfold]:
  numeral = natural
  nat-of-num = natural
  ⟨proof⟩

end

lemma [code abstract]:
  integer-of-nat (nat-of-num n) = integer-of-num n
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat 0 = 0
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat 1 = 1
  ⟨proof⟩

lemma [code]:
  Suc n = n + 1
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat (m + n) = of-nat m + of-nat n
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat (m - n) = max 0 (of-nat m - of-nat n)
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat (m * n) = of-nat m * of-nat n
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat (m div n) = of-nat m div of-nat n
  ⟨proof⟩

lemma [code abstract]:
  integer-of-nat (m mod n) = of-nat m mod of-nat n
  ⟨proof⟩

context
  includes integer.lifting
begin

lemma divmod-nat-code [code]:
  Euclidean-Rings.divmod-nat m n = (

```

```

    let k = integer-of-nat m; l = integer-of-nat n
    in map-prod nat-of-integer nat-of-integer
      (if k = 0 then (0, 0)
       else if l = 0 then (0, k) else
         Code-Numeral.divmod-abs k l)
    ⟨proof⟩

```

**end**

**lemma** [code]:  
 $\text{divmod } m \ n = \text{map-prod nat-of-integer nat-of-integer } (\text{divmod } m \ n)$   
 ⟨proof⟩

**lemma** [code]:  
 $\text{HOL.equal } m \ n = \text{HOL.equal } (\text{of-nat } m :: \text{integer}) \ (\text{of-nat } n)$   
 ⟨proof⟩

**lemma** [code]:  
 $m \leq n \iff (\text{of-nat } m :: \text{integer}) \leq \text{of-nat } n$   
 ⟨proof⟩

**lemma** [code]:  
 $m < n \iff (\text{of-nat } m :: \text{integer}) < \text{of-nat } n$   
 ⟨proof⟩

**lemma** num-of-nat-code [code]:  
 $\text{num-of-nat} = \text{num-of-integer} \circ \text{of-nat}$   
 ⟨proof⟩

**end**

**lemma** (in semiring-1) of-nat-code-if:  
 $\text{of-nat } n = (\text{if } n = 0 \text{ then } 0$   
   else let  
      $(m, q) = \text{Euclidean-Rings.divmod-nat } n \ 2;$   
      $m' = 2 * \text{of-nat } m$   
     in if  $q = 0$  then  $m'$  else  $m' + 1$ )  
 ⟨proof⟩

**declare** of-nat-code-if [code]

**definition** int-of-nat :: nat  $\Rightarrow$  int **where**  
 [code-abbrev]:  $\text{int-of-nat} = \text{of-nat}$

**lemma** [code]:  
 $\text{int-of-nat } n = \text{int-of-integer } (\text{of-nat } n)$   
 ⟨proof⟩

**lemma** [code abstract]:

*integer-of-nat* (nat *k*) = max 0 (*integer-of-int* *k*)  
**including** *integer.lifting* ⟨proof⟩

**definition** *char-of-nat* :: nat ⇒ char  
**where** [*code-abbrev*]: *char-of-nat* = *char-of*

**definition** *nat-of-char* :: char ⇒ nat  
**where** [*code-abbrev*]: *nat-of-char* = *of-char*

**lemma** [*code*]:  
*char-of-nat* = *char-of-integer* ∘ *integer-of-nat*  
**including** *integer.lifting* ⟨proof⟩

**lemma** [*code abstract*]:  
*integer-of-nat* (nat-of-char *c*) = *integer-of-char* *c*  
 ⟨proof⟩

**lemma** *term-of-nat-code* [*code*]:  
 — Use *nat-of-integer* in term reconstruction instead of *Code-Target-Nat.Nat* such that reconstructed terms can be fed back to the code generator  
*term-of-class.term-of* *n* =  
*Code-Evaluation.App*  
 (*Code-Evaluation.Const* (STR "Code-Numeral.nat-of-integer")  
 (*typerep.Typerep* (STR "fun")  
 [*typerep.Typerep* (STR "Code-Numeral.integer") []],  
*typerep.Typerep* (STR "Nat.nat") []]))  
 (*term-of-class.term-of* (*integer-of-nat* *n*))  
 ⟨proof⟩

**lemma** *nat-of-integer-code-post* [*code-post*]:  
*nat-of-integer* 0 = 0  
*nat-of-integer* 1 = 1  
*nat-of-integer* (numeral *k*) = numeral *k*  
**including** *integer.lifting* ⟨proof⟩

**code-identifier**

**code-module** *Code-Target-Nat* ↪  
 (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

## 123 Implementation of bit-shifts on target-language integers by built-in operations

**theory** *Code-Target-Bit-Shifts*  
**imports** *Main*  
**begin**

**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]:  
 $\langle \text{push-bit } i \ k = k * 2 ^ \text{nat-of-integer } |i| \rangle$   
 $\langle \text{proof} \rangle$

**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]:  
 $\langle \text{drop-bit } i \ k = k \text{ div } 2 ^ \text{nat-of-integer } |i| \rangle$   
 $\langle \text{proof} \rangle$

**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**

**code-printing code-module** *Bit-Shifts*  $\rightarrow$   
 (SML)  $\langle$

*structure Bit-Shifts : sig*  
*type int = IntInf.int*  
*val push : int  $\rightarrow$  int  $\rightarrow$  int*  
*val drop : int  $\rightarrow$  int  $\rightarrow$  int*  
*val word-max-index : Word.word (\*only for validation\*)*  
*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, Int. - (Word.wordSize, 3)) - fromInt 1; (\*experimentally determined\*)*

*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-Target-Bit-Shifts.push-bit Code-Target-Bit-Shifts.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 >~ 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-Target-Bit-Shifts.push-bit Code-Target-Bit-Shifts.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 >~ maxIndex will pro-
duce 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-Target-Bit-Shifts.push-bit Code-Target-Bit-Shifts.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-Target-Bit-Shifts.push-bit Code-Target-Bit-Shifts.drop-bit
| constant Code-Target-Bit-Shifts.push-bit →
  (SML) Bit'-Shifts.push
  and (OCaml) Bit'-Shifts.push
  and (Haskell) Bit'-Shifts.push
  and (Haskell-Quickcheck) Bit'-Shifts.push'
  and (Scala) Bit'-Shifts.push
| constant Code-Target-Bit-Shifts.drop-bit →
  (SML) Bit'-Shifts.drop
  and (OCaml) Bit'-Shifts.drop
  and (Haskell) Bit'-Shifts.drop
  and (Haskell-Quickcheck) Bit'-Shifts.drop'
  and (Scala) Bit'-Shifts.drop

code-reserved
  (SML) Bit-Shifts
  and (Haskell) Bit-Shifts
  and (Scala) Bit-Shifts

end

```

## 124 Implementation of natural and integer numbers by target-language integers

```

theory Code-Target-Numeral
imports Code-Target-Nat Code-Target-Int Code-Target-Bit-Shifts
begin

end

```

## 125 Preprocessor setup for floats implemented by target language numerals

```

theory Code-Target-Numeral-Float
imports Float Code-Target-Numeral
begin

```

```

lemma numeral-float-computation-unfold [code-computation-unfold]:

```

```

  ⟨numeral k = Float (int-of-integer (Code-Numeral.positive k)) 0⟩
  ⟨← numeral k = Float (int-of-integer (Code-Numeral.negative k)) 0⟩
  ⟨proof⟩

```

**end**

```

theory Complex-Order
  imports Complex-Main
begin

```

```

instantiation complex :: order begin

```

```

definition ⟨x < y ⟷ Re x < Re y ∧ Im x = Im y⟩

```

```

definition ⟨x ≤ y ⟷ Re x ≤ Re y ∧ Im x = Im y⟩

```

```

instance
  ⟨proof⟩
end

```

```

lemma nonnegative-complex-is-real: ⟨(x::complex) ≥ 0 ⟹ x ∈ ℝ⟩
  ⟨proof⟩

```

```

lemma complex-is-real-iff-compare0: ⟨(x::complex) ∈ ℝ ⟷ x ≤ 0 ∨ x ≥ 0⟩
  ⟨proof⟩

```

```

instance complex :: ordered-comm-ring
  ⟨proof⟩

```

```

instance complex :: ordered-real-vector
  ⟨proof⟩

```

```

instance complex :: ordered-cancel-comm-semiring
  ⟨proof⟩

```

**end**

## 126 Abstract type of association lists with unique keys

```

theory DAList
imports AList
begin

```

This was based on some existing fragments in the AFP-Collection framework.



### 126.1 Preliminaries

**lemma** *distinct-map-fst-filter*:

$distinct (map\ fst\ xs) \implies distinct (map\ fst (List.filter\ P\ xs))$   
 $\langle proof \rangle$

### 126.2 Type (*'key*, *'value*) *alist*

**typedef** (*'key*, *'value*) *alist* = {*xs* :: (*'key* × *'value*) *list*. (*distinct* ∘ *map fst*) *xs*}

**morphisms** *impl-of Alist*

$\langle proof \rangle$

**setup-lifting** *type-definition-alist*

**lemma** *alist-ext*:  $impl-of\ xs = impl-of\ ys \implies xs = ys$

$\langle proof \rangle$

**lemma** *alist-eq-iff*:  $xs = ys \iff impl-of\ xs = impl-of\ ys$

$\langle proof \rangle$

**lemma** *impl-of-distinct* [*simp*, *intro*]:  $distinct (map\ fst (impl-of\ xs))$

$\langle proof \rangle$

**lemma** *Alist-impl-of* [*code abstype*]:  $Alist (impl-of\ xs) = xs$

$\langle proof \rangle$

### 126.3 Primitive operations

**lift-definition** *lookup* :: (*'key*, *'value*) *alist*  $\Rightarrow$  *'key*  $\Rightarrow$  *'value option* **is** *map-of*

$\langle proof \rangle$

**lift-definition** *empty* :: (*'key*, *'value*) *alist* **is** []

$\langle proof \rangle$

**lift-definition** *update* :: *'key*  $\Rightarrow$  *'value*  $\Rightarrow$  (*'key*, *'value*) *alist*  $\Rightarrow$  (*'key*, *'value*) *alist*  
**is** *AList.update*

$\langle proof \rangle$

**lift-definition** *delete* :: *'key*  $\Rightarrow$  (*'key*, *'value*) *alist*  $\Rightarrow$  (*'key*, *'value*) *alist*

**is** *AList.delete*

$\langle proof \rangle$

**lift-definition** *map-entry* ::

*'key*  $\Rightarrow$  (*'value*  $\Rightarrow$  *'value*)  $\Rightarrow$  (*'key*, *'value*) *alist*  $\Rightarrow$  (*'key*, *'value*) *alist*

**is** *AList.map-entry*

$\langle proof \rangle$

**lift-definition** *filter* :: (*'key* × *'value*  $\Rightarrow$  *bool*)  $\Rightarrow$  (*'key*, *'value*) *alist*  $\Rightarrow$  (*'key*, *'value*) *alist*

**is** *List.filter*  
 ⟨*proof*⟩

**lift-definition** *map-default* ::

$'key \Rightarrow 'value \Rightarrow ('value \Rightarrow 'value) \Rightarrow ('key, 'value) \text{ alist} \Rightarrow ('key, 'value) \text{ alist}$   
**is** *AList.map-default*  
 ⟨*proof*⟩

## 126.4 Abstract operation properties

**lemma** *lookup-empty* [*simp*]: *lookup empty k = None*  
 ⟨*proof*⟩

**lemma** *lookup-update*:

$\text{lookup} (\text{update } k1 \ v \ xs) \ k2 = (\text{if } k1 = k2 \ \text{then } \text{Some } v \ \text{else } \text{lookup } xs \ k2)$   
 ⟨*proof*⟩

**lemma** *lookup-update-eq* [*simp*]:

$k1 = k2 \implies \text{lookup} (\text{update } k1 \ v \ xs) \ k2 = \text{Some } v$   
 ⟨*proof*⟩

**lemma** *lookup-update-neq* [*simp*]:

$k1 \neq k2 \implies \text{lookup} (\text{update } k1 \ v \ xs) \ k2 = \text{lookup } xs \ k2$   
 ⟨*proof*⟩

**lemma** *update-update-eq* [*simp*]:

$k1 = k2 \implies \text{update } k2 \ v2 (\text{update } k1 \ v1 \ xs) = \text{update } k2 \ v2 \ xs$   
 ⟨*proof*⟩

**lemma** *lookup-delete* [*simp*]: *lookup (delete k al) = (lookup al)(k := None)*  
 ⟨*proof*⟩

## 126.5 Further operations

### 126.5.1 Equality

**instantiation** *alist* :: (*equal*, *equal*) *equal*  
**begin**

**definition** *HOL.equal* (*xs* :: ('a, 'b) *alist*) *ys* == *impl-of xs = impl-of ys*

**instance**  
 ⟨*proof*⟩

**end**

### 126.5.2 Size

**instantiation** *alist* :: (*type*, *type*) *size*  
**begin**

**definition**  $size (al :: ('a, 'b) alist) = length (impl-of al)$

**instance**  $\langle proof \rangle$

**end**

## 126.6 Quickcheck generators

**context**

**includes** *state-combinator-syntax* **and** *term-syntax*

**begin**

**definition**

$valterm-empty :: ('key :: typerep, 'value :: typerep) alist \times (unit \Rightarrow Code-Evaluation.term)$   
**where**  $valterm-empty = Code-Evaluation.valtermify empty$

**definition**

$valterm-update :: 'key :: typerep \times (unit \Rightarrow Code-Evaluation.term) \Rightarrow$   
 $'value :: typerep \times (unit \Rightarrow Code-Evaluation.term) \Rightarrow$   
 $('key, 'value) alist \times (unit \Rightarrow Code-Evaluation.term) \Rightarrow$   
 $('key, 'value) alist \times (unit \Rightarrow Code-Evaluation.term)$  **where**  
 $[code-unfold]: valterm-update k v a = Code-Evaluation.valtermify update \{\cdot\} k \{\cdot\}$   
 $v \{\cdot\} a$

**fun** *random-aux-alist*

**where**

$random-aux-alist i j =$   
 $(if i = 0 then Pair valterm-empty$   
 $else Quickcheck-Random.collapse$   
 $(Random.select-weight$   
 $[(i, Quickcheck-Random.random j \circ \rightarrow (\lambda k. Quickcheck-Random.random j$   
 $\circ \rightarrow$   
 $(\lambda v. random-aux-alist (i - 1) j \circ \rightarrow (\lambda a. Pair (valterm-update k v a))))],$   
 $(1, Pair valterm-empty)]))$

**end**

**instantiation**  $alist :: (random, random) random$

**begin**

**definition** *random-alist*

**where**

$random-alist i = random-aux-alist i i$

**instance**  $\langle proof \rangle$

**end**

**instantiation** *alist* :: (*exhaustive, exhaustive*) *exhaustive*  
**begin**

**fun** *exhaustive-alist* ::  
 ((*'a, 'b*) *alist* ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*  
**where**  
*exhaustive-alist* *f* *i* =  
 (if *i* = 0 then *None*  
 else  
 case *f* *empty* of  
   *Some* *ts* ⇒ *Some* *ts*  
 | *None* ⇒  
   *exhaustive-alist*  
   ( $\lambda a.$  *Quickcheck-Exhaustive.exhaustive*  
   ( $\lambda k.$  *Quickcheck-Exhaustive.exhaustive* ( $\lambda v.$  *f* (*update* *k* *v* *a*)) (*i* - 1))  
 (*i* - 1))  
 (*i* - 1))

**instance** ⟨*proof*⟩

**end**

**instantiation** *alist* :: (*full-exhaustive, full-exhaustive*) *full-exhaustive*  
**begin**

**fun** *full-exhaustive-alist* ::  
 ((*'a, 'b*) *alist* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒  
 (*bool* × *term list*) *option*  
**where**  
*full-exhaustive-alist* *f* *i* =  
 (if *i* = 0 then *None*  
 else  
 case *f* *valterm-empty* of  
   *Some* *ts* ⇒ *Some* *ts*  
 | *None* ⇒  
   *full-exhaustive-alist*  
   ( $\lambda a.$   
   *Quickcheck-Exhaustive.full-exhaustive*  
   ( $\lambda k.$  *Quickcheck-Exhaustive.full-exhaustive* ( $\lambda v.$  *f* (*valterm-update* *k* *v*  
*a*)) (*i* - 1))  
 (*i* - 1))  
 (*i* - 1))

**instance** ⟨*proof*⟩

**end**

## 127 alist is a BNF

**lift-bnf** (*dead* 'k, *set*: 'v) *alist* [*wits*: [] :: ('k × 'v) list] **for** *map*: map *rel*: rel  
 ⟨*proof*⟩

**hide-const** *valterm-empty valterm-update random-aux-alist*

**hide-fact** (**open**) *lookup-def empty-def update-def delete-def map-entry-def filter-def  
 map-default-def*

**hide-const** (**open**) *impl-of lookup empty update delete map-entry filter map-default  
 map set rel*

**end**

## 128 Multisets partially implemented by association lists

**theory** *DAList-Multiset*

**imports** *Multiset DAList*

**begin**

Delete preexisting code equations

**declare** [[*code drop*: {#} *Multiset.is-empty add-mset  
 plus* :: 'a multiset ⇒ - *minus* :: 'a multiset ⇒ -  
*inter-mset union-mset image-mset filter-mset count  
 size* :: - multiset ⇒ nat *sum-mset prod-mset  
 set-mset sorted-list-of-multiset subset-mset subseteq-mset  
 equal-multiset-inst.equal-multiset*]]

Raw operations on lists

**definition** *join-raw* ::

(*'key* ⇒ *'val* × *'val* ⇒ *'val*) ⇒

(*'key* × *'val*) list ⇒ (*'key* × *'val*) list ⇒ (*'key* × *'val*) list

**where** *join-raw* *f* *xs* *ys* = *foldr* ( $\lambda(k, v). \text{map-default } k \ v \ (\lambda v'. f \ k \ (v', v))$ ) *ys* *xs*

**lemma** *join-raw-Nil* [*simp*]: *join-raw* *f* *xs* [] = *xs*

⟨*proof*⟩

**lemma** *join-raw-Cons* [*simp*]:

*join-raw* *f* *xs* ((*k*, *v*) # *ys*) = *map-default* *k* *v* ( $\lambda v'. f \ k \ (v', v)$ ) (*join-raw* *f* *xs* *ys*)

⟨*proof*⟩

**lemma** *map-of-join-raw*:

**assumes** *distinct* (*map fst* *ys*)

**shows** *map-of* (*join-raw* *f* *xs* *ys*) *x* =

(*case map-of* *xs* *x* of

*None* ⇒ *map-of* *ys* *x*

| *Some* *v* ⇒ (*case map-of* *ys* *x* of *None* ⇒ *Some* *v* | *Some* *v'* ⇒ *Some* (*f* *x* (*v*, *v'*))))

⟨proof⟩

**lemma** *distinct-join-raw*:

**assumes** *distinct* (*map fst xs*)

**shows** *distinct* (*map fst (join-raw f xs ys)*)

⟨proof⟩

**definition** *subtract-entries-raw* *xs ys* = *foldr* ( $\lambda(k, v). AList.map\text{-}entry\ k\ (\lambda v'. v' - v)$ ) *ys xs*

**lemma** *map-of-subtract-entries-raw*:

**assumes** *distinct* (*map fst ys*)

**shows** *map-of* (*subtract-entries-raw xs ys*) *x* =

(*case map-of xs x of*

*None*  $\Rightarrow$  *None*

| *Some v*  $\Rightarrow$  (*case map-of ys x of None*  $\Rightarrow$  *Some v* | *Some v'*  $\Rightarrow$  *Some (v - v')*)

⟨proof⟩

**lemma** *distinct-subtract-entries-raw*:

**assumes** *distinct* (*map fst xs*)

**shows** *distinct* (*map fst (subtract-entries-raw xs ys)*)

⟨proof⟩

Operations on alists with distinct keys

**lift-definition** *join* :: (*'a*  $\Rightarrow$  *'b*  $\times$  *'b*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'a*, *'b*) *alist*  $\Rightarrow$  (*'a*, *'b*) *alist*  $\Rightarrow$  (*'a*, *'b*) *alist*

**is** *join-raw*

⟨proof⟩

**lift-definition** *subtract-entries* :: (*'a*, (*'b* :: *minus*)) *alist*  $\Rightarrow$  (*'a*, *'b*) *alist*  $\Rightarrow$  (*'a*, *'b*) *alist*

**is** *subtract-entries-raw*

⟨proof⟩

Implementing multisets by means of association lists

**definition** *count-of* :: (*'a*  $\times$  *nat*) *list*  $\Rightarrow$  *'a*  $\Rightarrow$  *nat*

**where** *count-of xs x* = (*case map-of xs x of None*  $\Rightarrow$  0 | *Some n*  $\Rightarrow$  *n*)

**lemma** *count-of-multiset*: *finite* {*x*. 0 < *count-of xs x*}

⟨proof⟩

**lemma** *count-simps* [*simp*]:

*count-of []* = ( $\lambda\_. 0$ )

*count-of* ((*x*, *n*) # *xs*) = ( $\lambda y. \text{if } x = y \text{ then } n \text{ else } \text{count-of } xs\ y$ )

⟨proof⟩

**lemma** *count-of-empty*: *x*  $\notin$  *fst* ' *set xs*  $\Longrightarrow$  *count-of xs x* = 0

⟨proof⟩

**lemma** *count-of-filter*:  $\text{count-of } (\text{List.filter } (P \circ \text{fst}) \text{ } xs) \text{ } x = (\text{if } P \text{ } x \text{ then count-of } xs \text{ } x \text{ else } 0)$

*<proof>*

**lemma** *count-of-map-default* [*simp*]:

$\text{count-of } (\text{map-default } x \text{ } b \text{ } (\lambda x. x + b) \text{ } xs) \text{ } y =$   
*(if*  $x = y$  *then*  $\text{count-of } xs \text{ } x + b$  *else*  $\text{count-of } xs \text{ } y$ *)*

*<proof>*

**lemma** *count-of-join-raw*:

$\text{distinct } (\text{map fst } ys) \implies$   
 $\text{count-of } xs \text{ } x + \text{count-of } ys \text{ } x = \text{count-of } (\text{join-raw } (\lambda x \text{ } (x, y). x + y) \text{ } xs \text{ } ys) \text{ } x$

*<proof>*

**lemma** *count-of-subtract-entries-raw*:

$\text{distinct } (\text{map fst } ys) \implies$   
 $\text{count-of } xs \text{ } x - \text{count-of } ys \text{ } x = \text{count-of } (\text{subtract-entries-raw } xs \text{ } ys) \text{ } x$

*<proof>*

Code equations for multiset operations

**definition** *Bag* ::  $('a, \text{nat}) \text{ alist} \Rightarrow 'a \text{ multiset}$

**where**  $\text{Bag } xs = \text{Abs-multiset } (\text{count-of } (\text{DAList.impl-of } xs))$

**code-datatype** *Bag*

**lemma** *count-Bag* [*simp, code*]:  $\text{count } (\text{Bag } xs) = \text{count-of } (\text{DAList.impl-of } xs)$

*<proof>*

**lemma** *Mempty-Bag* [*code*]:  $\{\#\} = \text{Bag } (\text{DAList.empty})$

*<proof>*

**lift-definition** *is-empty-Bag-impl* ::  $('a, \text{nat}) \text{ alist} \Rightarrow \text{bool}$  **is**

$\lambda xs. \text{list-all } (\lambda x. \text{snd } x = 0) \text{ } xs$  *<proof>*

**lemma** *is-empty-Bag* [*code*]:  $\text{Multiset.is-empty } (\text{Bag } xs) \longleftrightarrow \text{is-empty-Bag-impl } xs$

*<proof>*

**lemma** *union-Bag* [*code*]:  $\text{Bag } xs + \text{Bag } ys = \text{Bag } (\text{join } (\lambda x \text{ } (n1, n2). n1 + n2) \text{ } xs \text{ } ys)$

*<proof>*

**lemma** *add-mset-Bag* [*code*]:  $\text{add-mset } x \text{ } (\text{Bag } xs) =$

$\text{Bag } (\text{join } (\lambda x \text{ } (n1, n2). n1 + n2) \text{ } (\text{DAList.update } x \text{ } 1 \text{ } \text{DAList.empty}) \text{ } xs)$

*<proof>*

**lemma** *minus-Bag* [*code*]:  $\text{Bag } xs - \text{Bag } ys = \text{Bag } (\text{subtract-entries } xs \text{ } ys)$

*<proof>*

**lemma** *filter-Bag* [*code*]:  $\text{filter-mset } P \text{ } (\text{Bag } xs) = \text{Bag } (\text{DAList.filter } (P \circ \text{fst}) \text{ } xs)$

*<proof>*

**lemma** *mset-eq* [code]: *HOL.equal* (*m1* :: 'a::equal multiset) *m2*  $\longleftrightarrow$   $m1 \subseteq\# m2 \wedge m2 \subseteq\# m1$   
*<proof>*

By default the code for  $<$  is  $(xs < ys) = (xs \leq ys \wedge xs \neq ys)$ . With equality implemented by  $\leq$ , this leads to three calls of  $\leq$ . Here is a more efficient version:

**lemma** *mset-less*[code]:  $xs \subset\# (ys :: 'a \text{ multiset}) \longleftrightarrow xs \subseteq\# ys \wedge \neg ys \subseteq\# xs$   
*<proof>*

**lemma** *mset-less-eq-Bag0*:

*Bag*  $xs \subseteq\# A \longleftrightarrow (\forall (x, n) \in \text{set } (DAList.impl\text{-of } xs). \text{count-of } (DAList.impl\text{-of } xs) x \leq \text{count } A x)$   
 (is ?lhs  $\longleftrightarrow$  ?rhs)  
*<proof>*

**lemma** *mset-less-eq-Bag* [code]:

*Bag*  $xs \subseteq\# (A :: 'a \text{ multiset}) \longleftrightarrow (\forall (x, n) \in \text{set } (DAList.impl\text{-of } xs). n \leq \text{count } A x)$   
*<proof>*

**declare** *inter-mset-def* [code]

**declare** *union-mset-def* [code]

**declare** *mset.simps* [code]

**fun** *fold-impl* :: ('a  $\Rightarrow$  nat  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\times$  nat) list  $\Rightarrow$  'b

**where**

*fold-impl* fn e ((a,n) # ms) = (*fold-impl* fn ((fn a n) e) ms)  
 | *fold-impl* fn e [] = e

**context**

**begin**

**qualified definition** *fold* :: ('a  $\Rightarrow$  nat  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'b  $\Rightarrow$  ('a, nat) alist  $\Rightarrow$  'b  
**where** *fold* f e al = *fold-impl* f e (DAList.impl-of al)

**end**

**context** *comp-fun-commute*

**begin**

**lemma** *DAList-Multiset-fold*:

**assumes** fn:  $\bigwedge a n x. \text{fn } a n x = (f a \widetilde{\sim} n) x$

**shows** *fold-mset* f e (Bag al) = *DAList-Multiset.fold* fn e al

*<proof>*



**end**

**context**  
**begin**

**private lift-definition** *single-alist-entry* :: 'a ⇒ 'b ⇒ ('a, 'b) alist **is** λa b. [(a, b)]  
 ⟨proof⟩

**lemma** *image-mset-Bag* [code]:  
*image-mset* f (Bag ms) =  
 DAList-Multiset.fold (λa n m. Bag (single-alist-entry (f a) n) + m) {#} ms  
 ⟨proof⟩

**end**

— we cannot use λa n. (+) (a \* n) for folding, since (\*) is not defined in *comm-monoid-add*

**lemma** *sum-mset-Bag*[code]: *sum-mset* (Bag ms) = DAList-Multiset.fold (λa n. ((+) a)  $\widetilde{\sim}$  n) 0 ms  
 ⟨proof⟩

**lemma** *prod-mset-Bag*[code]: *prod-mset* (Bag ms) = DAList-Multiset.fold (λa n. ((\* ) a)  $\widetilde{\sim}$  n) 1 ms  
 ⟨proof⟩

**lemma** *size-fold*: *size* A = *fold-mset* (λ-. Suc) 0 A (**is** - = *fold-mset* ?f - -)  
 ⟨proof⟩

**lemma** *size-Bag*[code]: *size* (Bag ms) = DAList-Multiset.fold (λa n. (+) n) 0 ms  
 ⟨proof⟩

**lemma** *set-mset-fold*: *set-mset* A = *fold-mset* insert {} A (**is** - = *fold-mset* ?f - -)  
 ⟨proof⟩

**lemma** *set-mset-Bag*[code]:  
*set-mset* (Bag ms) = DAList-Multiset.fold (λa n. (if n = 0 then (λm. m) else insert a)) {} ms  
 ⟨proof⟩

**instantiation** *multiset* :: (exhaustive) exhaustive  
**begin**

**definition** *exhaustive-multiset* ::  
 ('a multiset ⇒ (bool × term list) option) ⇒ natural ⇒ (bool × term list) option  
**where** *exhaustive-multiset* f i = Quickcheck-Exhaustive.exhaustive (λxs. f (Bag xs)) i

```
instance <proof>
```

```
end
```

```
end
```

## 129 Implementation of Red-Black Trees

```
theory RBT-Impl
```

```
imports Main
```

```
begin
```

For applications, you should use theory *RBT* which defines an abstract type of red-black tree obeying the invariant.

### 129.1 Datatype of RB trees

```
datatype color = R | B
```

```
datatype ('a, 'b) rbt = Empty | Branch color ('a, 'b) rbt 'a 'b ('a, 'b) rbt
```

```
lemma rbt-cases:
```

```
  obtains (Empty) t = Empty
```

```
  | (Red) l k v r where t = Branch R l k v r
```

```
  | (Black) l k v r where t = Branch B l k v r
```

```
<proof>
```

### 129.2 Tree properties

#### 129.2.1 Content of a tree

```
primrec entries :: ('a, 'b) rbt ⇒ ('a × 'b) list
```

```
where
```

```
  entries Empty = []
```

```
| entries (Branch c l k v r) = entries l @ (k,v) # entries r
```

```
abbreviation (input) entry-in-tree :: 'a ⇒ 'b ⇒ ('a, 'b) rbt ⇒ bool
```

```
where
```

```
  entry-in-tree k v t ≡ (k, v) ∈ set (entries t)
```

```
definition keys :: ('a, 'b) rbt ⇒ 'a list where
```

```
  keys t = map fst (entries t)
```

```
lemma keys-simps [simp, code]:
```

```
  keys Empty = []
```

```
  keys (Branch c l k v r) = keys l @ k # keys r
```

```
<proof>
```

```
lemma entry-in-tree-keys:
```

```
  assumes (k, v) ∈ set (entries t)
```

**shows**  $k \in \text{set } (\text{keys } t)$   
 ⟨proof⟩

**lemma** *keys-entries*:  
 $k \in \text{set } (\text{keys } t) \longleftrightarrow (\exists v. (k, v) \in \text{set } (\text{entries } t))$   
 ⟨proof⟩

**lemma** *non-empty-rbt-keys*:  
 $t \neq \text{rbt.Empty} \implies \text{keys } t \neq []$   
 ⟨proof⟩

## 129.2.2 Search tree properties

**context** *ord* **begin**

**definition** *rbt-less* ::  $'a \Rightarrow ('a, 'b) \text{rbt} \Rightarrow \text{bool}$   
**where**  
*rbt-less-prop*:  $\text{rbt-less } k \ t \longleftrightarrow (\forall x \in \text{set } (\text{keys } t). x < k)$

**abbreviation** *rbt-less-symbol* (**infix**  $\langle | \ll \rangle$  50)  
**where**  $t | \ll x \equiv \text{rbt-less } x \ t$

**definition** *rbt-greater* ::  $'a \Rightarrow ('a, 'b) \text{rbt} \Rightarrow \text{bool}$  (**infix**  $\langle \ll | \rangle$  50)  
**where**  
*rbt-greater-prop*:  $\text{rbt-greater } k \ t = (\forall x \in \text{set } (\text{keys } t). k < x)$

**lemma** *rbt-less-simps* [*simp*]:  
 $\text{Empty} | \ll k = \text{True}$   
 $\text{Branch } c \ lt \ kt \ v \ rt | \ll k \longleftrightarrow kt < k \wedge lt | \ll k \wedge rt | \ll k$   
 ⟨proof⟩

**lemma** *rbt-greater-simps* [*simp*]:  
 $k \ll | \text{Empty} = \text{True}$   
 $k \ll | (\text{Branch } c \ lt \ kt \ v \ rt) \longleftrightarrow k < kt \wedge k \ll | lt \wedge k \ll | rt$   
 ⟨proof⟩

**lemmas** *rbt-ord-props* = *rbt-less-prop* *rbt-greater-prop*

**lemmas** *rbt-greater-nit* = *rbt-greater-prop* *entry-in-tree-keys*

**lemmas** *rbt-less-nit* = *rbt-less-prop* *entry-in-tree-keys*

**lemma** (**in order**)  
**shows** *rbt-less-eq-trans*:  $l | \ll u \implies u \leq v \implies l | \ll v$   
**and** *rbt-less-trans*:  $t | \ll x \implies x < y \implies t | \ll y$   
**and** *rbt-greater-eq-trans*:  $u \leq v \implies v \ll | r \implies u \ll | r$   
**and** *rbt-greater-trans*:  $x < y \implies y \ll | t \implies x \ll | t$   
 ⟨proof⟩

**primrec** *rbt-sorted* ::  $('a, 'b) \text{rbt} \Rightarrow \text{bool}$

**where**

$rbt\text{-sorted } Empty = True$   
 $| rbt\text{-sorted } (Branch\ c\ l\ k\ v\ r) = (l \ll k \wedge k \ll r \wedge rbt\text{-sorted } l \wedge rbt\text{-sorted } r)$

**end**

**context** *linorder* **begin**

**lemma** *rbt-sorted-entries*:

$rbt\text{-sorted } t \implies List.sorted\ (map\ fst\ (entries\ t))$   
 $\langle proof \rangle$

**lemma** *distinct-entries*:

$rbt\text{-sorted } t \implies distinct\ (map\ fst\ (entries\ t))$   
 $\langle proof \rangle$

**lemma** *distinct-keys*:

$rbt\text{-sorted } t \implies distinct\ (keys\ t)$   
 $\langle proof \rangle$

### 129.2.3 Tree lookup

**primrec** (in *ord*) *rbt-lookup* :: ('a, 'b) *rbt*  $\Rightarrow$  'a  $\rightarrow$  'b

**where**

$rbt\text{-lookup } Empty\ k = None$   
 $| rbt\text{-lookup } (Branch\ -\ l\ x\ y\ r)\ k =$   
 $(if\ k < x\ then\ rbt\text{-lookup } l\ k\ else\ if\ x < k\ then\ rbt\text{-lookup } r\ k\ else\ Some\ y)$

**lemma** *rbt-lookup-keys*:  $rbt\text{-sorted } t \implies dom\ (rbt\text{-lookup } t) = set\ (keys\ t)$

$\langle proof \rangle$

**lemma** *dom-rbt-lookup-Branch*:

$rbt\text{-sorted } (Branch\ c\ t1\ k\ v\ t2) \implies$   
 $dom\ (rbt\text{-lookup } (Branch\ c\ t1\ k\ v\ t2))$   
 $= Set.insert\ k\ (dom\ (rbt\text{-lookup } t1) \cup dom\ (rbt\text{-lookup } t2))$   
 $\langle proof \rangle$

**lemma** *finite-dom-rbt-lookup* [*simp*, *intro!*]:  $finite\ (dom\ (rbt\text{-lookup } t))$

$\langle proof \rangle$

**end**

**context** *ord* **begin**

**lemma** *rbt-lookup-rbt-less*[*simp*]:  $t \ll k \implies rbt\text{-lookup } t\ k = None$

$\langle proof \rangle$

**lemma** *rbt-lookup-rbt-greater*[*simp*]:  $k \ll t \implies rbt\text{-lookup } t\ k = None$

$\langle proof \rangle$

**lemma** *rbt-lookup-Empty*:  $\text{rbt-lookup Empty} = \text{Map.empty}$   
 ⟨proof⟩

**end**

**context** *linorder* **begin**

**lemma** *map-of-entries*:  
 $\text{rbt-sorted } t \implies \text{map-of (entries } t) = \text{rbt-lookup } t$   
 ⟨proof⟩

**lemma** *rbt-lookup-in-tree*:  $\text{rbt-sorted } t \implies \text{rbt-lookup } t \ k = \text{Some } v \iff (k, v) \in \text{set (entries } t)$   
 ⟨proof⟩

**lemma** *set-entries-inject*:  
**assumes** *rbt-sorted*:  $\text{rbt-sorted } t1 \ \text{rbt-sorted } t2$   
**shows**  $\text{set (entries } t1) = \text{set (entries } t2) \iff \text{entries } t1 = \text{entries } t2$   
 ⟨proof⟩

**lemma** *entries-eqI*:  
**assumes** *rbt-sorted*:  $\text{rbt-sorted } t1 \ \text{rbt-sorted } t2$   
**assumes** *rbt-lookup*:  $\text{rbt-lookup } t1 = \text{rbt-lookup } t2$   
**shows**  $\text{entries } t1 = \text{entries } t2$   
 ⟨proof⟩

**lemma** *entries-rbt-lookup*:  
**assumes** *rbt-sorted*  $t1 \ \text{rbt-sorted } t2$   
**shows**  $\text{entries } t1 = \text{entries } t2 \iff \text{rbt-lookup } t1 = \text{rbt-lookup } t2$   
 ⟨proof⟩

**lemma** *rbt-lookup-from-in-tree*:  
**assumes** *rbt-sorted*  $t1 \ \text{rbt-sorted } t2$   
**and**  $\bigwedge v. (k, v) \in \text{set (entries } t1) \iff (k, v) \in \text{set (entries } t2)$   
**shows**  $\text{rbt-lookup } t1 \ k = \text{rbt-lookup } t2 \ k$   
 ⟨proof⟩

**end**

#### 129.2.4 Red-black properties

**primrec** *color-of* ::  $('a, 'b) \text{ rbt} \Rightarrow \text{color}$   
**where**

$\text{color-of Empty} = B$   
 $|\ \text{color-of (Branch } c \ \_ \ \_ \ \_) = c$

**primrec** *bheight* ::  $('a, 'b) \text{ rbt} \Rightarrow \text{nat}$   
**where**

$bheight\ Empty = 0$   
 $| bheight\ (Branch\ c\ lt\ k\ v\ rt) = (if\ c = B\ then\ Suc\ (bheight\ lt)\ else\ bheight\ lt)$

**primrec**  $inv1 :: ('a, 'b)\ rbt \Rightarrow bool$

**where**

$inv1\ Empty = True$   
 $| inv1\ (Branch\ c\ lt\ k\ v\ rt) \longleftrightarrow inv1\ lt \wedge inv1\ rt \wedge (c = B \vee color-of\ lt = B \wedge color-of\ rt = B)$

**primrec**  $inv1l :: ('a, 'b)\ rbt \Rightarrow bool$  — Weaker version

**where**

$inv1l\ Empty = True$   
 $| inv1l\ (Branch\ c\ l\ k\ v\ r) = (inv1\ l \wedge inv1\ r)$   
**lemma**  $[simp]: inv1\ t \Longrightarrow inv1l\ t$   $\langle proof \rangle$

**primrec**  $inv2 :: ('a, 'b)\ rbt \Rightarrow bool$

**where**

$inv2\ Empty = True$   
 $| inv2\ (Branch\ c\ lt\ k\ v\ rt) = (inv2\ lt \wedge inv2\ rt \wedge bheight\ lt = bheight\ rt)$

**context**  $ord\ begin$

**definition**  $is-rbt :: ('a, 'b)\ rbt \Rightarrow bool$  **where**

$is-rbt\ t \longleftrightarrow inv1\ t \wedge inv2\ t \wedge color-of\ t = B \wedge rbt-sorted\ t$

**lemma**  $is-rbt-rbt-sorted$   $[simp]:$

$is-rbt\ t \Longrightarrow rbt-sorted\ t$   $\langle proof \rangle$

**theorem**  $Empty-is-rbt$   $[simp]:$

$is-rbt\ Empty$   $\langle proof \rangle$

**end**

### 129.3 Insertion

The function definitions are based on the book by Okasaki.

**fun**

$balance :: ('a, 'b)\ rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

**where**

$balance\ (Branch\ R\ a\ w\ x\ b)\ s\ t\ (Branch\ R\ c\ y\ z\ d) = Branch\ R\ (Branch\ B\ a\ w\ x\ b)\ s\ t\ (Branch\ B\ c\ y\ z\ d) |$   
 $balance\ (Branch\ R\ (Branch\ R\ a\ w\ x\ b)\ s\ t\ c)\ y\ z\ d = Branch\ R\ (Branch\ B\ a\ w\ x\ b)\ s\ t\ (Branch\ B\ c\ y\ z\ d) |$   
 $balance\ (Branch\ R\ a\ w\ x\ (Branch\ R\ b\ s\ t\ c))\ y\ z\ d = Branch\ R\ (Branch\ B\ a\ w\ x\ b)\ s\ t\ (Branch\ B\ c\ y\ z\ d) |$   
 $balance\ a\ w\ x\ (Branch\ R\ b\ s\ t\ (Branch\ R\ c\ y\ z\ d)) = Branch\ R\ (Branch\ B\ a\ w\ x\ b)\ s\ t\ (Branch\ B\ c\ y\ z\ d) |$   
 $balance\ a\ w\ x\ (Branch\ R\ (Branch\ R\ b\ s\ t\ c)\ y\ z\ d) = Branch\ R\ (Branch\ B\ a\ w\ x\ b)\ s\ t\ (Branch\ B\ c\ y\ z\ d) |$

$balance\ a\ s\ t\ b = Branch\ B\ a\ s\ t\ b$

**lemma** *balance-inv1*:  $\llbracket inv1\ l; inv1\ r \rrbracket \implies inv1\ (balance\ l\ k\ v\ r)$   
 ⟨proof⟩

**lemma** *balance-bheight*:  $bheight\ l = bheight\ r \implies bheight\ (balance\ l\ k\ v\ r) = Suc\ (bheight\ l)$   
 ⟨proof⟩

**lemma** *balance-inv2*:  
**assumes**  $inv2\ l\ inv2\ r\ bheight\ l = bheight\ r$   
**shows**  $inv2\ (balance\ l\ k\ v\ r)$   
 ⟨proof⟩

**context** *ord* **begin**

**lemma** *balance-rbt-greater[simp]*:  $(v\ \ll\ | balance\ a\ k\ x\ b) = (v\ \ll\ | a\ \wedge\ v\ \ll\ | b\ \wedge\ v\ <\ k)$   
 ⟨proof⟩

**lemma** *balance-rbt-less[simp]*:  $(balance\ a\ k\ x\ b\ |\ll\ v) = (a\ |\ll\ v\ \wedge\ b\ |\ll\ v\ \wedge\ k\ <\ v)$   
 ⟨proof⟩

**end**

**lemma** (**in** *linorder*) *balance-rbt-sorted*:  
**fixes**  $k :: 'a$   
**assumes**  $rbt\ sorted\ l\ rbt\ sorted\ r\ l\ |\ll\ k\ k\ \ll\ | r$   
**shows**  $rbt\ sorted\ (balance\ l\ k\ v\ r)$   
 ⟨proof⟩

**lemma** *entries-balance [simp]*:  
 $entries\ (balance\ l\ k\ v\ r) = entries\ l\ @\ (k,\ v)\ \#\ entries\ r$   
 ⟨proof⟩

**lemma** *keys-balance [simp]*:  
 $keys\ (balance\ l\ k\ v\ r) = keys\ l\ @\ k\ \#\ keys\ r$   
 ⟨proof⟩

**lemma** *balance-in-tree*:  
 $entry\ in\ tree\ k\ x\ (balance\ l\ v\ y\ r) \iff entry\ in\ tree\ k\ x\ l\ \vee\ k = v\ \wedge\ x = y\ \vee\ entry\ in\ tree\ k\ x\ r$   
 ⟨proof⟩

**lemma** (**in** *linorder*) *rbt-lookup-balance[simp]*:  
**fixes**  $k :: 'a$   
**assumes**  $rbt\ sorted\ l\ rbt\ sorted\ r\ l\ |\ll\ k\ k\ \ll\ | r$   
**shows**  $rbt\ lookup\ (balance\ l\ k\ v\ r)\ x = rbt\ lookup\ (Branch\ B\ l\ k\ v\ r)\ x$   
 ⟨proof⟩

**primrec** *paint* :: *color*  $\Rightarrow$  ('a,'b) *rbt*  $\Rightarrow$  ('a,'b) *rbt*

**where**

*paint* *c* *Empty* = *Empty*

| *paint* *c* (*Branch* - *l k v r*) = *Branch* *c l k v r*

**lemma** *paint-inv1l[simp]*: *inv1l* *t*  $\Longrightarrow$  *inv1l* (*paint* *c* *t*)  $\langle$ *proof* $\rangle$

**lemma** *paint-inv1[simp]*: *inv1* *t*  $\Longrightarrow$  *inv1* (*paint* *B* *t*)  $\langle$ *proof* $\rangle$

**lemma** *paint-inv2[simp]*: *inv2* *t*  $\Longrightarrow$  *inv2* (*paint* *c* *t*)  $\langle$ *proof* $\rangle$

**lemma** *paint-color-of[simp]*: *color-of* (*paint* *B* *t*) = *B*  $\langle$ *proof* $\rangle$

**lemma** *paint-in-tree[simp]*: *entry-in-tree* *k* *x* (*paint* *c* *t*) = *entry-in-tree* *k* *x* *t*  $\langle$ *proof* $\rangle$

**context** *ord* **begin**

**lemma** *paint-rbt-sorted[simp]*: *rbt-sorted* *t*  $\Longrightarrow$  *rbt-sorted* (*paint* *c* *t*)  $\langle$ *proof* $\rangle$

**lemma** *paint-rbt-lookup[simp]*: *rbt-lookup* (*paint* *c* *t*) = *rbt-lookup* *t*  $\langle$ *proof* $\rangle$

**lemma** *paint-rbt-greater[simp]*: (*v*  $\ll$  | *paint* *c* *t*) = (*v*  $\ll$  | *t*)  $\langle$ *proof* $\rangle$

**lemma** *paint-rbt-less[simp]*: (*paint* *c* *t* |  $\ll$  *v*) = (*t* |  $\ll$  *v*)  $\langle$ *proof* $\rangle$

**fun**

*rbt-ins* :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  ('a,'b) *rbt*  $\Rightarrow$  ('a,'b) *rbt*

**where**

*rbt-ins* *f* *k* *v* *Empty* = *Branch* *R* *Empty* *k* *v* *Empty* |

*rbt-ins* *f* *k* *v* (*Branch* *B* *l* *x* *y* *r*) = (if *k* < *x* then *balance* (*rbt-ins* *f* *k* *v* *l*) *x* *y* *r*  
 else if *k* > *x* then *balance* *l* *x* *y* (*rbt-ins* *f* *k* *v* *r*)  
 else *Branch* *B* *l* *x* (*f* *k* *y* *v*) *r*) |

*rbt-ins* *f* *k* *v* (*Branch* *R* *l* *x* *y* *r*) = (if *k* < *x* then *Branch* *R* (*rbt-ins* *f* *k* *v* *l*) *x* *y* *r*  
 else if *k* > *x* then *Branch* *R* *l* *x* *y* (*rbt-ins* *f* *k* *v* *r*)  
 else *Branch* *R* *l* *x* (*f* *k* *y* *v*) *r*)

**lemma** *ins-inv1-inv2*:

**assumes** *inv1* *t* *inv2* *t*

**shows** *inv2* (*rbt-ins* *f* *k* *x* *t*) *bheight* (*rbt-ins* *f* *k* *x* *t*) = *bheight* *t*

*color-of* *t* = *B*  $\Longrightarrow$  *inv1* (*rbt-ins* *f* *k* *x* *t*) *inv1l* (*rbt-ins* *f* *k* *x* *t*)

$\langle$ *proof* $\rangle$

**end**

**context** *linorder* **begin**

**lemma** *ins-rbt-greater[simp]*: (*v*  $\ll$  | *rbt-ins* *f* (*k* :: 'a) *x* *t*) = (*v*  $\ll$  | *t*  $\wedge$  *k* > *v*)  
 $\langle$ *proof* $\rangle$

**lemma** *ins-rbt-less[simp]*: (*rbt-ins* *f* *k* *x* *t* |  $\ll$  *v*) = (*t* |  $\ll$  *v*  $\wedge$  *k* < *v*)  
 $\langle$ *proof* $\rangle$

**lemma** *ins-rbt-sorted[simp]*: *rbt-sorted* *t*  $\Longrightarrow$  *rbt-sorted* (*rbt-ins* *f* *k* *x* *t*)  
 $\langle$ *proof* $\rangle$

**lemma** *keys-ins*: *set* (*keys* (*rbt-ins* *f* *k* *v* *t*)) = { *k* }  $\cup$  *set* (*keys* *t*)  
 $\langle$ *proof* $\rangle$



**lemma** *rbt-lookup-ins*:

**fixes**  $k :: 'a$

**assumes** *rbt-sorted*  $t$

**shows**  $rbt\_lookup (rbt\_ins f k v t) x = ((rbt\_lookup t)(k \mid\rightarrow case\ rbt\_lookup\ t\ k$   
*of* *None*  $\Rightarrow v$

$\mid Some\ w \Rightarrow f\ k\ w\ v))\ x$

*<proof>*

**end**

**context** *ord* **begin**

**definition** *rbt-insert-with-key*  $:: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b)\ rbt \Rightarrow$   
 $('a, 'b)\ rbt$

**where** *rbt-insert-with-key*  $f\ k\ v\ t = paint\ B (rbt\_ins\ f\ k\ v\ t)$

**definition** *rbt-insertw-def*: *rbt-insert-with*  $f = rbt\_insert\_with\_key\ (\lambda\cdot. f)$

**definition** *rbt-insert*  $:: 'a \Rightarrow 'b \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$  **where**

*rbt-insert*  $= rbt\_insert\_with\_key\ (\lambda\cdot - nv. nv)$

**end**

**context** *linorder* **begin**

**lemma** *rbt-insertwk-rbt-sorted*: *rbt-sorted*  $t \Longrightarrow rbt\_sorted (rbt\_insert\_with\_key\ f (k$   
 $:: 'a)\ x\ t)$

*<proof>*

**theorem** *rbt-insertwk-is-rbt*:

**assumes** *inv*: *is-rbt*  $t$

**shows** *is-rbt*  $(rbt\_insert\_with\_key\ f\ k\ x\ t)$

*<proof>*

**lemma** *rbt-lookup-rbt-insertwk*:

**assumes** *rbt-sorted*  $t$

**shows**  $rbt\_lookup (rbt\_insert\_with\_key\ f\ k\ v\ t) x = ((rbt\_lookup t)(k \mid\rightarrow case\ rbt\_lookup\ t\ k$   
*of* *None*  $\Rightarrow v$

$\mid Some\ w \Rightarrow f\ k\ w\ v))\ x$

*<proof>*

**lemma** *rbt-insertw-rbt-sorted*: *rbt-sorted*  $t \Longrightarrow rbt\_sorted (rbt\_insert\_with\ f\ k\ v\ t)$

*<proof>*

**theorem** *rbt-insertw-is-rbt*: *is-rbt*  $t \Longrightarrow is\_rbt (rbt\_insert\_with\ f\ k\ v\ t)$

*<proof>*

**lemma** *rbt-lookup-rbt-insertw*:

*is-rbt*  $t \Longrightarrow$

$\text{rbt-lookup } (\text{rbt-insert-with } f \ k \ v \ t) =$   
 $(\text{rbt-lookup } t)(k \mapsto (\text{if } k \in \text{dom } (\text{rbt-lookup } t) \text{ then } f \ (\text{the } (\text{rbt-lookup } t \ k)) \ v$   
 $\text{else } v))$   
 ⟨proof⟩

**lemma** *rbt-insert-rbt-sorted*:  $\text{rbt-sorted } t \implies \text{rbt-sorted } (\text{rbt-insert } k \ v \ t)$   
 ⟨proof⟩

**theorem** *rbt-insert-is-rbt [simp]*:  $\text{is-rbt } t \implies \text{is-rbt } (\text{rbt-insert } k \ v \ t)$   
 ⟨proof⟩

**lemma** *rbt-lookup-rbt-insert*:  $\text{is-rbt } t \implies \text{rbt-lookup } (\text{rbt-insert } k \ v \ t) = (\text{rbt-lookup } t)(k \mapsto v)$   
 ⟨proof⟩

end

## 129.4 Deletion

**lemma** *bheight-paintR'[simp]*:  $\text{color-of } t = B \implies \text{bheight } (\text{paint } R \ t) = \text{bheight } t - 1$   
 ⟨proof⟩

The function definitions are based on the Haskell code by Stefan Kahrs at <http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html>.

**fun**

$\text{balance-left} :: ('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

**where**

$\text{balance-left } (\text{Branch } R \ a \ k \ x \ b) \ s \ y \ c = \text{Branch } R \ (\text{Branch } B \ a \ k \ x \ b) \ s \ y \ c \mid$   
 $\text{balance-left } \text{bl } k \ x \ (\text{Branch } B \ a \ s \ y \ b) = \text{balance } \text{bl } k \ x \ (\text{Branch } R \ a \ s \ y \ b) \mid$   
 $\text{balance-left } \text{bl } k \ x \ (\text{Branch } R \ (\text{Branch } B \ a \ s \ y \ b) \ t \ z \ c) = \text{Branch } R \ (\text{Branch } B \ \text{bl}$   
 $k \ x \ a) \ s \ y \ (\text{balance } b \ t \ z \ (\text{paint } R \ c)) \mid$   
 $\text{balance-left } t \ k \ x \ s = \text{Empty}$

**lemma** *balance-left-inv2-with-inv1*:

**assumes**  $\text{inv2 } lt \ \text{inv2 } rt \ \text{bheight } lt + 1 = \text{bheight } rt \ \text{inv1 } rt$

**shows**  $\text{bheight } (\text{balance-left } lt \ k \ v \ rt) = \text{bheight } lt + 1$

**and**  $\text{inv2 } (\text{balance-left } lt \ k \ v \ rt)$

⟨proof⟩

**lemma** *balance-left-inv2-app*:

**assumes**  $\text{inv2 } lt \ \text{inv2 } rt \ \text{bheight } lt + 1 = \text{bheight } rt \ \text{color-of } rt = B$

**shows**  $\text{inv2 } (\text{balance-left } lt \ k \ v \ rt)$

$\text{bheight } (\text{balance-left } lt \ k \ v \ rt) = \text{bheight } rt$

⟨proof⟩

**lemma** *balance-left-inv1*:  $[\text{inv1 } a; \text{inv1 } b; \text{color-of } b = B] \implies \text{inv1 } (\text{balance-left } a \ k \ x \ b)$

⟨proof⟩

**lemma** *balance-left-inv1*:  $\llbracket \text{inv1 } lt; \text{inv1 } rt \rrbracket \implies \text{inv1 } (\text{balance-left } lt \ k \ x \ rt)$   
 <proof>

**lemma** (in *linorder*) *balance-left-rbt-sorted*:  
 $\llbracket \text{rbt-sorted } l; \text{rbt-sorted } r; \text{rbt-less } k \ l; k \ll r \rrbracket \implies \text{rbt-sorted } (\text{balance-left } l \ k \ v \ r)$   
 <proof>

**context** *order* **begin**

**lemma** *balance-left-rbt-greater*:  
 fixes  $k :: 'a$   
 assumes  $k \ll a \ k \ll b \ k < x$   
 shows  $k \ll \text{balance-left } a \ x \ t \ b$   
 <proof>

**lemma** *balance-left-rbt-less*:  
 fixes  $k :: 'a$   
 assumes  $a \ll k \ b \ll k \ x < k$   
 shows  $\text{balance-left } a \ x \ t \ b \ll k$   
 <proof>

**end**

**lemma** *balance-left-in-tree*:  
 assumes  $\text{inv1 } l \ \text{inv1 } r \ \text{bheight } l + 1 = \text{bheight } r$   
 shows  $\text{entry-in-tree } k \ v \ (\text{balance-left } l \ a \ b \ r) = (\text{entry-in-tree } k \ v \ l \ \vee \ k = a \ \wedge \ v = b \ \vee \ \text{entry-in-tree } k \ v \ r)$   
 <proof>

**fun**

*balance-right* ::  $('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

**where**

*balance-right*  $a \ k \ x \ (\text{Branch } R \ b \ s \ y \ c) = \text{Branch } R \ a \ k \ x \ (\text{Branch } B \ b \ s \ y \ c) \mid$   
*balance-right*  $(\text{Branch } B \ a \ k \ x \ b) \ s \ y \ bl = \text{balance } (\text{Branch } R \ a \ k \ x \ b) \ s \ y \ bl \mid$   
*balance-right*  $(\text{Branch } R \ a \ k \ x \ (\text{Branch } B \ b \ s \ y \ c)) \ t \ z \ bl = \text{Branch } R \ (\text{balance } (\text{paint } R \ a) \ k \ x \ b) \ s \ y \ (\text{Branch } B \ c \ t \ z \ bl) \mid$   
*balance-right*  $t \ k \ x \ s = \text{Empty}$

**lemma** *balance-right-inv2-with-inv1*:

assumes  $\text{inv2 } lt \ \text{inv2 } rt \ \text{bheight } lt = \text{bheight } rt + 1 \ \text{inv1 } lt$   
 shows  $\text{inv2 } (\text{balance-right } lt \ k \ v \ rt) \ \wedge \ \text{bheight } (\text{balance-right } lt \ k \ v \ rt) = \text{bheight } lt$   
 <proof>

**lemma** *balance-right-inv1*:  $\llbracket \text{inv1 } a; \text{inv1 } b; \text{color-of } a = B \rrbracket \implies \text{inv1 } (\text{balance-right } a \ k \ x \ b)$   
 <proof>

**lemma** *balance-right-inv1l*:  $\llbracket \text{inv1 } lt; \text{inv1l } rt \rrbracket \implies \text{inv1l } (\text{balance-right } lt \ k \ x \ rt)$   
 $\langle \text{proof} \rangle$

**lemma** (in *linorder*) *balance-right-rbt-sorted*:  
 $\llbracket \text{rbt-sorted } l; \text{rbt-sorted } r; \text{rbt-less } k \ l; k \ll r \rrbracket \implies \text{rbt-sorted } (\text{balance-right } l \ k \ v \ r)$   
 $\langle \text{proof} \rangle$

**context** *order begin*

**lemma** *balance-right-rbt-greater*:  
**fixes**  $k :: 'a$   
**assumes**  $k \ll a \ k \ll b \ k < x$   
**shows**  $k \ll \text{balance-right } a \ x \ t \ b$   
 $\langle \text{proof} \rangle$

**lemma** *balance-right-rbt-less*:  
**fixes**  $k :: 'a$   
**assumes**  $a \ll k \ b \ll k \ x < k$   
**shows**  $\text{balance-right } a \ x \ t \ b \ll k$   
 $\langle \text{proof} \rangle$

**end**

**lemma** *balance-right-in-tree*:  
**assumes**  $\text{inv1 } l \ \text{inv1l } r \ \text{bheight } l = \text{bheight } r + 1 \ \text{inv2 } l \ \text{inv2 } r$   
**shows**  $\text{entry-in-tree } x \ y \ (\text{balance-right } l \ k \ v \ r) = (\text{entry-in-tree } x \ y \ l \vee x = k \wedge y = v \vee \text{entry-in-tree } x \ y \ r)$   
 $\langle \text{proof} \rangle$

**fun**

*combine*  $:: ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$

**where**

*combine* *Empty*  $x = x$   
| *combine*  $x \ \text{Empty} = x$   
| *combine*  $(\text{Branch } R \ a \ k \ x \ b) \ (\text{Branch } R \ c \ s \ y \ d) = (\text{case } (\text{combine } b \ c) \ \text{of}$   
 $\text{Branch } R \ b2 \ t \ z \ c2 \Rightarrow (\text{Branch } R \ (\text{Branch } R \ a \ k \ x \ b2) \ t \ z \ (\text{Branch } R \ c2 \ s \ y \ d))) \ |$   
 $bc \Rightarrow \text{Branch } R \ a \ k \ x \ (\text{Branch } R \ bc \ s \ y \ d))$   
| *combine*  $(\text{Branch } B \ a \ k \ x \ b) \ (\text{Branch } B \ c \ s \ y \ d) = (\text{case } (\text{combine } b \ c) \ \text{of}$   
 $\text{Branch } R \ b2 \ t \ z \ c2 \Rightarrow \text{Branch } R \ (\text{Branch } B \ a \ k \ x \ b2) \ t \ z \ (\text{Branch } B \ c2 \ s \ y \ d)) \ |$   
 $bc \Rightarrow \text{balance-left } a \ k \ x \ (\text{Branch } B \ bc \ s \ y \ d))$   
| *combine*  $a \ (\text{Branch } R \ b \ k \ x \ c) = \text{Branch } R \ (\text{combine } a \ b) \ k \ x \ c$   
| *combine*  $(\text{Branch } R \ a \ k \ x \ b) \ c = \text{Branch } R \ a \ k \ x \ (\text{combine } b \ c)$

**lemma** *combine-inv2*:

**assumes**  $\text{inv2 } lt \ \text{inv2 } rt \ \text{bheight } lt = \text{bheight } rt$   
**shows**  $\text{bheight } (\text{combine } lt \ rt) = \text{bheight } lt \ \text{inv2 } (\text{combine } lt \ rt)$

*<proof>*

**lemma** *combine-inv1*:

**assumes** *inv1 lt inv1 rt*

**shows** *color-of lt = B  $\implies$  color-of rt = B  $\implies$  inv1 (combine lt rt)*

*inv1l (combine lt rt)*

*<proof>*

**context** *linorder begin*

**lemma** *combine-rbt-greater[simp]*:

**fixes** *k :: 'a*

**assumes** *k  $\ll$  l k  $\ll$  r*

**shows** *k  $\ll$  combine l r*

*<proof>*

**lemma** *combine-rbt-less[simp]*:

**fixes** *k :: 'a*

**assumes** *l  $\ll$  k r  $\ll$  k*

**shows** *combine l r  $\ll$  k*

*<proof>*

**lemma** *combine-rbt-sorted*:

**fixes** *k :: 'a*

**assumes** *rbt-sorted l rbt-sorted r l  $\ll$  k k  $\ll$  r*

**shows** *rbt-sorted (combine l r)*

*<proof>*

**end**

**lemma** *combine-in-tree*:

**assumes** *inv2 l inv2 r bheight l = bheight r inv1 l inv1 r*

**shows** *entry-in-tree k v (combine l r) = (entry-in-tree k v l  $\vee$  entry-in-tree k v r)*

*<proof>*

**context** *ord begin*

**fun**

*rbt-del-from-left* :: *'a  $\Rightarrow$  ('a,'b) rbt  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  ('a,'b) rbt  $\Rightarrow$  ('a,'b) rbt* **and**

*rbt-del-from-right* :: *'a  $\Rightarrow$  ('a,'b) rbt  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  ('a,'b) rbt  $\Rightarrow$  ('a,'b) rbt* **and**

*rbt-del* :: *'a  $\Rightarrow$  ('a,'b) rbt  $\Rightarrow$  ('a,'b) rbt*

**where**

*rbt-del x Empty = Empty* |

*rbt-del x (Branch c a y s b) =*

*(if x < y then rbt-del-from-left x a y s b*

*else (if x > y then rbt-del-from-right x a y s b else combine a b))* |

*rbt-del-from-left x (Branch B lt z v rt) y s b = balance-left (rbt-del x (Branch B lt z v rt)) y s b* |

*rbt-del-from-left x a y s b = Branch R (rbt-del x a) y s b* |

$\text{rbt-del-from-right } x \ a \ y \ s \ (\text{Branch } B \ lt \ z \ v \ rt) = \text{balance-right } a \ y \ s \ (\text{rbt-del } x \ (\text{Branch } B \ lt \ z \ v \ rt)) \ |$

$\text{rbt-del-from-right } x \ a \ y \ s \ b = \text{Branch } R \ a \ y \ s \ (\text{rbt-del } x \ b)$

**end**

**context** *linorder* **begin**

**lemma**

**assumes**  $\text{inv2 } lt \ \text{inv1 } lt$

**shows**

$\llbracket \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; \text{inv1 } rt \rrbracket \implies$

$\text{inv2 } (\text{rbt-del-from-left } x \ lt \ k \ v \ rt) \wedge$

$\text{bheight } (\text{rbt-del-from-left } x \ lt \ k \ v \ rt) = \text{bheight } lt \wedge$

$(\text{color-of } lt = B \wedge \text{color-of } rt = B \wedge \text{inv1 } (\text{rbt-del-from-left } x \ lt \ k \ v \ rt) \vee$

$(\text{color-of } lt \neq B \vee \text{color-of } rt \neq B) \wedge \text{inv1l } (\text{rbt-del-from-left } x \ lt \ k \ v \ rt))$

**and**  $\llbracket \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; \text{inv1 } rt \rrbracket \implies$

$\text{inv2 } (\text{rbt-del-from-right } x \ lt \ k \ v \ rt) \wedge$

$\text{bheight } (\text{rbt-del-from-right } x \ lt \ k \ v \ rt) = \text{bheight } lt \wedge$

$(\text{color-of } lt = B \wedge \text{color-of } rt = B \wedge \text{inv1 } (\text{rbt-del-from-right } x \ lt \ k \ v \ rt) \vee$

$(\text{color-of } lt \neq B \vee \text{color-of } rt \neq B) \wedge \text{inv1l } (\text{rbt-del-from-right } x \ lt \ k \ v \ rt))$

**and**  $\text{rbt-del-inv1-inv2: } \text{inv2 } (\text{rbt-del } x \ lt) \wedge (\text{color-of } lt = R \wedge \text{bheight } (\text{rbt-del } x \ lt) = \text{bheight } lt \wedge \text{inv1 } (\text{rbt-del } x \ lt))$

$\vee \text{color-of } lt = B \wedge \text{bheight } (\text{rbt-del } x \ lt) = \text{bheight } lt - 1 \wedge \text{inv1l } (\text{rbt-del } x \ lt)$

*<proof>*

**lemma**

$\text{rbt-del-from-left-rbt-less: } \llbracket lt \ | \ll v; rt \ | \ll v; k < v \rrbracket \implies \text{rbt-del-from-left } x \ lt \ k \ y \ rt \ | \ll v$

**and**  $\text{rbt-del-from-right-rbt-less: } \llbracket lt \ | \ll v; rt \ | \ll v; k < v \rrbracket \implies \text{rbt-del-from-right } x \ lt \ k \ y \ rt \ | \ll v$

**and**  $\text{rbt-del-rbt-less: } lt \ | \ll v \implies \text{rbt-del } x \ lt \ | \ll v$

*<proof>*

**lemma**  $\text{rbt-del-from-left-rbt-greater: } \llbracket v \ll | lt; v \ll | rt; k > v \rrbracket \implies v \ll | \text{rbt-del-from-left } x \ lt \ k \ y \ rt$

**and**  $\text{rbt-del-from-right-rbt-greater: } \llbracket v \ll | lt; v \ll | rt; k > v \rrbracket \implies v \ll | \text{rbt-del-from-right } x \ lt \ k \ y \ rt$

**and**  $\text{rbt-del-rbt-greater: } v \ll | lt \implies v \ll | \text{rbt-del } x \ lt$

*<proof>*

**lemma**  $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \ | \ll k; k \ll | rt \rrbracket \implies \text{rbt-sorted } (\text{rbt-del-from-left } x \ lt \ k \ y \ rt)$

**and**  $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \ | \ll k; k \ll | rt \rrbracket \implies \text{rbt-sorted } (\text{rbt-del-from-right } x \ lt \ k \ y \ rt)$

**and**  $\text{rbt-del-rbt-sorted: } \text{rbt-sorted } lt \implies \text{rbt-sorted } (\text{rbt-del } x \ lt)$

*<proof>*

**lemma**  $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \ | \ll kt; kt \ll | rt; \text{inv1 } lt; \text{inv1 } rt; \text{inv2 } lt; \text{inv2 } rt \rrbracket$

$rt; \text{bheight } lt = \text{bheight } rt; x < kt \implies \text{entry-in-tree } k \ v \ (\text{rbt-del-from-left } x \ lt \ kt \ y \ rt) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \ v \ (\text{Branch } c \ lt \ kt \ y \ rt)))$   
**and**  $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \lll kt; kt \lll rt; \text{inv1 } lt; \text{inv1 } rt; \text{inv2 } lt; \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; x > kt \rrbracket \implies \text{entry-in-tree } k \ v \ (\text{rbt-del-from-right } x \ lt \ kt \ y \ rt) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \ v \ (\text{Branch } c \ lt \ kt \ y \ rt)))$   
**and**  $\text{rbt-del-in-tree: } \llbracket \text{rbt-sorted } t; \text{inv1 } t; \text{inv2 } t \rrbracket \implies \text{entry-in-tree } k \ v \ (\text{rbt-del } x \ t) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \ v \ t))$   
 $\langle \text{proof} \rangle$

**definition** (in *ord*) *rbt-delete* **where**  
 $\text{rbt-delete } k \ t = \text{paint } B \ (\text{rbt-del } k \ t)$

**theorem** *rbt-delete-is-rbt* [*simp*]: **assumes** *is-rbt* *t* **shows** *is-rbt* (*rbt-delete* *k* *t*)  
 $\langle \text{proof} \rangle$

**lemma** *rbt-delete-in-tree*:  
**assumes** *is-rbt* *t*  
**shows**  $\text{entry-in-tree } k \ v \ (\text{rbt-delete } x \ t) = (x \neq k \wedge \text{entry-in-tree } k \ v \ t)$   
 $\langle \text{proof} \rangle$

**lemma** *rbt-lookup-rbt-delete*:  
**assumes** *is-rbt*: *is-rbt* *t*  
**shows**  $\text{rbt-lookup } (\text{rbt-delete } k \ t) = (\text{rbt-lookup } t)|'(-\{k\})$   
 $\langle \text{proof} \rangle$

**end**

## 129.5 Modifying existing entries

**context** *ord* **begin**

**primrec**  
 $\text{rbt-map-entry} :: 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

**where**

$\text{rbt-map-entry } k \ f \ \text{Empty} = \text{Empty}$   
 $| \text{rbt-map-entry } k \ f \ (\text{Branch } c \ lt \ x \ v \ rt) =$   
     (if  $k < x$  then  $\text{Branch } c \ (\text{rbt-map-entry } k \ f \ lt) \ x \ v \ rt$   
     else if  $k > x$  then  $\text{Branch } c \ lt \ x \ v \ (\text{rbt-map-entry } k \ f \ rt)$   
     else  $\text{Branch } c \ lt \ x \ (f \ v) \ rt$ )

**lemma** *rbt-map-entry-color-of*:  $\text{color-of } (\text{rbt-map-entry } k \ f \ t) = \text{color-of } t \ \langle \text{proof} \rangle$

**lemma** *rbt-map-entry-inv1*:  $\text{inv1 } (\text{rbt-map-entry } k \ f \ t) = \text{inv1 } t \ \langle \text{proof} \rangle$

**lemma** *rbt-map-entry-inv2*:  $\text{inv2 } (\text{rbt-map-entry } k \ f \ t) = \text{inv2 } t \ \text{bheight } (\text{rbt-map-entry } k \ f \ t) = \text{bheight } t \ \langle \text{proof} \rangle$

**lemma** *rbt-map-entry-rbt-greater*:  $\text{rbt-greater } a \ (\text{rbt-map-entry } k \ f \ t) = \text{rbt-greater } a \ t \ \langle \text{proof} \rangle$

**lemma** *rbt-map-entry-rbt-less*:  $\text{rbt-less } a \ (\text{rbt-map-entry } k \ f \ t) = \text{rbt-less } a \ t \ \langle \text{proof} \rangle$

**lemma** *rbt-map-entry-rbt-sorted*:  $\text{rbt-sorted } (\text{rbt-map-entry } k \ f \ t) = \text{rbt-sorted } t$

*<proof>*

**theorem** *rbt-map-entry-is-rbt* [*simp*]: *is-rbt (rbt-map-entry k f t) = is-rbt t*  
*<proof>*

**end**

**theorem** (**in** *linorder*) *rbt-lookup-rbt-map-entry*:  
*rbt-lookup (rbt-map-entry k f t) = (rbt-lookup t)(k := map-option f (rbt-lookup t k))*  
*<proof>*

## 129.6 Mapping all entries

**primrec**

*map :: ('a ⇒ 'b ⇒ 'c) ⇒ ('a, 'b) rbt ⇒ ('a, 'c) rbt*

**where**

*map f Empty = Empty*

| *map f (Branch c lt k v rt) = Branch c (map f lt) k (f k v) (map f rt)*

**lemma** *map-entries* [*simp*]: *entries (map f t) = List.map (λ(k, v). (k, f k v)) (entries t)*  
*<proof>*

**lemma** *map-keys* [*simp*]: *keys (map f t) = keys t* *<proof>*

**lemma** *map-color-of*: *color-of (map f t) = color-of t* *<proof>*

**lemma** *map-inv1*: *inv1 (map f t) = inv1 t* *<proof>*

**lemma** *map-inv2*: *inv2 (map f t) = inv2 t bheight (map f t) = bheight t* *<proof>*

**context** *ord* **begin**

**lemma** *map-rbt-greater*: *rbt-greater k (map f t) = rbt-greater k t* *<proof>*

**lemma** *map-rbt-less*: *rbt-less k (map f t) = rbt-less k t* *<proof>*

**lemma** *map-rbt-sorted*: *rbt-sorted (map f t) = rbt-sorted t* *<proof>*

**theorem** *map-is-rbt* [*simp*]: *is-rbt (map f t) = is-rbt t*

*<proof>*

**end**

**theorem** (**in** *linorder*) *rbt-lookup-map*: *rbt-lookup (map f t) x = map-option (f x) (rbt-lookup t x)*  
*<proof>*

**hide-const** (**open**) *map*

## 129.7 Folding over entries

**definition** *fold :: ('a ⇒ 'b ⇒ 'c ⇒ 'c) ⇒ ('a, 'b) rbt ⇒ 'c ⇒ 'c* **where**  
*fold f t = List.fold (case-prod f) (entries t)*



**lemma** *fold-simps* [*simp*]:

*fold f Empty = id*

*fold f (Branch c lt k v rt) = fold f rt ∘ f k v ∘ fold f lt*

*<proof>*

**lemma** *fold-code* [*code*]:

*fold f Empty x = x*

*fold f (Branch c lt k v rt) x = fold f rt (f k v (fold f lt x))*

*<proof>*

**fun** *foldi* :: (*'c* ⇒ *bool*) ⇒ (*'a* ⇒ *'b* ⇒ *'c* ⇒ *'c*) ⇒ (*'a* :: *linorder*, *'b*) *rbt* ⇒ *'c* ⇒ *'c*

**where**

*foldi c f Empty s = s |*

*foldi c f (Branch col l k v r) s = (*

*if (c s) then*

*let s' = foldi c f l s in*

*if (c s') then*

*foldi c f r (f k v s')*

*else s'*

*else*

*s*

*)*

## 129.8 Bulkloading a tree

**definition** (*in ord*) *rbt-bulkload* :: (*'a* × *'b*) *list* ⇒ (*'a*, *'b*) *rbt* **where**

*rbt-bulkload xs = foldr (λ(k, v). rbt-insert k v) xs Empty*

**context** *linorder* **begin**

**lemma** *rbt-bulkload-is-rbt* [*simp*, *intro*]:

*is-rbt (rbt-bulkload xs)*

*<proof>*

**lemma** *rbt-lookup-rbt-bulkload*:

*rbt-lookup (rbt-bulkload xs) = map-of xs*

*<proof>*

**end**

## 129.9 Building a RBT from a sorted list

These functions have been adapted from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

**fun** *rbtreeify-f* :: *nat* ⇒ (*'a* × *'b*) *list* ⇒ (*'a*, *'b*) *rbt* × (*'a* × *'b*) *list*

**and** *rbtreeify-g* :: *nat* ⇒ (*'a* × *'b*) *list* ⇒ (*'a*, *'b*) *rbt* × (*'a* × *'b*) *list*

**where**

*rbtreeify-f n kvs =*

*(if n = 0 then (Empty, kvs)*

```

else if n = 1 then
  case kvs of (k, v) # kvs' ⇒ (Branch R Empty k v Empty, kvs')
else if (n mod 2 = 0) then
  case rbtreeify-f (n div 2) kvs of (t1, (k, v) # kvs') ⇒
    apfst (Branch B t1 k v) (rbtreeify-g (n div 2) kvs')
  else case rbtreeify-f (n div 2) kvs of (t1, (k, v) # kvs') ⇒
    apfst (Branch B t1 k v) (rbtreeify-f (n div 2) kvs')

```

```

| rbtreeify-g n kvs =
  (if n = 0 ∨ n = 1 then (Empty, kvs)
  else if n mod 2 = 0 then
    case rbtreeify-g (n div 2) kvs of (t1, (k, v) # kvs') ⇒
      apfst (Branch B t1 k v) (rbtreeify-g (n div 2) kvs')
    else case rbtreeify-f (n div 2) kvs of (t1, (k, v) # kvs') ⇒
      apfst (Branch B t1 k v) (rbtreeify-g (n div 2) kvs')

```

**definition** *rbtreeify* :: ('a × 'b) list ⇒ ('a, 'b) rbt  
**where** *rbtreeify* kvs = fst (rbtreeify-g (Suc (length kvs)) kvs)

**declare** *rbtreeify-f.simps* [simp del] *rbtreeify-g.simps* [simp del]

**lemma** *rbtreeify-f-code* [code]:

```

rbtreeify-f n kvs =
  (if n = 0 then (Empty, kvs)
  else if n = 1 then
    case kvs of (k, v) # kvs' ⇒
      (Branch R Empty k v Empty, kvs')
  else let (n', r) = Euclidean-Rings.divmod-nat n 2 in
    if r = 0 then
      case rbtreeify-f n' kvs of (t1, (k, v) # kvs') ⇒
        apfst (Branch B t1 k v) (rbtreeify-g n' kvs')
      else case rbtreeify-f n' kvs of (t1, (k, v) # kvs') ⇒
        apfst (Branch B t1 k v) (rbtreeify-f n' kvs')

```

⟨proof⟩

**lemma** *rbtreeify-g-code* [code]:

```

rbtreeify-g n kvs =
  (if n = 0 ∨ n = 1 then (Empty, kvs)
  else let (n', r) = Euclidean-Rings.divmod-nat n 2 in
    if r = 0 then
      case rbtreeify-g n' kvs of (t1, (k, v) # kvs') ⇒
        apfst (Branch B t1 k v) (rbtreeify-g n' kvs')
      else case rbtreeify-f n' kvs of (t1, (k, v) # kvs') ⇒
        apfst (Branch B t1 k v) (rbtreeify-g n' kvs')

```

⟨proof⟩

**lemma** *Suc-double-half*: Suc (2 \* n) div 2 = n

⟨proof⟩

**lemma** *div2-plus-div2*:  $n \text{ div } 2 + n \text{ div } 2 = (n :: \text{nat}) - n \text{ mod } 2$   
 ⟨proof⟩

**lemma** *rbtreeify-f-rec-aux-lemma*:  
 $\llbracket k - n \text{ div } 2 = \text{Suc } k'; n \leq k; n \text{ mod } 2 = \text{Suc } 0 \rrbracket$   
 $\implies k' - n \text{ div } 2 = k - n$   
 ⟨proof⟩

**lemma** *rbtreeify-f-simps*:  
 $\text{rbtreeify-f } 0 \text{ kvs} = (\text{Empty}, \text{kvs})$   
 $\text{rbtreeify-f } (\text{Suc } 0) ((k, v) \# \text{kvs}) =$   
 $(\text{Branch } R \text{ Empty } k \text{ v Empty}, \text{kvs})$   
 $0 < n \implies \text{rbtreeify-f } (2 * n) \text{ kvs} =$   
 $(\text{case } \text{rbtreeify-f } n \text{ kvs of } (t1, (k, v) \# \text{kvs}') \implies$   
 $\text{apfst } (\text{Branch } B \text{ t1 } k \text{ v}) (\text{rbtreeify-g } n \text{ kvs}'))$   
 $0 < n \implies \text{rbtreeify-f } (\text{Suc } (2 * n)) \text{ kvs} =$   
 $(\text{case } \text{rbtreeify-f } n \text{ kvs of } (t1, (k, v) \# \text{kvs}') \implies$   
 $\text{apfst } (\text{Branch } B \text{ t1 } k \text{ v}) (\text{rbtreeify-f } n \text{ kvs}'))$   
 ⟨proof⟩

**lemma** *rbtreeify-g-simps*:  
 $\text{rbtreeify-g } 0 \text{ kvs} = (\text{Empty}, \text{kvs})$   
 $\text{rbtreeify-g } (\text{Suc } 0) \text{ kvs} = (\text{Empty}, \text{kvs})$   
 $0 < n \implies \text{rbtreeify-g } (2 * n) \text{ kvs} =$   
 $(\text{case } \text{rbtreeify-g } n \text{ kvs of } (t1, (k, v) \# \text{kvs}') \implies$   
 $\text{apfst } (\text{Branch } B \text{ t1 } k \text{ v}) (\text{rbtreeify-f } n \text{ kvs}'))$   
 $0 < n \implies \text{rbtreeify-g } (\text{Suc } (2 * n)) \text{ kvs} =$   
 $(\text{case } \text{rbtreeify-f } n \text{ kvs of } (t1, (k, v) \# \text{kvs}') \implies$   
 $\text{apfst } (\text{Branch } B \text{ t1 } k \text{ v}) (\text{rbtreeify-g } n \text{ kvs}'))$   
 ⟨proof⟩

**declare** *rbtreeify-f-simps*[simp] *rbtreeify-g-simps*[simp]

**lemma** *length-rbtreeify-f*:  $n \leq \text{length } \text{kvs}$   
 $\implies \text{length } (\text{snd } (\text{rbtreeify-f } n \text{ kvs})) = \text{length } \text{kvs} - n$   
**and** *length-rbtreeify-g*:  $\llbracket 0 < n; n \leq \text{Suc } (\text{length } \text{kvs}) \rrbracket$   
 $\implies \text{length } (\text{snd } (\text{rbtreeify-g } n \text{ kvs})) = \text{Suc } (\text{length } \text{kvs}) - n$   
 ⟨proof⟩

**lemma** *rbtreeify-induct* [consumes 1, case-names *f-0 f-1 f-even f-odd g-0 g-1 g-even g-odd*]:

**fixes** *P Q*  
**defines** *f0* ==  $(\bigwedge \text{kvs}. P \ 0 \ \text{kvs})$   
**and** *f1* ==  $(\bigwedge k \ v \ \text{kvs}. P \ (\text{Suc } 0) \ ((k, v) \# \ \text{kvs}))$   
**and** *feven* ==  
 $(\bigwedge n \ \text{kvs} \ t \ k \ v \ \text{kvs}'. \llbracket n > 0; n \leq \text{length } \text{kvs}; P \ n \ \text{kvs};$   
 $\text{rbtreeify-f } n \ \text{kvs} = (t, (k, v) \# \ \text{kvs}'); n \leq \text{Suc } (\text{length } \text{kvs}'); Q \ n \ \text{kvs}' \rrbracket$   
 $\implies P \ (2 * n) \ \text{kvs})$   
**and** *fodd* ==

```

  (∧ n kvs t k v kvs'. [ n > 0; n ≤ length kvs; P n kvs;
    rbtreeify-f n kvs = (t, (k, v) # kvs'); n ≤ length kvs'; P n kvs' ])
  ⇒ P (Suc (2 * n)) kvs)
and g0 == (∧ kvs. Q 0 kvs)
and g1 == (∧ kvs. Q (Suc 0) kvs)
and geven ==
  (∧ n kvs t k v kvs'. [ n > 0; n ≤ Suc (length kvs); Q n kvs;
    rbtreeify-g n kvs = (t, (k, v) # kvs'); n ≤ Suc (length kvs'); Q n kvs' ])
  ⇒ Q (2 * n) kvs)
and godd ==
  (∧ n kvs t k v kvs'. [ n > 0; n ≤ length kvs; P n kvs;
    rbtreeify-f n kvs = (t, (k, v) # kvs'); n ≤ Suc (length kvs'); Q n kvs' ])
  ⇒ Q (Suc (2 * n)) kvs)
shows [ n ≤ length kvs;
  PROP f0; PROP f1; PROP feven; PROP fodd;
  PROP g0; PROP g1; PROP geven; PROP godd ]
  ⇒ P n kvs
and [ n ≤ Suc (length kvs);
  PROP f0; PROP f1; PROP feven; PROP fodd;
  PROP g0; PROP g1; PROP geven; PROP godd ]
  ⇒ Q n kvs
⟨proof⟩

```

```

lemma inv1-rbtreeify-f: n ≤ length kvs
  ⇒ inv1 (fst (rbtreeify-f n kvs))
and inv1-rbtreeify-g: n ≤ Suc (length kvs)
  ⇒ inv1 (fst (rbtreeify-g n kvs))
⟨proof⟩

```

```

fun plog2 :: nat ⇒ nat
where plog2 n = (if n ≤ 1 then 0 else plog2 (n div 2) + 1)

```

```

declare plog2.simps [simp del]

```

```

lemma plog2-simps [simp]:
  plog2 0 = 0 plog2 (Suc 0) = 0
  0 < n ⇒ plog2 (2 * n) = 1 + plog2 n
  0 < n ⇒ plog2 (Suc (2 * n)) = 1 + plog2 n
⟨proof⟩

```

```

lemma bheight-rbtreeify-f: n ≤ length kvs
  ⇒ bheight (fst (rbtreeify-f n kvs)) = plog2 n
and bheight-rbtreeify-g: n ≤ Suc (length kvs)
  ⇒ bheight (fst (rbtreeify-g n kvs)) = plog2 n
⟨proof⟩

```

```

lemma bheight-rbtreeify-f-eq-plog2I:
  [ rbtreeify-f n kvs = (t, kvs'); n ≤ length kvs ]
  ⇒ bheight t = plog2 n

```

*<proof>*

**lemma** *bheight-rbtreeify-g-eq-plog2I:*

$\llbracket \text{rbtreeify-g } n \text{ kvs} = (t, \text{kvs}^{\wedge}); n \leq \text{Suc } (\text{length } \text{kvs}) \rrbracket$   
 $\implies \text{bheight } t = \text{plog2 } n$

*<proof>*

**hide-const (open)** *plog2*

**lemma** *inv2-rbtreeify-f: n ≤ length kvs*

$\implies \text{inv2 } (\text{fst } (\text{rbtreeify-f } n \text{ kvs}))$

**and** *inv2-rbtreeify-g: n ≤ Suc (length kvs)*

$\implies \text{inv2 } (\text{fst } (\text{rbtreeify-g } n \text{ kvs}))$

*<proof>*

**lemma** *n ≤ length kvs ⇒ True*

**and** *color-of-rbtreeify-g:*

$\llbracket n \leq \text{Suc } (\text{length } \text{kvs}); 0 < n \rrbracket$

$\implies \text{color-of } (\text{fst } (\text{rbtreeify-g } n \text{ kvs})) = B$

*<proof>*

**lemma** *entries-rbtreeify-f-append:*

$n \leq \text{length } \text{kvs}$

$\implies \text{entries } (\text{fst } (\text{rbtreeify-f } n \text{ kvs})) @ \text{snd } (\text{rbtreeify-f } n \text{ kvs}) = \text{kvs}$

**and** *entries-rbtreeify-g-append:*

$n \leq \text{Suc } (\text{length } \text{kvs})$

$\implies \text{entries } (\text{fst } (\text{rbtreeify-g } n \text{ kvs})) @ \text{snd } (\text{rbtreeify-g } n \text{ kvs}) = \text{kvs}$

*<proof>*

**lemma** *length-entries-rbtreeify-f:*

$n \leq \text{length } \text{kvs} \implies \text{length } (\text{entries } (\text{fst } (\text{rbtreeify-f } n \text{ kvs}))) = n$

**and** *length-entries-rbtreeify-g:*

$n \leq \text{Suc } (\text{length } \text{kvs}) \implies \text{length } (\text{entries } (\text{fst } (\text{rbtreeify-g } n \text{ kvs}))) = n - 1$

*<proof>*

**lemma** *rbtreeify-f-conv-drop:*

$n \leq \text{length } \text{kvs} \implies \text{snd } (\text{rbtreeify-f } n \text{ kvs}) = \text{drop } n \text{ kvs}$

*<proof>*

**lemma** *rbtreeify-g-conv-drop:*

$n \leq \text{Suc } (\text{length } \text{kvs}) \implies \text{snd } (\text{rbtreeify-g } n \text{ kvs}) = \text{drop } (n - 1) \text{ kvs}$

*<proof>*

**lemma** *entries-rbtreeify-f [simp]:*

$n \leq \text{length } \text{kvs} \implies \text{entries } (\text{fst } (\text{rbtreeify-f } n \text{ kvs})) = \text{take } n \text{ kvs}$

*<proof>*

**lemma** *entries-rbtreeify-g [simp]:*

$n \leq \text{Suc } (\text{length } \text{kvs}) \implies$

$entries (fst (rbtreeify-g n kvs)) = take (n - 1) kvs$   
 ⟨proof⟩

**lemma** *keys-rbtreeify-f* [simp]:  $n \leq length\ kvs$   
 $\implies keys (fst (rbtreeify-f n kvs)) = take\ n (map\ fst\ kvs)$   
 ⟨proof⟩

**lemma** *keys-rbtreeify-g* [simp]:  $n \leq Suc (length\ kvs)$   
 $\implies keys (fst (rbtreeify-g n kvs)) = take (n - 1) (map\ fst\ kvs)$   
 ⟨proof⟩

**lemma** *rbtreeify-fD*:  
 $\llbracket rbtreeify-f\ n\ kvs = (t, kvs');\ n \leq length\ kvs \rrbracket$   
 $\implies entries\ t = take\ n\ kvs \wedge kvs' = drop\ n\ kvs$   
 ⟨proof⟩

**lemma** *rbtreeify-gD*:  
 $\llbracket rbtreeify-g\ n\ kvs = (t, kvs');\ n \leq Suc (length\ kvs) \rrbracket$   
 $\implies entries\ t = take (n - 1) kvs \wedge kvs' = drop (n - 1) kvs$   
 ⟨proof⟩

**lemma** *entries-rbtreeify* [simp]:  $entries (rbtreeify\ kvs) = kvs$   
 ⟨proof⟩

**context** *linorder begin*

**lemma** *rbt-sorted-rbtreeify-f*:  
 $\llbracket n \leq length\ kvs; sorted (map\ fst\ kvs); distinct (map\ fst\ kvs) \rrbracket$   
 $\implies rbt-sorted (fst (rbtreeify-f\ n\ kvs))$   
**and** *rbt-sorted-rbtreeify-g*:  
 $\llbracket n \leq Suc (length\ kvs); sorted (map\ fst\ kvs); distinct (map\ fst\ kvs) \rrbracket$   
 $\implies rbt-sorted (fst (rbtreeify-g\ n\ kvs))$   
 ⟨proof⟩

**lemma** *rbt-sorted-rbtreeify*:  
 $\llbracket sorted (map\ fst\ kvs); distinct (map\ fst\ kvs) \rrbracket \implies rbt-sorted (rbtreeify\ kvs)$   
 ⟨proof⟩

**lemma** *is-rbt-rbtreeify*:  
 $\llbracket sorted (map\ fst\ kvs); distinct (map\ fst\ kvs) \rrbracket$   
 $\implies is-rbt (rbtreeify\ kvs)$   
 ⟨proof⟩

**lemma** *rbt-lookup-rbtreeify*:  
 $\llbracket sorted (map\ fst\ kvs); distinct (map\ fst\ kvs) \rrbracket \implies$   
 $rbt-lookup (rbtreeify\ kvs) = map-of\ kvs$   
 ⟨proof⟩

**end**

Functions to compare the height of two rbt trees, taken from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

**fun** *skip-red* :: ('a, 'b) rbt ⇒ ('a, 'b) rbt

**where**

*skip-red* (Branch color.R l k v r) = l  
| *skip-red* t = t

**definition** *skip-black* :: ('a, 'b) rbt ⇒ ('a, 'b) rbt

**where**

*skip-black* t = (let t' = *skip-red* t in case t' of Branch color.B l k v r ⇒ l | - ⇒ t')

**datatype** *compare* = LT | GT | EQ

**partial-function** (*tailrec*) *compare-height* :: ('a, 'b) rbt ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt ⇒ *compare*

**where**

*compare-height* s<sub>x</sub> s t t<sub>x</sub> =  
(case (*skip-red* s<sub>x</sub>, *skip-red* s, *skip-red* t, *skip-red* t<sub>x</sub>) of  
  (Branch - s<sub>x</sub>' - - -, Branch - s' - - -, Branch - t' - - -, Branch - t<sub>x</sub>' - - -) ⇒  
    *compare-height* (*skip-black* s<sub>x</sub>') s' t' (*skip-black* t<sub>x</sub>')  
  | (-, rbt.Empty, -, Branch - - - -) ⇒ LT  
  | (Branch - - - - -, -, rbt.Empty, -) ⇒ GT  
  | (Branch - s<sub>x</sub>' - - -, Branch - s' - - -, Branch - t' - - -, rbt.Empty) ⇒  
    *compare-height* (*skip-black* s<sub>x</sub>') s' t' rbt.Empty  
  | (rbt.Empty, Branch - s' - - -, Branch - t' - - -, Branch - t<sub>x</sub>' - - -) ⇒  
    *compare-height* rbt.Empty s' t' (*skip-black* t<sub>x</sub>')  
  | - ⇒ EQ)

**declare** *compare-height.simps* [code]

**hide-type** (**open**) *compare*

**hide-const** (**open**)

*compare-height skip-black skip-red LT GT EQ case-compare rec-compare*  
*Abs-compare Rep-compare*

**hide-fact** (**open**)

*Abs-compare-cases Abs-compare-induct Abs-compare-inject Abs-compare-inverse*  
*Rep-compare Rep-compare-cases Rep-compare-induct Rep-compare-inject Rep-compare-inverse*  
*compare.simps compare.exhaust compare.induct compare.rec compare.simps*  
*compare.size compare.case-cong compare.case-cong-weak compare.case*  
*compare.nchotomy compare.split compare.split-asm compare.eq.refl compare.eq.simps*  
*equal-compare-def*  
*skip-red.simps skip-red.cases skip-red.induct*  
*skip-black-def*  
*compare-height.simps*

## 129.10 union and intersection of sorted associative lists

**context** *ord* **begin**

**function** *sunion-with* :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a × 'b) list ⇒ ('a × 'b) list ⇒ ('a × 'b) list

**where**

*sunion-with* f ((k, v) # as) ((k', v') # bs) =  
 (if k > k' then (k', v') # *sunion-with* f ((k, v) # as) bs  
 else if k < k' then (k, v) # *sunion-with* f as ((k', v') # bs)  
 else (k, f k v v') # *sunion-with* f as bs)

| *sunion-with* f [] bs = bs

| *sunion-with* f as [] = as

⟨proof⟩

**termination** ⟨proof⟩

**function** *sinter-with* :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a × 'b) list ⇒ ('a × 'b) list ⇒ ('a × 'b) list

**where**

*sinter-with* f ((k, v) # as) ((k', v') # bs) =  
 (if k > k' then *sinter-with* f ((k, v) # as) bs  
 else if k < k' then *sinter-with* f as ((k', v') # bs)  
 else (k, f k v v') # *sinter-with* f as bs)

| *sinter-with* f [] - = []

| *sinter-with* f - [] = []

⟨proof⟩

**termination** ⟨proof⟩

**end**

**declare** *ord.sunion-with.simps* [code] *ord.sinter-with.simps*[code]

**context** *linorder* **begin**

**lemma** *set-fst-sunion-with*:

*set* (map fst (*sunion-with* f xs ys)) = *set* (map fst xs) ∪ *set* (map fst ys)

⟨proof⟩

**lemma** *sorted-sunion-with* [simp]:

[[ sorted (map fst xs); sorted (map fst ys) ]]  
 ⇒ sorted (map fst (*sunion-with* f xs ys))

⟨proof⟩

**lemma** *distinct-sunion-with* [simp]:

[[ distinct (map fst xs); distinct (map fst ys); sorted (map fst xs); sorted (map fst ys) ]]

⇒ distinct (map fst (*sunion-with* f xs ys))

⟨proof⟩

**lemma** *map-of-sunion-with*:

[[ sorted (map fst xs); sorted (map fst ys) ]]  
 ⇒ map-of (*sunion-with* f xs ys) k =



(*case map-of xs k of None*  $\Rightarrow$  *map-of ys k*  
 | *Some v*  $\Rightarrow$  *case map-of ys k of None*  $\Rightarrow$  *Some v*  
 | *Some w*  $\Rightarrow$  *Some (f k v w)*)  
 <proof>

**lemma** *set-fst-sinter-with* [*simp*]:  
 $\llbracket$  *sorted (map fst xs); sorted (map fst ys)*  $\rrbracket$   
 $\implies$  *set (map fst (sinter-with f xs ys)) = set (map fst xs)  $\cap$  set (map fst ys)*  
 <proof>

**lemma** *set-fst-sinter-with-subset1*:  
*set (map fst (sinter-with f xs ys))  $\subseteq$  set (map fst xs)*  
 <proof>

**lemma** *set-fst-sinter-with-subset2*:  
*set (map fst (sinter-with f xs ys))  $\subseteq$  set (map fst ys)*  
 <proof>

**lemma** *sorted-sinter-with* [*simp*]:  
 $\llbracket$  *sorted (map fst xs); sorted (map fst ys)*  $\rrbracket$   
 $\implies$  *sorted (map fst (sinter-with f xs ys))*  
 <proof>

**lemma** *distinct-sinter-with* [*simp*]:  
 $\llbracket$  *distinct (map fst xs); distinct (map fst ys)*  $\rrbracket$   
 $\implies$  *distinct (map fst (sinter-with f xs ys))*  
 <proof>

**lemma** *map-of-sinter-with*:  
 $\llbracket$  *sorted (map fst xs); sorted (map fst ys)*  $\rrbracket$   
 $\implies$  *map-of (sinter-with f xs ys) k =*  
 (*case map-of xs k of None*  $\Rightarrow$  *None* | *Some v*  $\Rightarrow$  *map-option (f k v) (map-of ys k)*)  
 <proof>

**end**

**lemma** *distinct-map-of-rev*: *distinct (map fst xs)  $\implies$  map-of (rev xs) = map-of xs*  
 <proof>

**lemma** *map-map-filter*:  
*map f (List.map-filter g xs) = List.map-filter (map-option f  $\circ$  g) xs*  
 <proof>

**lemma** *map-filter-map-option-const*:  
*List.map-filter ( $\lambda x. \text{map-option } (\lambda y. f x) (g (f x))$ ) xs = filter ( $\lambda x. g x \neq \text{None}$ )*  
*(map f xs)*  
 <proof>

**lemma** *set-map-filter*:  $set (List.map-filter P xs) = the \ ' (P \ set xs - \{None\})$   
 ⟨proof⟩

**definition** *is-rbt-empty* ::  $(\ 'a, \ 'b) \ rbt \Rightarrow \ bool$  **where**  
 $is-rbt-empty \ t \ \longleftrightarrow \ (case \ t \ of \ RBT-Impl.Empty \Rightarrow \ True \ | \ - \Rightarrow \ False)$

**lemma** *is-rbt-empty-prop[simp]*:  $is-rbt-empty \ t \ \longleftrightarrow \ t = RBT-Impl.Empty$   
 ⟨proof⟩

**definition** *small-rbt* ::  $(\ 'a, \ 'b) \ rbt \Rightarrow \ bool$  **where**  
 $small-rbt \ t \ \longleftrightarrow \ bheight \ t < 4$

**definition** *flip-rbt* ::  $(\ 'a, \ 'b) \ rbt \Rightarrow (\ 'a, \ 'b) \ rbt \Rightarrow \ bool$  **where**  
 $flip-rbt \ t1 \ t2 \ \longleftrightarrow \ bheight \ t2 < bheight \ t1$

**abbreviation** (input) *MR* **where**  $MR \ l \ a \ b \ r \equiv Branch \ RBT-Impl.R \ l \ a \ b \ r$   
**abbreviation** (input) *MB* **where**  $MB \ l \ a \ b \ r \equiv Branch \ RBT-Impl.B \ l \ a \ b \ r$

**fun** *rbt-baliL* ::  $(\ 'a, \ 'b) \ rbt \Rightarrow \ 'a \Rightarrow \ 'b \Rightarrow (\ 'a, \ 'b) \ rbt \Rightarrow (\ 'a, \ 'b) \ rbt$  **where**  
 $rbt-baliL \ (MR \ (MR \ t1 \ a \ b \ t2) \ a' \ b' \ t3) \ a'' \ b'' \ t4 = MR \ (MB \ t1 \ a \ b \ t2) \ a' \ b' \ (MB \ t3 \ a'' \ b'' \ t4)$   
 $| \ rbt-baliL \ (MR \ t1 \ a \ b \ (MR \ t2 \ a' \ b' \ t3)) \ a'' \ b'' \ t4 = MR \ (MB \ t1 \ a \ b \ t2) \ a' \ b' \ (MB \ t3 \ a'' \ b'' \ t4)$   
 $| \ rbt-baliL \ t1 \ a \ b \ t2 = MB \ t1 \ a \ b \ t2$

**fun** *rbt-baliR* ::  $(\ 'a, \ 'b) \ rbt \Rightarrow \ 'a \Rightarrow \ 'b \Rightarrow (\ 'a, \ 'b) \ rbt \Rightarrow (\ 'a, \ 'b) \ rbt$  **where**  
 $rbt-baliR \ t1 \ a \ b \ (MR \ t2 \ a' \ b' \ (MR \ t3 \ a'' \ b'' \ t4)) = MR \ (MB \ t1 \ a \ b \ t2) \ a' \ b' \ (MB \ t3 \ a'' \ b'' \ t4)$   
 $| \ rbt-baliR \ t1 \ a \ b \ (MR \ (MR \ t2 \ a' \ b' \ t3) \ a'' \ b'' \ t4) = MR \ (MB \ t1 \ a \ b \ t2) \ a' \ b' \ (MB \ t3 \ a'' \ b'' \ t4)$   
 $| \ rbt-baliR \ t1 \ a \ b \ t2 = MB \ t1 \ a \ b \ t2$

**fun** *rbt-baldL* ::  $(\ 'a, \ 'b) \ rbt \Rightarrow \ 'a \Rightarrow \ 'b \Rightarrow (\ 'a, \ 'b) \ rbt \Rightarrow (\ 'a, \ 'b) \ rbt$  **where**  
 $rbt-baldL \ (MR \ t1 \ a \ b \ t2) \ a' \ b' \ t3 = MR \ (MB \ t1 \ a \ b \ t2) \ a' \ b' \ t3$   
 $| \ rbt-baldL \ t1 \ a \ b \ (MB \ t2 \ a' \ b' \ t3) = rbt-baliR \ t1 \ a \ b \ (MR \ t2 \ a' \ b' \ t3)$   
 $| \ rbt-baldL \ t1 \ a \ b \ (MR \ (MB \ t2 \ a' \ b' \ t3) \ a'' \ b'' \ t4) =$   
 $MR \ (MB \ t1 \ a \ b \ t2) \ a' \ b' \ (rbt-baliR \ t3 \ a'' \ b'' \ (paint \ RBT-Impl.R \ t4))$   
 $| \ rbt-baldL \ t1 \ a \ b \ t2 = MR \ t1 \ a \ b \ t2$

**fun** *rbt-baldR* ::  $(\ 'a, \ 'b) \ rbt \Rightarrow \ 'a \Rightarrow \ 'b \Rightarrow (\ 'a, \ 'b) \ rbt \Rightarrow (\ 'a, \ 'b) \ rbt$  **where**  
 $rbt-baldR \ t1 \ a \ b \ (MR \ t2 \ a' \ b' \ t3) = MR \ t1 \ a \ b \ (MB \ t2 \ a' \ b' \ t3)$   
 $| \ rbt-baldR \ (MB \ t1 \ a \ b \ t2) \ a' \ b' \ t3 = rbt-baliL \ (MR \ t1 \ a \ b \ t2) \ a' \ b' \ t3$   
 $| \ rbt-baldR \ (MR \ t1 \ a \ b \ (MB \ t2 \ a' \ b' \ t3)) \ a'' \ b'' \ t4 =$   
 $MR \ (rbt-baliL \ (paint \ RBT-Impl.R \ t1) \ a \ b \ t2) \ a' \ b' \ (MB \ t3 \ a'' \ b'' \ t4)$   
 $| \ rbt-baldR \ t1 \ a \ b \ t2 = MR \ t1 \ a \ b \ t2$

**fun** *rbt-app* ::  $(\ 'a, \ 'b) \ rbt \Rightarrow (\ 'a, \ 'b) \ rbt \Rightarrow (\ 'a, \ 'b) \ rbt$  **where**

```

  rbt-app RBT-Impl.Empty t = t
| rbt-app t RBT-Impl.Empty = t
| rbt-app (MR t1 a b t2) (MR t3 a'' b'' t4) = (case rbt-app t2 t3 of
  MR u2 a' b' u3 => (MR (MR t1 a b u2) a' b' (MR u3 a'' b'' t4))
| t23 => MR t1 a b (MR t23 a'' b'' t4))
| rbt-app (MB t1 a b t2) (MB t3 a'' b'' t4) = (case rbt-app t2 t3 of
  MR u2 a' b' u3 => MR (MB t1 a b u2) a' b' (MB u3 a'' b'' t4)
| t23 => rbt-baldL t1 a b (MB t23 a'' b'' t4))
| rbt-app t1 (MR t2 a b t3) = MR (rbt-app t1 t2) a b t3
| rbt-app (MR t1 a b t2) t3 = MR t1 a b (rbt-app t2 t3)

```

```

fun rbt-joinL :: ('a, 'b) rbt => 'a => 'b => ('a, 'b) rbt => ('a, 'b) rbt where
  rbt-joinL l a b r = (if bheight l >= bheight r then MR l a b r
  else case r of MB l' a' b' r' => rbt-baliL (rbt-joinL l a b l') a' b' r'
| MR l' a' b' r' => MR (rbt-joinL l a b l') a' b' r')

```

```

declare rbt-joinL.simps[simp del]

```

```

fun rbt-joinR :: ('a, 'b) rbt => 'a => 'b => ('a, 'b) rbt => ('a, 'b) rbt where
  rbt-joinR l a b r = (if bheight l <= bheight r then MR l a b r
  else case l of MB l' a' b' r' => rbt-baliR l' a' b' (rbt-joinR r' a b r)
| MR l' a' b' r' => MR l' a' b' (rbt-joinR r' a b r))

```

```

declare rbt-joinR.simps[simp del]

```

```

definition rbt-join :: ('a, 'b) rbt => 'a => 'b => ('a, 'b) rbt => ('a, 'b) rbt where
  rbt-join l a b r =
  (let bhl = bheight l; bhr = bheight r
  in if bhl > bhr
  then paint RBT-Impl.B (rbt-joinR l a b r)
  else if bhl < bhr
  then paint RBT-Impl.B (rbt-joinL l a b r)
  else MB l a b r)

```

```

lemma size-paint[simp]: size (paint c t) = size t
  <proof>

```

```

lemma size-baliL[simp]: size (rbt-baliL t1 a b t2) = Suc (size t1 + size t2)
  <proof>

```

```

lemma size-baliR[simp]: size (rbt-baliR t1 a b t2) = Suc (size t1 + size t2)
  <proof>

```

```

lemma size-baldL[simp]: size (rbt-baldL t1 a b t2) = Suc (size t1 + size t2)
  <proof>

```

```

lemma size-baldR[simp]: size (rbt-baldR t1 a b t2) = Suc (size t1 + size t2)
  <proof>

```

**lemma** *size-rbt-app[simp]*:  $\text{size } (\text{rbt-app } t1 \ t2) = \text{size } t1 + \text{size } t2$   
 ⟨proof⟩

**lemma** *size-rbt-joinL[simp]*:  $\text{size } (\text{rbt-joinL } t1 \ a \ b \ t2) = \text{Suc } (\text{size } t1 + \text{size } t2)$   
 ⟨proof⟩

**lemma** *size-rbt-joinR[simp]*:  $\text{size } (\text{rbt-joinR } t1 \ a \ b \ t2) = \text{Suc } (\text{size } t1 + \text{size } t2)$   
 ⟨proof⟩

**lemma** *size-rbt-join[simp]*:  $\text{size } (\text{rbt-join } t1 \ a \ b \ t2) = \text{Suc } (\text{size } t1 + \text{size } t2)$   
 ⟨proof⟩

**definition** *inv-12*  $t \longleftrightarrow \text{inv1 } t \wedge \text{inv2 } t$

**lemma** *rbt-Node*:  $\text{inv-12 } (\text{RBT-Impl.Branch } c \ l \ a \ b \ r) \Longrightarrow \text{inv-12 } l \wedge \text{inv-12 } r$   
 ⟨proof⟩

**lemma** *paint2*:  $\text{paint } c2 \ (\text{paint } c1 \ t) = \text{paint } c2 \ t$   
 ⟨proof⟩

**lemma** *inv1-rbt-baliL*:  $\text{inv1l } l \Longrightarrow \text{inv1 } r \Longrightarrow \text{inv1 } (\text{rbt-baliL } l \ a \ b \ r)$   
 ⟨proof⟩

**lemma** *inv1-rbt-baliR*:  $\text{inv1 } l \Longrightarrow \text{inv1l } r \Longrightarrow \text{inv1 } (\text{rbt-baliR } l \ a \ b \ r)$   
 ⟨proof⟩

**lemma** *rbt-bheight-rbt-baliL*:  $\text{bheight } l = \text{bheight } r \Longrightarrow \text{bheight } (\text{rbt-baliL } l \ a \ b \ r) = \text{Suc } (\text{bheight } l)$   
 ⟨proof⟩

**lemma** *rbt-bheight-rbt-baliR*:  $\text{bheight } l = \text{bheight } r \Longrightarrow \text{bheight } (\text{rbt-baliR } l \ a \ b \ r) = \text{Suc } (\text{bheight } l)$   
 ⟨proof⟩

**lemma** *inv2-rbt-baliL*:  $\text{inv2 } l \Longrightarrow \text{inv2 } r \Longrightarrow \text{bheight } l = \text{bheight } r \Longrightarrow \text{inv2 } (\text{rbt-baliL } l \ a \ b \ r)$   
 ⟨proof⟩

**lemma** *inv2-rbt-baliR*:  $\text{inv2 } l \Longrightarrow \text{inv2 } r \Longrightarrow \text{bheight } l = \text{bheight } r \Longrightarrow \text{inv2 } (\text{rbt-baliR } l \ a \ b \ r)$   
 ⟨proof⟩

**lemma** *inv-rbt-baliR*:  $\text{inv2 } l \Longrightarrow \text{inv2 } r \Longrightarrow \text{inv1 } l \Longrightarrow \text{inv1l } r \Longrightarrow \text{bheight } l = \text{bheight } r \Longrightarrow$   
 $\text{inv1 } (\text{rbt-baliR } l \ a \ b \ r) \wedge \text{inv2 } (\text{rbt-baliR } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baliR } l \ a \ b \ r) = \text{Suc } (\text{bheight } l)$   
 ⟨proof⟩

**lemma** *inv-rbt-baliL*:  $\text{inv2 } l \Longrightarrow \text{inv2 } r \Longrightarrow \text{inv1l } l \Longrightarrow \text{inv1 } r \Longrightarrow \text{bheight } l =$

$bheight\ r \implies$   
 $inv1\ (rbt-baliL\ l\ a\ b\ r) \wedge inv2\ (rbt-baliL\ l\ a\ b\ r) \wedge bheight\ (rbt-baliL\ l\ a\ b\ r) =$   
 $Suc\ (bheight\ l)$   
 ⟨proof⟩

**lemma**  $inv2\text{-}rbt\text{-}baldL\text{-}inv1$ :  $inv2\ l \implies inv2\ r \implies bheight\ l + 1 = bheight\ r \implies$   
 $inv1\ r \implies$   
 $inv2\ (rbt-baldL\ l\ a\ b\ r) \wedge bheight\ (rbt-baldL\ l\ a\ b\ r) = bheight\ r$   
 ⟨proof⟩

**lemma**  $inv2\text{-}rbt\text{-}baldL\text{-}B$ :  $inv2\ l \implies inv2\ r \implies bheight\ l + 1 = bheight\ r \implies$   
 $color\text{-}of\ r = RBT\text{-}Impl.B \implies$   
 $inv2\ (rbt-baldL\ l\ a\ b\ r) \wedge bheight\ (rbt-baldL\ l\ a\ b\ r) = bheight\ r$   
 ⟨proof⟩

**lemma**  $inv1\text{-}rbt\text{-}baldL$ :  $inv1\ l \implies inv1\ r \implies color\text{-}of\ r = RBT\text{-}Impl.B \implies inv1$   
 $(rbt-baldL\ l\ a\ b\ r)$   
 ⟨proof⟩

**lemma**  $inv1I$ :  $inv1\ t \implies inv1\ t$   
 ⟨proof⟩

**lemma**  $neg\text{-}Black[simp]$ :  $(c \neq RBT\text{-}Impl.B) = (c = RBT\text{-}Impl.R)$   
 ⟨proof⟩

**lemma**  $inv1l\text{-}rbt\text{-}baldL$ :  $inv1\ l \implies inv1\ r \implies inv1\ (rbt-baldL\ l\ a\ b\ r)$   
 ⟨proof⟩

**lemma**  $inv2\text{-}rbt\text{-}baldR\text{-}inv1$ :  $inv2\ l \implies inv2\ r \implies bheight\ l = bheight\ r + 1 \implies$   
 $inv1\ l \implies$   
 $inv2\ (rbt-baldR\ l\ a\ b\ r) \wedge bheight\ (rbt-baldR\ l\ a\ b\ r) = bheight\ l$   
 ⟨proof⟩

**lemma**  $inv1\text{-}rbt\text{-}baldR$ :  $inv1\ l \implies inv1\ r \implies color\text{-}of\ l = RBT\text{-}Impl.B \implies inv1$   
 $(rbt-baldR\ l\ a\ b\ r)$   
 ⟨proof⟩

**lemma**  $inv1l\text{-}rbt\text{-}baldR$ :  $inv1\ l \implies inv1\ r \implies inv1\ (rbt-baldR\ l\ a\ b\ r)$   
 ⟨proof⟩

**lemma**  $inv2\text{-}rbt\text{-}app$ :  $inv2\ l \implies inv2\ r \implies bheight\ l = bheight\ r \implies$   
 $inv2\ (rbt-app\ l\ r) \wedge bheight\ (rbt-app\ l\ r) = bheight\ l$   
 ⟨proof⟩

**lemma**  $inv1\text{-}rbt\text{-}app$ :  $inv1\ l \implies inv1\ r \implies (color\text{-}of\ l = RBT\text{-}Impl.B \wedge$   
 $color\text{-}of\ r = RBT\text{-}Impl.B \implies inv1\ (rbt-app\ l\ r)) \wedge inv1\ (rbt-app\ l\ r)$   
 ⟨proof⟩

**lemma**  $inv\text{-}rbt\text{-}baldL$ :  $inv2\ l \implies inv2\ r \implies bheight\ l + 1 = bheight\ r \implies inv1\ l$

$\implies \text{inv1 } r \implies$   
 $\text{inv2 } (\text{rbt-baldL } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baldL } l \ a \ b \ r) = \text{bheight } r \wedge$   
 $\text{inv1l } (\text{rbt-baldL } l \ a \ b \ r) \wedge (\text{color-of } r = \text{RBT-Impl.B} \longrightarrow \text{inv1 } (\text{rbt-baldL } l \ a \ b \ r))$   
 ⟨proof⟩

**lemma** *inv-rbt-baldR*:  $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l = \text{bheight } r + 1 \implies \text{inv1 } l$   
 $\implies \text{inv1l } r \implies$   
 $\text{inv2 } (\text{rbt-baldR } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baldR } l \ a \ b \ r) = \text{bheight } l \wedge$   
 $\text{inv1l } (\text{rbt-baldR } l \ a \ b \ r) \wedge (\text{color-of } l = \text{RBT-Impl.B} \longrightarrow \text{inv1 } (\text{rbt-baldR } l \ a \ b \ r))$   
 ⟨proof⟩

**lemma** *inv-rbt-app*:  $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l = \text{bheight } r \implies \text{inv1 } l \implies$   
 $\text{inv1 } r \implies$   
 $\text{inv2 } (\text{rbt-app } l \ r) \wedge \text{bheight } (\text{rbt-app } l \ r) = \text{bheight } l \wedge$   
 $\text{inv1l } (\text{rbt-app } l \ r) \wedge (\text{color-of } l = \text{RBT-Impl.B} \wedge \text{color-of } r = \text{RBT-Impl.B} \longrightarrow$   
 $\text{inv1 } (\text{rbt-app } l \ r))$   
 ⟨proof⟩

**lemma** *inv1l-rbt-joinL*:  $\text{inv1 } l \implies \text{inv1 } r \implies \text{bheight } l \leq \text{bheight } r \implies$   
 $\text{inv1l } (\text{rbt-joinL } l \ a \ b \ r) \wedge$   
 $(\text{bheight } l \neq \text{bheight } r \wedge \text{color-of } r = \text{RBT-Impl.B} \longrightarrow \text{inv1 } (\text{rbt-joinL } l \ a \ b \ r))$   
 ⟨proof⟩

**lemma** *inv1l-rbt-joinR*:  $\text{inv1 } l \implies \text{inv2 } l \implies \text{inv1 } r \implies \text{inv2 } r \implies \text{bheight } l \geq$   
 $\text{bheight } r \implies$   
 $\text{inv1l } (\text{rbt-joinR } l \ a \ b \ r) \wedge$   
 $(\text{bheight } l \neq \text{bheight } r \wedge \text{color-of } l = \text{RBT-Impl.B} \longrightarrow \text{inv1 } (\text{rbt-joinR } l \ a \ b \ r))$   
 ⟨proof⟩

**lemma** *bheight-rbt-joinL*:  $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l \leq \text{bheight } r \implies$   
 $\text{bheight } (\text{rbt-joinL } l \ a \ b \ r) = \text{bheight } r$   
 ⟨proof⟩

**lemma** *inv2-rbt-joinL*:  $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l \leq \text{bheight } r \implies \text{inv2}$   
 $(\text{rbt-joinL } l \ a \ b \ r)$   
 ⟨proof⟩

**lemma** *bheight-rbt-joinR*:  $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l \geq \text{bheight } r \implies$   
 $\text{bheight } (\text{rbt-joinR } l \ a \ b \ r) = \text{bheight } l$   
 ⟨proof⟩

**lemma** *inv2-rbt-joinR*:  $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l \geq \text{bheight } r \implies \text{inv2}$   
 $(\text{rbt-joinR } l \ a \ b \ r)$   
 ⟨proof⟩

**lemma** *keys-paint[simp]*:  $\text{RBT-Impl.keys } (\text{paint } c \ t) = \text{RBT-Impl.keys } t$   
 ⟨proof⟩

**lemma** *keys-rbt-baliL*:  $RBT-Impl.keys (rbt-baliL l a b r) = RBT-Impl.keys l @ a$   
 $\# RBT-Impl.keys r$   
 ⟨proof⟩

**lemma** *keys-rbt-baliR*:  $RBT-Impl.keys (rbt-baliR l a b r) = RBT-Impl.keys l @ a$   
 $\# RBT-Impl.keys r$   
 ⟨proof⟩

**lemma** *keys-rbt-baldL*:  $RBT-Impl.keys (rbt-baldL l a b r) = RBT-Impl.keys l @ a$   
 $\# RBT-Impl.keys r$   
 ⟨proof⟩

**lemma** *keys-rbt-baldR*:  $RBT-Impl.keys (rbt-baldR l a b r) = RBT-Impl.keys l @ a$   
 $\# RBT-Impl.keys r$   
 ⟨proof⟩

**lemma** *keys-rbt-app*:  $RBT-Impl.keys (rbt-app l r) = RBT-Impl.keys l @ RBT-Impl.keys r$   
 ⟨proof⟩

**lemma** *keys-rbt-joinL*:  $bheight l \leq bheight r \implies$   
 $RBT-Impl.keys (rbt-joinL l a b r) = RBT-Impl.keys l @ a \# RBT-Impl.keys r$   
 ⟨proof⟩

**lemma** *keys-rbt-joinR*:  $RBT-Impl.keys (rbt-joinR l a b r) = RBT-Impl.keys l @ a$   
 $\# RBT-Impl.keys r$   
 ⟨proof⟩

**lemma** *rbt-set-rbt-baliL*:  $set (RBT-Impl.keys (rbt-baliL l a b r)) =$   
 $set (RBT-Impl.keys l) \cup \{a\} \cup set (RBT-Impl.keys r)$   
 ⟨proof⟩

**lemma** *set-rbt-joinL*:  $set (RBT-Impl.keys (rbt-joinL l a b r)) =$   
 $set (RBT-Impl.keys l) \cup \{a\} \cup set (RBT-Impl.keys r)$   
 ⟨proof⟩

**lemma** *rbt-set-rbt-baliR*:  $set (RBT-Impl.keys (rbt-baliR l a b r)) =$   
 $set (RBT-Impl.keys l) \cup \{a\} \cup set (RBT-Impl.keys r)$   
 ⟨proof⟩

**lemma** *set-rbt-joinR*:  $set (RBT-Impl.keys (rbt-joinR l a b r)) =$   
 $set (RBT-Impl.keys l) \cup \{a\} \cup set (RBT-Impl.keys r)$   
 ⟨proof⟩

**lemma** *set-keys-paint*:  $set (RBT-Impl.keys (paint c t)) = set (RBT-Impl.keys t)$   
 ⟨proof⟩

**lemma** *set-rbt-join*:  $set (RBT-Impl.keys (rbt-join l a b r)) =$

*set* (*RBT-Impl.keys* *l*)  $\cup$  {*a*}  $\cup$  *set* (*RBT-Impl.keys* *r*)  
 ⟨*proof*⟩

**lemma** *inv-rbt-join*: *inv-12* *l*  $\implies$  *inv-12* *r*  $\implies$  *inv-12* (*rbt-join* *l a b r*)  
 ⟨*proof*⟩

**fun** *rbt-recolor* :: ('*a*, '*b*) *rbt*  $\Rightarrow$  ('*a*, '*b*) *rbt* **where**  
*rbt-recolor* (*Branch* *RBT-Impl.R* *t1 k v t2*) =  
 (if *color-of* *t1* = *RBT-Impl.B*  $\wedge$  *color-of* *t2* = *RBT-Impl.B* then *Branch*  
*RBT-Impl.B* *t1 k v t2*  
 else *Branch* *RBT-Impl.R* *t1 k v t2*)  
 | *rbt-recolor* *t* = *t*

**lemma** *rbt-recolor*: *inv-12* *t*  $\implies$  *inv-12* (*rbt-recolor* *t*)  
 ⟨*proof*⟩

**fun** *rbt-split-min* :: ('*a*, '*b*) *rbt*  $\Rightarrow$  '*a*  $\times$  '*b*  $\times$  ('*a*, '*b*) *rbt* **where**  
*rbt-split-min* *RBT-Impl.Empty* = *undefined*  
 | *rbt-split-min* (*RBT-Impl.Branch* - *l a b r*) =  
 (if *is-rbt-empty* *l* then (*a,b,r*) else let (*a',b',l'*) = *rbt-split-min* *l* in (*a',b',rbt-join*  
*l' a b r*))

**lemma** *rbt-split-min-set*:  
*rbt-split-min* *t* = (*a,b,t'*)  $\implies$  *t*  $\neq$  *RBT-Impl.Empty*  $\implies$   
*a*  $\in$  *set* (*RBT-Impl.keys* *t*)  $\wedge$  *set* (*RBT-Impl.keys* *t*) = {*a*}  $\cup$  *set* (*RBT-Impl.keys*  
*t'*)  
 ⟨*proof*⟩

**lemma** *rbt-split-min-inv*: *rbt-split-min* *t* = (*a,b,t'*)  $\implies$  *inv-12* *t*  $\implies$  *t*  $\neq$  *RBT-Impl.Empty*  
 $\implies$  *inv-12* *t'*  
 ⟨*proof*⟩

**definition** *rbt-join2* :: ('*a*, '*b*) *rbt*  $\Rightarrow$  ('*a*, '*b*) *rbt*  $\Rightarrow$  ('*a*, '*b*) *rbt* **where**  
*rbt-join2* *l r* = (if *is-rbt-empty* *r* then *l* else let (*a,b,r'*) = *rbt-split-min* *r* in *rbt-join*  
*l a b r'*)

**lemma** *set-rbt-join2[simp]*: *set* (*RBT-Impl.keys* (*rbt-join2* *l r*)) =  
*set* (*RBT-Impl.keys* *l*)  $\cup$  *set* (*RBT-Impl.keys* *r*)  
 ⟨*proof*⟩

**lemma** *inv-rbt-join2*: *inv-12* *l*  $\implies$  *inv-12* *r*  $\implies$  *inv-12* (*rbt-join2* *l r*)  
 ⟨*proof*⟩

**context** *ord* **begin**

**fun** *rbt-split* :: ('*a*, '*b*) *rbt*  $\Rightarrow$  '*a*  $\Rightarrow$  ('*a*, '*b*) *rbt*  $\times$  '*b* *option*  $\times$  ('*a*, '*b*) *rbt* **where**  
*rbt-split* *RBT-Impl.Empty* *k* = (*RBT-Impl.Empty*, *None*, *RBT-Impl.Empty*)  
 | *rbt-split* (*RBT-Impl.Branch* - *l a b r*) *x* =  
 (if *x* < *a* then (case *rbt-split* *l* *x* of (*l1*,  $\beta$ , *l2*)  $\Rightarrow$  (*l1*,  $\beta$ , *rbt-join* *l2 a b r*))



else if  $a < x$  then (case *rbt-split*  $r$   $x$  of ( $r1, \beta, r2$ )  $\Rightarrow$  (*rbt-join*  $l$   $a$   $b$   $r1, \beta, r2$ ))  
else ( $l, \text{Some } b, r$ )

**lemma** *rbt-split*: *rbt-split*  $t$   $x = (l, \beta, r) \Longrightarrow \text{inv-12 } t \Longrightarrow \text{inv-12 } l \wedge \text{inv-12 } r$   
(*proof*)

**lemma** *rbt-split-size*: ( $l2, \beta, r2$ ) = *rbt-split*  $t2$   $a \Longrightarrow \text{size } l2 + \text{size } r2 \leq \text{size } t2$   
(*proof*)

**function** *rbt-union-rec* :: ( $'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$ )  $\Rightarrow ('a, 'b)$  *rbt*  $\Rightarrow ('a, 'b)$  *rbt*  $\Rightarrow ('a,$   
 $'b)$  *rbt* **where**

*rbt-union-rec*  $f$   $t1$   $t2 = (\text{let } (f, t2, t1) =$   
if *flip-rbt*  $t2$   $t1$  then ( $\lambda k v v'. f k v' v, t1, t2$ ) else ( $f, t2, t1$ ) in  
if *small-rbt*  $t2$  then *RBT-Impl.fold* (*rbt-insert-with-key*  $f$ )  $t2$   $t1$   
else (case  $t1$  of *RBT-Impl.Empty*  $\Rightarrow t2$   
| *RBT-Impl.Branch* -  $l1$   $a$   $b$   $r1 \Rightarrow$   
case *rbt-split*  $t2$   $a$  of ( $l2, \beta, r2$ )  $\Rightarrow$   
*rbt-join* (*rbt-union-rec*  $f$   $l1$   $l2$ )  $a$  (case  $\beta$  of *None*  $\Rightarrow b$  | *Some*  $b' \Rightarrow f a b$   
 $b'$ ) (*rbt-union-rec*  $f$   $r1$   $r2$ )))  
(*proof*)

**termination**  
(*proof*)

**declare** *rbt-union-rec.simps*[*simp del*]

**function** *rbt-union-swap-rec* :: ( $'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$ )  $\Rightarrow \text{bool} \Rightarrow ('a, 'b)$  *rbt*  $\Rightarrow ('a,$   
 $'b)$  *rbt*  $\Rightarrow ('a, 'b)$  *rbt* **where**

*rbt-union-swap-rec*  $f$   $\gamma$   $t1$   $t2 = (\text{let } (\gamma, t2, t1) =$   
if *flip-rbt*  $t2$   $t1$  then ( $\neg \gamma, t1, t2$ ) else ( $\gamma, t2, t1$ );  
 $f' = (\text{if } \gamma \text{ then } (\lambda k v v'. f k v' v) \text{ else } f)$  in  
if *small-rbt*  $t2$  then *RBT-Impl.fold* (*rbt-insert-with-key*  $f'$ )  $t2$   $t1$   
else (case  $t1$  of *RBT-Impl.Empty*  $\Rightarrow t2$   
| *RBT-Impl.Branch* -  $l1$   $a$   $b$   $r1 \Rightarrow$   
case *rbt-split*  $t2$   $a$  of ( $l2, \beta, r2$ )  $\Rightarrow$   
*rbt-join* (*rbt-union-swap-rec*  $f$   $\gamma$   $l1$   $l2$ )  $a$  (case  $\beta$  of *None*  $\Rightarrow b$  | *Some*  $b' \Rightarrow$   
 $f' a b b'$ ) (*rbt-union-swap-rec*  $f$   $\gamma$   $r1$   $r2$ )))  
(*proof*)

**termination**  
(*proof*)

**declare** *rbt-union-swap-rec.simps*[*simp del*]

**lemma** *rbt-union-swap-rec*: *rbt-union-swap-rec*  $f$   $\gamma$   $t1$   $t2 =$   
*rbt-union-rec* (*if*  $\gamma$  then ( $\lambda k v v'. f k v' v$ ) else  $f$ )  $t1$   $t2$   
(*proof*)

**lemma** *rbt-fold-rbt-insert*:

**assumes** *inv-12*  $t2$

**shows** *inv-12* (*RBT-Impl.fold* (*rbt-insert-with-key*  $f$ )  $t1$   $t2$ )

⟨proof⟩

**lemma** *rbt-union-rec*:  $inv-12\ t1 \implies inv-12\ t2 \implies inv-12\ (rbt-union-rec\ f\ t1\ t2)$   
 ⟨proof⟩

**definition** *map-filter-inter*  $f\ t1\ t2 = List.map-filter\ (\lambda(k, v).$   
   *case* *rbt-lookup*  $t1\ k$  *of* *None*  $\implies None$   
   | *Some*  $v' \implies Some\ (k, f\ k\ v'\ v)$  *(RBT-Impl.entries*  $t2)$

**function** *rbt-inter-rec*  $:: ('a \implies 'b \implies 'b \implies 'b) \implies ('a, 'b)\ rbt \implies ('a, 'b)\ rbt \implies ('a,$   
 $'b)\ rbt$  **where**  
   *rbt-inter-rec*  $f\ t1\ t2 = (let\ (f, t2, t1) =$   
     *if* *flip-rbt*  $t2\ t1$  *then*  $(\lambda k\ v\ v'. f\ k\ v'\ v, t1, t2)$  *else*  $(f, t2, t1)$  *in*  
     *if* *small-rbt*  $t2$  *then* *rbtreeify*  $(map-filter-inter\ f\ t1\ t2)$   
     *else* *case*  $t1$  *of* *RBT-Impl.Empty*  $\implies RBT-Impl.Empty$   
     | *RBT-Impl.Branch* -  $l1\ a\ b\ r1 \implies$   
       *case* *rbt-split*  $t2\ a$  *of*  $(l2, \beta, r2) \implies let\ l' = rbt-inter-rec\ f\ l1\ l2; r' = rbt-inter-rec$   
        $f\ r1\ r2$  *in*  
       *(case*  $\beta$  *of* *None*  $\implies rbt-join2\ l'\ r'$  | *Some*  $b' \implies rbt-join\ l'\ a\ (f\ a\ b\ b')\ r')$ )  
   ⟨proof⟩

**termination**  
 ⟨proof⟩

**declare** *rbt-inter-rec.simps*[*simp del*]

**function** *rbt-inter-swap-rec*  $:: ('a \implies 'b \implies 'b \implies 'b) \implies bool \implies ('a, 'b)\ rbt \implies ('a,$   
 $'b)\ rbt \implies ('a, 'b)\ rbt$  **where**  
   *rbt-inter-swap-rec*  $f\ \gamma\ t1\ t2 = (let\ (\gamma, t2, t1) =$   
     *if* *flip-rbt*  $t2\ t1$  *then*  $(\neg\gamma, t1, t2)$  *else*  $(\gamma, t2, t1);$   
      $f' = (if\ \gamma$  *then*  $(\lambda k\ v\ v'. f\ k\ v'\ v)$  *else*  $f)$  *in*  
     *if* *small-rbt*  $t2$  *then* *rbtreeify*  $(map-filter-inter\ f'\ t1\ t2)$   
     *else* *case*  $t1$  *of* *RBT-Impl.Empty*  $\implies RBT-Impl.Empty$   
     | *RBT-Impl.Branch* -  $l1\ a\ b\ r1 \implies$   
       *case* *rbt-split*  $t2\ a$  *of*  $(l2, \beta, r2) \implies let\ l' = rbt-inter-swap-rec\ f\ \gamma\ l1\ l2; r' =$   
        $rbt-inter-swap-rec\ f\ \gamma\ r1\ r2$  *in*  
       *(case*  $\beta$  *of* *None*  $\implies rbt-join2\ l'\ r'$  | *Some*  $b' \implies rbt-join\ l'\ a\ (f'\ a\ b\ b')\ r')$ )  
   ⟨proof⟩

**termination**  
 ⟨proof⟩

**declare** *rbt-inter-swap-rec.simps*[*simp del*]

**lemma** *rbt-inter-swap-rec*:  $rbt-inter-swap-rec\ f\ \gamma\ t1\ t2 =$   
 $rbt-inter-rec\ (if\ \gamma$  *then*  $(\lambda k\ v\ v'. f\ k\ v'\ v)$  *else*  $f)\ t1\ t2$   
 ⟨proof⟩

**lemma** *rbt-rbtreeify*[*simp*]:  $inv-12\ (rbtreeify\ kus)$   
 ⟨proof⟩

**lemma** *rbt-inter-rec*:  $inv-12\ t1 \implies inv-12\ t2 \implies inv-12\ (rbt-inter-rec\ f\ t1\ t2)$   
 ⟨*proof*⟩

**definition** *filter-minus*  $t1\ t2 = filter\ (\lambda(k, -).\ rbt-lookup\ t2\ k = None)\ (RBT-Impl.entries\ t1)$

**fun** *rbt-minus-rec* ::  $('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$  **where**  
 $rbt-minus-rec\ t1\ t2 = (if\ small-rbt\ t2\ then\ RBT-Impl.fold\ (\lambda k - t.\ rbt-delete\ k\ t)\ t2\ t1$   
 $\quad else\ if\ small-rbt\ t1\ then\ rbtreeify\ (filter-minus\ t1\ t2)$   
 $\quad else\ case\ t2\ of\ RBT-Impl.Empty \Rightarrow t1$   
 $\quad \quad | RBT-Impl.Branch - l2\ a\ b\ r2 \Rightarrow$   
 $\quad \quad \quad case\ rbt-split\ t1\ a\ of\ (l1, -, r1) \Rightarrow rbt-join2\ (rbt-minus-rec\ l1\ l2)\ (rbt-minus-rec\ r1\ r2))$

**declare** *rbt-minus-rec.simps*[*simp del*]

**end**

**context** *linorder* **begin**

**lemma** *rbt-sorted-entries-right-unique*:  
 $\llbracket (k, v) \in set\ (entries\ t); (k, v') \in set\ (entries\ t);$   
 $\quad rbt-sorted\ t \rrbracket \implies v = v'$   
 ⟨*proof*⟩

**lemma** *rbt-sorted-fold-rbt-insertwk*:  
 $rbt-sorted\ t \implies rbt-sorted\ (List.fold\ (\lambda(k, v).\ rbt-insert-with-key\ f\ k\ v)\ xs\ t)$   
 ⟨*proof*⟩

**lemma** *is-rbt-fold-rbt-insertwk*:  
**assumes** *is-rbt*  $t1$   
**shows** *is-rbt*  $(fold\ (rbt-insert-with-key\ f)\ t2\ t1)$   
 ⟨*proof*⟩

**lemma** *rbt-delete*:  $inv-12\ t \implies inv-12\ (rbt-delete\ x\ t)$   
 ⟨*proof*⟩

**lemma** *rbt-sorted-delete*:  $rbt-sorted\ t \implies rbt-sorted\ (rbt-delete\ x\ t)$   
 ⟨*proof*⟩

**lemma** *rbt-fold-rbt-delete*:  
**assumes** *inv-12*  $t2$   
**shows** *inv-12*  $(RBT-Impl.fold\ (\lambda k - t.\ rbt-delete\ k\ t)\ t1\ t2)$   
 ⟨*proof*⟩

**lemma** *rbt-minus-rec*:  $inv-12\ t1 \implies inv-12\ t2 \implies inv-12\ (rbt-minus-rec\ t1\ t2)$   
 ⟨*proof*⟩

**end**

**context** *linorder* **begin**

**lemma** *rbt-sorted-rbt-baliL*:  $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid\ll a \implies a \ll\mid r \implies \text{rbt-sorted } (\text{rbt-baliL } l \ a \ b \ r)$   
 ⟨*proof*⟩

**lemma** *rbt-lookup-rbt-baliL*:  $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid\ll a \implies a \ll\mid r \implies \text{rbt-lookup } (\text{rbt-baliL } l \ a \ b \ r) \ k =$   
 (if  $k < a$  then  $\text{rbt-lookup } l \ k$  else if  $k = a$  then  $\text{Some } b$  else  $\text{rbt-lookup } r \ k$ )  
 ⟨*proof*⟩

**lemma** *rbt-sorted-rbt-baliR*:  $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid\ll a \implies a \ll\mid r \implies \text{rbt-sorted } (\text{rbt-baliR } l \ a \ b \ r)$   
 ⟨*proof*⟩

**lemma** *rbt-lookup-rbt-baliR*:  $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid\ll a \implies a \ll\mid r \implies \text{rbt-lookup } (\text{rbt-baliR } l \ a \ b \ r) \ k =$   
 (if  $k < a$  then  $\text{rbt-lookup } l \ k$  else if  $k = a$  then  $\text{Some } b$  else  $\text{rbt-lookup } r \ k$ )  
 ⟨*proof*⟩

**lemma** *rbt-sorted-rbt-joinL*:  $\text{rbt-sorted } (\text{RBT-Impl.Branch } c \ l \ a \ b \ r) \implies \text{bheight } l \leq \text{bheight } r \implies \text{rbt-sorted } (\text{rbt-joinL } l \ a \ b \ r)$   
 ⟨*proof*⟩

**lemma** *rbt-lookup-rbt-joinL*:  $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid\ll a \implies a \ll\mid r \implies \text{rbt-lookup } (\text{rbt-joinL } l \ a \ b \ r) \ k =$   
 (if  $k < a$  then  $\text{rbt-lookup } l \ k$  else if  $k = a$  then  $\text{Some } b$  else  $\text{rbt-lookup } r \ k$ )  
 ⟨*proof*⟩

**lemma** *rbt-sorted-rbt-joinR*:  $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid\ll a \implies a \ll\mid r \implies \text{rbt-sorted } (\text{rbt-joinR } l \ a \ b \ r)$   
 ⟨*proof*⟩

**lemma** *rbt-lookup-rbt-joinR*:  $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid\ll a \implies a \ll\mid r \implies \text{rbt-lookup } (\text{rbt-joinR } l \ a \ b \ r) \ k =$   
 (if  $k < a$  then  $\text{rbt-lookup } l \ k$  else if  $k = a$  then  $\text{Some } b$  else  $\text{rbt-lookup } r \ k$ )  
 ⟨*proof*⟩

**lemma** *rbt-sorted-paint*:  $\text{rbt-sorted } (\text{paint } c \ t) = \text{rbt-sorted } t$   
 ⟨*proof*⟩

**lemma** *rbt-sorted-rbt-join*:  $\text{rbt-sorted } (\text{RBT-Impl.Branch } c \ l \ a \ b \ r) \implies \text{rbt-sorted } (\text{rbt-join } l \ a \ b \ r)$   
 ⟨*proof*⟩

**lemma** *rbt-lookup-rbt-join*:  $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \ll a \implies a \ll r \implies$   
 $\text{rbt-lookup } (\text{rbt-join } l \ a \ b \ r) \ k =$   
*(if*  $k < a$  *then*  $\text{rbt-lookup } l \ k$  *else if*  $k = a$  *then* *Some*  $b$  *else*  $\text{rbt-lookup } r \ k$ *)*  
*<proof>*

**lemma** *rbt-split-min-rbt-sorted*:  $\text{rbt-split-min } t = (a, b, t') \implies \text{rbt-sorted } t \implies t \neq$   
 $\text{RBT-Impl.Empty} \implies$   
 $\text{rbt-sorted } t' \wedge (\forall x \in \text{set } (\text{RBT-Impl.keys } t'). \ a < x)$   
*<proof>*

**lemma** *rbt-split-min-rbt-lookup*:  $\text{rbt-split-min } t = (a, b, t') \implies \text{rbt-sorted } t \implies t \neq$   
 $\text{RBT-Impl.Empty} \implies$   
 $\text{rbt-lookup } t \ k = (\text{if } k < a \text{ then None else if } k = a \text{ then Some } b \text{ else rbt-lookup } t'$   
 $k)$   
*<proof>*

**lemma** *rbt-sorted-rbt-join2*:  $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies$   
 $\forall x \in \text{set } (\text{RBT-Impl.keys } l). \ \forall y \in \text{set } (\text{RBT-Impl.keys } r). \ x < y \implies \text{rbt-sorted}$   
 $(\text{rbt-join2 } l \ r)$   
*<proof>*

**lemma** *rbt-lookup-rbt-join2*:  $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies$   
 $\forall x \in \text{set } (\text{RBT-Impl.keys } l). \ \forall y \in \text{set } (\text{RBT-Impl.keys } r). \ x < y \implies$   
 $\text{rbt-lookup } (\text{rbt-join2 } l \ r) \ k = (\text{case } \text{rbt-lookup } l \ k \ \text{of None} \Rightarrow \text{rbt-lookup } r \ k \mid \text{Some}$   
 $v \Rightarrow \text{Some } v)$   
*<proof>*

**lemma** *rbt-split-props*:  $\text{rbt-split } t \ x = (l, \beta, r) \implies \text{rbt-sorted } t \implies$   
 $\text{set } (\text{RBT-Impl.keys } l) = \{a \in \text{set } (\text{RBT-Impl.keys } t). \ a < x\} \wedge$   
 $\text{set } (\text{RBT-Impl.keys } r) = \{a \in \text{set } (\text{RBT-Impl.keys } t). \ x < a\} \wedge$   
 $\text{rbt-sorted } l \wedge \text{rbt-sorted } r$   
*<proof>*

**lemma** *rbt-split-lookup*:  $\text{rbt-split } t \ x = (l, \beta, r) \implies \text{rbt-sorted } t \implies$   
 $\text{rbt-lookup } t \ k = (\text{if } k < x \text{ then } \text{rbt-lookup } l \ k \ \text{else if } k = x \text{ then } \beta \ \text{else } \text{rbt-lookup}$   
 $r \ k)$   
*<proof>*

**lemma** *rbt-sorted-fold-insertwk*:  $\text{rbt-sorted } t \implies$   
 $\text{rbt-sorted } (\text{RBT-Impl.fold } (\text{rbt-insert-with-key } f) \ t' \ t)$   
*<proof>*

**lemma** *rbt-lookup-iff-keys*:  
 $\text{rbt-sorted } t \implies \text{set } (\text{RBT-Impl.keys } t) = \{k. \ \exists v. \ \text{rbt-lookup } t \ k = \text{Some } v\}$   
 $\text{rbt-sorted } t \implies \text{rbt-lookup } t \ k = \text{None} \iff k \notin \text{set } (\text{RBT-Impl.keys } t)$   
 $\text{rbt-sorted } t \implies (\exists v. \ \text{rbt-lookup } t \ k = \text{Some } v) \iff k \in \text{set } (\text{RBT-Impl.keys } t)$   
*<proof>*

**lemma** *rbt-lookup-fold-rbt-insertwk*:

**assumes**  $t1$ : *rbt-sorted*  $t1$  **and**  $t2$ : *rbt-sorted*  $t2$   
**shows** *rbt-lookup* (fold (*rbt-insert-with-key*  $f$ )  $t1$   $t2$ )  $k$  =  
 (case *rbt-lookup*  $t1$   $k$  of *None*  $\Rightarrow$  *rbt-lookup*  $t2$   $k$   
 | *Some*  $v$   $\Rightarrow$  case *rbt-lookup*  $t2$   $k$  of *None*  $\Rightarrow$  *Some*  $v$   
 | *Some*  $w$   $\Rightarrow$  *Some* ( $f$   $k$   $w$   $v$ ))

*<proof>*

**lemma** *rbt-lookup-union-rec*: *rbt-sorted*  $t1$   $\Longrightarrow$  *rbt-sorted*  $t2$   $\Longrightarrow$   
*rbt-sorted* (*rbt-union-rec*  $f$   $t1$   $t2$ )  $\wedge$  *rbt-lookup* (*rbt-union-rec*  $f$   $t1$   $t2$ )  $k$  =  
 (case *rbt-lookup*  $t1$   $k$  of *None*  $\Rightarrow$  *rbt-lookup*  $t2$   $k$   
 | *Some*  $v$   $\Rightarrow$  (case *rbt-lookup*  $t2$   $k$  of *None*  $\Rightarrow$  *Some*  $v$   
 | *Some*  $w$   $\Rightarrow$  *Some* ( $f$   $k$   $v$   $w$ )))

*<proof>*

**lemma** *rbtreeify-map-filter-inter*:

**fixes**  $f$  ::  $'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$

**assumes** *rbt-sorted*  $t2$

**shows** *rbt-sorted* (*rbtreeify* (*map-filter-inter*  $f$   $t1$   $t2$ ))  
*rbt-lookup* (*rbtreeify* (*map-filter-inter*  $f$   $t1$   $t2$ ))  $k$  =  
 (case *rbt-lookup*  $t1$   $k$  of *None*  $\Rightarrow$  *None*  
 | *Some*  $v$   $\Rightarrow$  (case *rbt-lookup*  $t2$   $k$  of *None*  $\Rightarrow$  *None* | *Some*  $w$   $\Rightarrow$  *Some* ( $f$   $k$   $v$   $w$ )))

*<proof>*

**lemma** *rbt-lookup-inter-rec*: *rbt-sorted*  $t1$   $\Longrightarrow$  *rbt-sorted*  $t2$   $\Longrightarrow$

*rbt-sorted* (*rbt-inter-rec*  $f$   $t1$   $t2$ )  $\wedge$  *rbt-lookup* (*rbt-inter-rec*  $f$   $t1$   $t2$ )  $k$  =  
 (case *rbt-lookup*  $t1$   $k$  of *None*  $\Rightarrow$  *None*  
 | *Some*  $v$   $\Rightarrow$  (case *rbt-lookup*  $t2$   $k$  of *None*  $\Rightarrow$  *None* | *Some*  $w$   $\Rightarrow$  *Some* ( $f$   $k$   $v$   $w$ )))

*<proof>*

**lemma** *rbt-lookup-delete*:

**assumes** *inv-12*  $t$  *rbt-sorted*  $t$

**shows** *rbt-lookup* (*rbt-delete*  $x$   $t$ )  $k$  = (*if*  $x = k$  *then* *None* *else* *rbt-lookup*  $t$   $k$ )

*<proof>*

**lemma** *fold-rbt-delete*:

**assumes** *inv-12*  $t1$  *rbt-sorted*  $t1$  *rbt-sorted*  $t2$

**shows** *inv-12* (*RBT-Impl.fold* ( $\lambda k - t. \text{rbt-delete } k \ t$ )  $t2$   $t1$ )  $\wedge$   
*rbt-sorted* (*RBT-Impl.fold* ( $\lambda k - t. \text{rbt-delete } k \ t$ )  $t2$   $t1$ )  $\wedge$   
*rbt-lookup* (*RBT-Impl.fold* ( $\lambda k - t. \text{rbt-delete } k \ t$ )  $t2$   $t1$ )  $k$  =  
 (case *rbt-lookup*  $t1$   $k$  of *None*  $\Rightarrow$  *None*  
 | *Some*  $v$   $\Rightarrow$  (case *rbt-lookup*  $t2$   $k$  of *None*  $\Rightarrow$  *Some*  $v$  | -  $\Rightarrow$  *None*))

*<proof>*

**lemma** *rbtreeify-filter-minus*:

**assumes** *rbt-sorted*  $t1$

**shows** *rbt-sorted* (*rbtreeify* (*filter-minus*  $t1$   $t2$ ))  $\wedge$   
*rbt-lookup* (*rbtreeify* (*filter-minus*  $t1$   $t2$ ))  $k$  =  
 (case *rbt-lookup*  $t1$   $k$  of *None*  $\Rightarrow$  *None*)

| *Some v*  $\Rightarrow$  (case *rbt-lookup t2 k* of *None*  $\Rightarrow$  *Some v* | -  $\Rightarrow$  *None*)  
 ⟨*proof*⟩

**lemma** *rbt-lookup-minus-rec*: *inv-12 t1*  $\Longrightarrow$  *rbt-sorted t1*  $\Longrightarrow$  *rbt-sorted t2*  $\Longrightarrow$   
*rbt-sorted (rbt-minus-rec t1 t2)*  $\wedge$  *rbt-lookup (rbt-minus-rec t1 t2) k* =  
 (case *rbt-lookup t1 k* of *None*  $\Rightarrow$  *None*  
 | *Some v*  $\Rightarrow$  (case *rbt-lookup t2 k* of *None*  $\Rightarrow$  *Some v* | -  $\Rightarrow$  *None*)  
 ⟨*proof*⟩

**end**

**context** *ord* **begin**

**definition** *rbt-union-with-key* :: (*'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *'b*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'a*, *'b*) *rbt*  $\Rightarrow$  (*'a*, *'b*) *rbt*  
 $\Rightarrow$  (*'a*, *'b*) *rbt*

**where**

*rbt-union-with-key f t1 t2* = *paint B (rbt-union-swap-rec f False t1 t2)*

**definition** *rbt-union-with where*

*rbt-union-with f* = *rbt-union-with-key* ( $\lambda$ -. *f*)

**definition** *rbt-union where*

*rbt-union* = *rbt-union-with-key* (%- - *rv. rv*)

**definition** *rbt-inter-with-key* :: (*'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *'b*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'a*, *'b*) *rbt*  $\Rightarrow$  (*'a*, *'b*) *rbt*  
 $\Rightarrow$  (*'a*, *'b*) *rbt*

**where**

*rbt-inter-with-key f t1 t2* = *paint B (rbt-inter-swap-rec f False t1 t2)*

**definition** *rbt-inter-with where*

*rbt-inter-with f* = *rbt-inter-with-key* ( $\lambda$ -. *f*)

**definition** *rbt-inter where*

*rbt-inter* = *rbt-inter-with-key* ( $\lambda$ - - *rv. rv*)

**definition** *rbt-minus where*

*rbt-minus t1 t2* = *paint B (rbt-minus-rec t1 t2)*

**end**

**context** *linorder* **begin**

**lemma** *is-rbt-rbt-unionwk* [*simp*]:

[[ *is-rbt t1*; *is-rbt t2* ]]  $\Longrightarrow$  *is-rbt (rbt-union-with-key f t1 t2)*  
 ⟨*proof*⟩

**lemma** *rbt-lookup-rbt-unionwk*:

[[ *rbt-sorted t1*; *rbt-sorted t2* ]]  
 $\Longrightarrow$  *rbt-lookup (rbt-union-with-key f t1 t2) k* =

(*case* *rbt-lookup* *t1* *k* of *None*  $\Rightarrow$  *rbt-lookup* *t2* *k*  
 | *Some* *v*  $\Rightarrow$  *case* *rbt-lookup* *t2* *k* of *None*  $\Rightarrow$  *Some* *v*  
 | *Some* *w*  $\Rightarrow$  *Some* (*f* *k* *v* *w*)  
 ⟨*proof*⟩

**lemma** *rbt-unionw-is-rbt*:  $\llbracket \text{is-rbt } lt; \text{is-rbt } rt \rrbracket \Longrightarrow \text{is-rbt } (\text{rbt-union-with } f \text{ } lt \text{ } rt)$   
 ⟨*proof*⟩

**lemma** *rbt-union-is-rbt*:  $\llbracket \text{is-rbt } lt; \text{is-rbt } rt \rrbracket \Longrightarrow \text{is-rbt } (\text{rbt-union } lt \text{ } rt)$   
 ⟨*proof*⟩

**lemma** *rbt-lookup-rbt-union*:  
 $\llbracket \text{rbt-sorted } s; \text{rbt-sorted } t \rrbracket \Longrightarrow$   
 $\text{rbt-lookup } (\text{rbt-union } s \text{ } t) = \text{rbt-lookup } s \text{ } ++ \text{rbt-lookup } t$   
 ⟨*proof*⟩

**lemma** *rbt-interwk-is-rbt* [*simp*]:  
 $\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \Longrightarrow \text{is-rbt } (\text{rbt-inter-with-key } f \text{ } t1 \text{ } t2)$   
 ⟨*proof*⟩

**lemma** *rbt-interw-is-rbt*:  
 $\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \Longrightarrow \text{is-rbt } (\text{rbt-inter-with } f \text{ } t1 \text{ } t2)$   
 ⟨*proof*⟩

**lemma** *rbt-inter-is-rbt*:  
 $\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \Longrightarrow \text{is-rbt } (\text{rbt-inter } t1 \text{ } t2)$   
 ⟨*proof*⟩

**lemma** *rbt-lookup-rbt-interwk*:  
 $\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket$   
 $\Longrightarrow \text{rbt-lookup } (\text{rbt-inter-with-key } f \text{ } t1 \text{ } t2) \text{ } k =$   
 (*case* *rbt-lookup* *t1* *k* of *None*  $\Rightarrow$  *None*  
 | *Some* *v*  $\Rightarrow$  *case* *rbt-lookup* *t2* *k* of *None*  $\Rightarrow$  *None*  
 | *Some* *w*  $\Rightarrow$  *Some* (*f* *k* *v* *w*)  
 ⟨*proof*⟩

**lemma** *rbt-lookup-rbt-inter*:  
 $\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket$   
 $\Longrightarrow \text{rbt-lookup } (\text{rbt-inter } t1 \text{ } t2) = \text{rbt-lookup } t2 \text{ } |' \text{ dom } (\text{rbt-lookup } t1)$   
 ⟨*proof*⟩

**lemma** *rbt-minus-is-rbt*:  
 $\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \Longrightarrow \text{is-rbt } (\text{rbt-minus } t1 \text{ } t2)$   
 ⟨*proof*⟩

**lemma** *rbt-lookup-rbt-minus*:  
 $\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket$   
 $\Longrightarrow \text{rbt-lookup } (\text{rbt-minus } t1 \text{ } t2) = \text{rbt-lookup } t1 \text{ } |' (- \text{ dom } (\text{rbt-lookup } t2))$   
 ⟨*proof*⟩



end

## 129.11 Code generator setup

```

lemmas [code] =
  ord.rbt-less-prop
  ord.rbt-greater-prop
  ord.rbt-sorted.simps
  ord.rbt-lookup.simps
  ord.is-rbt-def
  ord.rbt-ins.simps
  ord.rbt-insert-with-key-def
  ord.rbt-insertw-def
  ord.rbt-insert-def
  ord.rbt-del-from-left.simps
  ord.rbt-del-from-right.simps
  ord.rbt-del.simps
  ord.rbt-delete-def
  ord.rbt-split.simps
  ord.rbt-union-swap-rec.simps
  ord.map-filter-inter-def
  ord.rbt-inter-swap-rec.simps
  ord.filter-minus-def
  ord.rbt-minus-rec.simps
  ord.rbt-union-with-key-def
  ord.rbt-union-with-def
  ord.rbt-union-def
  ord.rbt-inter-with-key-def
  ord.rbt-inter-with-def
  ord.rbt-inter-def
  ord.rbt-minus-def
  ord.rbt-map-entry.simps
  ord.rbt-bulkload-def

```

More efficient implementations for *entries* and *keys*

**definition** *gen-entries* ::

$$((\text{'a} \times \text{'b}) \times (\text{'a}, \text{'b}) \text{ rbt}) \text{ list} \Rightarrow (\text{'a}, \text{'b}) \text{ rbt} \Rightarrow (\text{'a} \times \text{'b}) \text{ list}$$

**where**

$$\text{gen-entries } kvs \ t = \text{entries } t \ @ \ \text{concat } (\text{map } (\lambda(kv, t). kv \ # \ \text{entries } t) \ kvs)$$

**lemma** *gen-entries-simps* [simp, code]:

$$\text{gen-entries } [] \ \text{Empty} = []$$

$$\text{gen-entries } ((kv, t) \ # \ kvs) \ \text{Empty} = kv \ # \ \text{gen-entries } kvs \ t$$

$$\text{gen-entries } kvs \ (\text{Branch } c \ l \ k \ v \ r) = \text{gen-entries } (((k, v), r) \ # \ kvs) \ l$$

<proof>

**lemma** *entries-code* [code]:

$$\text{entries} = \text{gen-entries } []$$

<proof>

**definition** *gen-keys* :: ('a × ('a, 'b) rbt) list ⇒ ('a, 'b) rbt ⇒ 'a list  
**where** *gen-keys kts t* = *RBT-Impl.keys t @ concat (List.map (λ(k, t). k # keys t) kts)*

**lemma** *gen-keys-simps* [*simp, code*]:  
*gen-keys [] Empty* = []  
*gen-keys ((k, t) # kts) Empty* = *k # gen-keys kts t*  
*gen-keys kts (Branch c l k v r)* = *gen-keys ((k, r) # kts) l*  
 ⟨*proof*⟩

**lemma** *keys-code* [*code*]:  
*keys* = *gen-keys []*  
 ⟨*proof*⟩

Restore original type constraints for constants

⟨*ML*⟩

**hide-const** (**open**) *MR MB R B Empty entries keys fold gen-keys gen-entries*

**end**

## 130 Abstract type of RBT trees

**theory** *RBT*  
**imports** *Main RBT-Impl*  
**begin**

### 130.1 Type definition

**typedef** (**overloaded**) ('a, 'b) rbt = {*t* :: ('a::linorder, 'b) *RBT-Impl.rbt.is-rbt t*}  
**morphisms** *impl-of RBT*  
 ⟨*proof*⟩

**lemma** *rbt-eq-iff*:  
*t1 = t2* ↔ *impl-of t1 = impl-of t2*  
 ⟨*proof*⟩

**lemma** *rbt-eqI*:  
*impl-of t1 = impl-of t2* ⇒ *t1 = t2*  
 ⟨*proof*⟩

**lemma** *is-rbt-impl-of* [*simp, intro*]:  
*is-rbt (impl-of t)*  
 ⟨*proof*⟩

**lemma** *RBT-impl-of* [*simp, code abstype*]:  
*RBT (impl-of t)* = *t*

$\langle proof \rangle$

## 130.2 Primitive operations

**setup-lifting** *type-definition-rbt*

**lift-definition** *lookup* :: ( $'a::linorder, 'b$ ) *rbt*  $\Rightarrow 'a \rightarrow 'b$  **is** *rbt-lookup*  $\langle proof \rangle$

**lift-definition** *empty* :: ( $'a::linorder, 'b$ ) *rbt* **is** *RBT-Impl.Empty*  
 $\langle proof \rangle$

**lift-definition** *insert* ::  $'a::linorder \Rightarrow 'b \Rightarrow ('a, 'b)$  *rbt*  $\Rightarrow ('a, 'b)$  *rbt* **is** *rbt-insert*  
 $\langle proof \rangle$

**lift-definition** *delete* :: ( $'a::linorder \Rightarrow ('a, 'b)$  *rbt*  $\Rightarrow ('a, 'b)$  *rbt* **is** *rbt-delete*  
 $\langle proof \rangle$

**lift-definition** *entries* :: ( $'a::linorder, 'b$ ) *rbt*  $\Rightarrow ('a \times 'b)$  *list* **is** *RBT-Impl.entries*  
 $\langle proof \rangle$

**lift-definition** *keys* :: ( $'a::linorder, 'b$ ) *rbt*  $\Rightarrow 'a$  *list* **is** *RBT-Impl.keys*  $\langle proof \rangle$

**lift-definition** *bulkload* :: ( $'a::linorder \times 'b$ ) *list*  $\Rightarrow ('a, 'b)$  *rbt* **is** *rbt-bulkload*  
 $\langle proof \rangle$

**lift-definition** *map-entry* ::  $'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a::linorder, 'b)$  *rbt*  $\Rightarrow ('a, 'b)$  *rbt* **is** *rbt-map-entry*  
 $\langle proof \rangle$

**lift-definition** *map* :: ( $'a \Rightarrow 'b \Rightarrow 'c$ )  $\Rightarrow ('a::linorder, 'b)$  *rbt*  $\Rightarrow ('a, 'c)$  *rbt* **is** *RBT-Impl.map*  
 $\langle proof \rangle$

**lift-definition** *fold* :: ( $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c$ )  $\Rightarrow ('a::linorder, 'b)$  *rbt*  $\Rightarrow 'c \Rightarrow 'c$  **is** *RBT-Impl.fold*  $\langle proof \rangle$

**lift-definition** *union* :: ( $'a::linorder, 'b$ ) *rbt*  $\Rightarrow ('a, 'b)$  *rbt*  $\Rightarrow ('a, 'b)$  *rbt* **is** *rbt-union*  
 $\langle proof \rangle$

**lift-definition** *foldi* :: ( $'c \Rightarrow bool$ )  $\Rightarrow ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a :: linorder, 'b)$  *rbt*  $\Rightarrow 'c \Rightarrow 'c$   
**is** *RBT-Impl.foldi*  $\langle proof \rangle$

**lift-definition** *combine-with-key* :: ( $'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$ )  $\Rightarrow ('a::linorder, 'b)$  *rbt*  $\Rightarrow ('a, 'b)$  *rbt*  $\Rightarrow ('a, 'b)$  *rbt* **is** *RBT-Impl.rbt-union-with-key*  $\langle proof \rangle$

**lift-definition** *combine* :: ('b ⇒ 'b ⇒ 'b) ⇒ ('a::linorder, 'b) rbt ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt  
 is *RBT-Impl.rbt-union-with* ⟨proof⟩

### 130.3 Derived operations

**definition** *is-empty* :: ('a::linorder, 'b) rbt ⇒ bool **where**  
 [code]: *is-empty* t = (case *impl-of* t of *RBT-Impl.Empty* ⇒ True | - ⇒ False)

**definition** *filter* :: ('a ⇒ 'b ⇒ bool) ⇒ ('a::linorder, 'b) rbt ⇒ ('a, 'b) rbt **where**  
 [code]: *filter* P t = fold (λk v t. if P k v then insert k v t else t) t empty

### 130.4 Abstract lookup properties

**lemma** *lookup-RBT*:  
*is-rbt* t ⇒ lookup (RBT t) = rbt-lookup t  
 ⟨proof⟩

**lemma** *lookup-impl-of*:  
 rbt-lookup (*impl-of* t) = lookup t  
 ⟨proof⟩

**lemma** *entries-impl-of*:  
*RBT-Impl.entries* (*impl-of* t) = entries t  
 ⟨proof⟩

**lemma** *keys-impl-of*:  
*RBT-Impl.keys* (*impl-of* t) = keys t  
 ⟨proof⟩

**lemma** *lookup-keys*:  
 dom (lookup t) = set (keys t)  
 ⟨proof⟩

**lemma** *lookup-empty* [simp]:  
 lookup empty = Map.empty  
 ⟨proof⟩

**lemma** *lookup-insert* [simp]:  
 lookup (insert k v t) = (lookup t)(k ↦ v)  
 ⟨proof⟩

**lemma** *lookup-delete* [simp]:  
 lookup (delete k t) = (lookup t)(k := None)  
 ⟨proof⟩

**lemma** *map-of-entries* [simp]:  
 map-of (entries t) = lookup t  
 ⟨proof⟩

**lemma** *entries-lookup*:

$entries\ t1 = entries\ t2 \iff lookup\ t1 = lookup\ t2$   
 ⟨proof⟩

**lemma** *lookup-bulkload* [simp]:

$lookup\ (bulkload\ xs) = map-of\ xs$   
 ⟨proof⟩

**lemma** *lookup-map-entry* [simp]:

$lookup\ (map-entry\ k\ f\ t) = (lookup\ t)(k := map-option\ f\ (lookup\ t\ k))$   
 ⟨proof⟩

**lemma** *lookup-map* [simp]:

$lookup\ (map\ f\ t)\ k = map-option\ (f\ k)\ (lookup\ t\ k)$   
 ⟨proof⟩

**lemma** *lookup-combine-with-key* [simp]:

$lookup\ (combine-with-key\ f\ t1\ t2)\ k = combine-options\ (f\ k)\ (lookup\ t1\ k)\ (lookup\ t2\ k)$   
 ⟨proof⟩

**lemma** *combine-altdef*:  $combine\ f\ t1\ t2 = combine-with-key\ (\lambda-. f)\ t1\ t2$

⟨proof⟩

**lemma** *lookup-combine* [simp]:

$lookup\ (combine\ f\ t1\ t2)\ k = combine-options\ f\ (lookup\ t1\ k)\ (lookup\ t2\ k)$   
 ⟨proof⟩

**lemma** *fold-fold*:

$fold\ f\ t = List.fold\ (case-prod\ f)\ (entries\ t)$   
 ⟨proof⟩

**lemma** *impl-of-empty*:

$impl-of\ empty = RBT-Impl.Empty$   
 ⟨proof⟩

**lemma** *is-empty-empty* [simp]:

$is-empty\ t \iff t = empty$   
 ⟨proof⟩

**lemma** *RBT-lookup-empty* [simp]:

$rbt-lookup\ t = Map.empty \iff t = RBT-Impl.Empty$   
 ⟨proof⟩

**lemma** *lookup-empty-empty* [simp]:

$lookup\ t = Map.empty \iff t = empty$   
 ⟨proof⟩

**lemma** *sorted-keys* [*iff*]:  
*sorted* (*keys* *t*)  
 ⟨*proof*⟩

**lemma** *distinct-keys* [*iff*]:  
*distinct* (*keys* *t*)  
 ⟨*proof*⟩

**lemma** *finite-dom-lookup* [*simp, intro!*]: *finite* (*dom* (*lookup* *t*))  
 ⟨*proof*⟩

**lemma** *lookup-union*: *lookup* (*union* *s* *t*) = *lookup* *s* ++ *lookup* *t*  
 ⟨*proof*⟩

**lemma** *lookup-in-tree*: (*lookup* *t* *k* = *Some* *v*) = ((*k*, *v*) ∈ *set* (*entries* *t*))  
 ⟨*proof*⟩

**lemma** *keys-entries*: (*k* ∈ *set* (*keys* *t*)) = (∃ *v*. (*k*, *v*) ∈ *set* (*entries* *t*))  
 ⟨*proof*⟩

**lemma** *fold-def-alt*:  
*fold* *f* *t* = *List.fold* (*case-prod* *f*) (*entries* *t*)  
 ⟨*proof*⟩

**lemma** *distinct-entries*: *distinct* (*List.map* *fst* (*entries* *t*))  
 ⟨*proof*⟩

**lemma** *sorted-entries*: *sorted* (*List.map* *fst* (*entries* *t*))  
 ⟨*proof*⟩

**lemma** *non-empty-keys*: *t* ≠ *empty* ⇒ *keys* *t* ≠ []  
 ⟨*proof*⟩

**lemma** *keys-def-alt*:  
*keys* *t* = *List.map* *fst* (*entries* *t*)  
 ⟨*proof*⟩

**context**  
**begin**

**private lemma** *lookup-filter-aux*:  
**assumes** *distinct* (*List.map* *fst* *xs*)  
**shows** *lookup* (*List.fold* (λ(*k*, *v*) *t*. *if* *P* *k* *v* *then insert* *k* *v* *t* *else t*) *xs* *t*) *k* =  
 (*case map-of* *xs* *k* of

- None* ⇒ *lookup* *t* *k*
- | *Some* *v* ⇒ *if* *P* *k* *v* *then Some* *v* *else lookup* *t* *k*)

 ⟨*proof*⟩

**lemma** *lookup-filter*:

*lookup* (*filter P t*) *k* =  
 (case *lookup t k* of *None*  $\Rightarrow$  *None* | *Some v*  $\Rightarrow$  if *P k v* then *Some v* else *None*)  
 ⟨*proof*⟩

end

### 130.5 Quickcheck generators

**quickcheck-generator** *rbt predicate: is-rbt constructors: empty, insert*

### 130.6 Hide implementation details

**lifting-update** *rbt.lifting*

**lifting-forget** *rbt.lifting*

**hide-const** (**open**) *impl-of empty lookup keys entries bulkload delete map fold union insert map-entry foldi*

*is-empty filter*

**hide-fact** (**open**) *empty-def lookup-def keys-def entries-def bulkload-def delete-def map-def fold-def*

*union-def insert-def map-entry-def foldi-def is-empty-def filter-def*

end

## 131 Implementation of mappings with Red-Black Trees

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

This theory defines abstract red-black trees as an efficient representation of finite maps, backed by the implementation in *HOL-Library.RBT-Impl*.

### 131.1 Data type and invariant

The type  $(\prime k, \prime v)$  *RBT-Impl.rbt* denotes red-black trees with keys of type  $\prime k$  and values of type  $\prime v$ . To function properly, the key type must belong to the *linorder* class.

A value *t* of this type is a valid red-black tree if it satisfies the invariant *is-rbt t*. The abstract type  $(\prime k, \prime v)$  *RBT.rbt* always obeys this invariant, and for this reason you should only use this in our application. Going back to  $(\prime k, \prime v)$  *RBT-Impl.rbt* may be necessary in proofs if not yet proven properties about the operations must be established.

The interpretation function *RBT.lookup* returns the partial map represented by a red-black tree:

*RBT.lookup*

This function should be used for reasoning about the semantics of the RBT operations. Furthermore, it implements the lookup functionality for the data structure: It is executable and the lookup is performed in  $O(\log n)$ .

### 131.2 Operations

Currently, the following operations are supported:

*RBT.empty*

Returns the empty tree.  $O(1)$

*RBT.insert*

Updates the map at a given position.  $O(\log n)$

*RBT.delete*

Deletes a map entry at a given position.  $O(\log n)$

*RBT.entries*

Return a corresponding key-value list for a tree.

*RBT.bulkload*

Builds a tree from a key-value list.

*RBT.map-entry*

Maps a single entry in a tree.

*RBT.map*

Maps all values in a tree.  $O(n)$

*RBT.fold*

Folds over all entries in a tree.  $O(n)$

### 131.3 Invariant preservation

*is-rbt rbt.Empty*

(*Empty-is-rbt*)

*is-rbt ?t*  $\implies$  *is-rbt (rbt-insert ?k ?v ?t)*

(*rbt-insert-is-rbt*)

*is-rbt ?t*  $\implies$  *is-rbt (rbt-delete ?k ?t)*

(*delete-is-rbt*)

*is-rbt (rbt-bulkload ?xs)*

(*bulkload-is-rbt*)

*is-rbt (rbt-map-entry ?k ?f ?t) = is-rbt ?t*

(*map-entry-is-rbt*)

*is-rbt (RBT-Impl.map ?f ?t) = is-rbt ?t*

(*map-is-rbt*)

$\llbracket is-rbt ?lt; is-rbt ?rt \rrbracket \implies is-rbt (rbt-union ?lt ?rt)$

(*union-is-rbt*)



**131.4 Map Semantics**lookup-empty

*Mapping.lookup Mapping.empty ?k = None*

lookup-insert

*RBT.lookup (RBT.insert ?k ?v ?t) = (RBT.lookup ?t)(?k ↦ ?v)*

lookup-delete

*Mapping.lookup (Mapping.delete ?k ?m) ?k = None*

lookup-bulkload

*RBT.lookup (RBT.bulkload ?xs) = map-of ?xs*

lookup-map

*RBT.lookup (RBT.map ?f ?t) ?k = map-option (?f ?k) (RBT.lookup ?t ?k)*

**end**

**132 Implementation of sets using RBT trees**

```
theory RBT-Set
imports RBT Product-Lexorder
begin
```

**133 Definition of code datatype constructors**

**definition** *Set* :: (*'a::linorder, unit*) *rbt* ⇒ *'a set*  
**where** *Set t = {x . RBT.lookup t x = Some ()}*

**definition** *Coset* :: (*'a::linorder, unit*) *rbt* ⇒ *'a set*  
**where** [*simp*]: *Coset t = - Set t*

**134 Deletion of already existing code equations**

```
declare [[code drop: Set.empty Set.is-empty uminus-set-inst.uminus-set
Set.member Set.insert Set.remove UNIV Set.filter image
Set.subset-eq Ball Bex can-select Set.union minus-set-inst.minus-set Set.inter
card the-elem Pow sum prod Product-Type.product Id-on
Image trancl relcomp wf-on wf-code Min Inf-fin Max Sup-fin
(Inf :: 'a set set ⇒ 'a set) (Sup :: 'a set set ⇒ 'a set)
sorted-list-of-set List.map-project List.Bleast]]
```

## 135 Lemmas

### 135.1 Auxiliary lemmas

**lemma** *[simp]*:  $x \neq \text{Some } () \iff x = \text{None}$   
*<proof>*

**lemma** *Set-set-keys*:  $\text{Set } x = \text{dom } (\text{RBT.lookup } x)$   
*<proof>*

**lemma** *finite-Set* *[simp, intro!]*:  $\text{finite } (\text{Set } x)$   
*<proof>*

**lemma** *set-keys*:  $\text{Set } t = \text{set}(\text{RBT.keys } t)$   
*<proof>*

### 135.2 fold and filter

**lemma** *finite-fold-rbt-fold-eq*:  
**assumes** *comp-fun-commute* *f*  
**shows**  $\text{Finite-Set.fold } f A (\text{set } (\text{RBT.entries } t)) = \text{RBT.fold } (\text{curry } f) t A$   
*<proof>*

**definition** *fold-keys* ::  $(\text{'a} :: \text{linorder} \Rightarrow \text{'b} \Rightarrow \text{'b}) \Rightarrow (\text{'a}, -) \text{rbt} \Rightarrow \text{'b} \Rightarrow \text{'b}$   
**where** *[code-unfold]*:  $\text{fold-keys } f t A = \text{RBT.fold } (\lambda k - t. f k t) t A$

**lemma** *fold-keys-def-alt*:  
 $\text{fold-keys } f t s = \text{List.fold } f (\text{RBT.keys } t) s$   
*<proof>*

**lemma** *finite-fold-fold-keys*:  
**assumes** *comp-fun-commute* *f*  
**shows**  $\text{Finite-Set.fold } f A (\text{Set } t) = \text{fold-keys } f t A$   
*<proof>*

**definition** *rbt-filter* ::  $(\text{'a} :: \text{linorder} \Rightarrow \text{bool}) \Rightarrow (\text{'a}, \text{'b}) \text{rbt} \Rightarrow \text{'a} \text{ set}$  **where**  
 $\text{rbt-filter } P t = \text{RBT.fold } (\lambda k - A'. \text{if } P k \text{ then } \text{Set.insert } k A' \text{ else } A') t \{\}$

**lemma** *Set-filter-rbt-filter*:  
 $\text{Set.filter } P (\text{Set } t) = \text{rbt-filter } P t$   
*<proof>*

### 135.3 foldi and Ball

**lemma** *Ball-False*:  $\text{RBT-Impl.fold } (\lambda k v s. s \wedge P k) t \text{False} = \text{False}$   
*<proof>*

**lemma** *rbt-foldi-fold-conj*:  
 $\text{RBT-Impl.foldi } (\lambda s. s = \text{True}) (\lambda k v s. s \wedge P k) t \text{val} = \text{RBT-Impl.fold } (\lambda k v s. s \wedge P k) t \text{val}$

*<proof>*

**lemma** *foldi-fold-conj*:  $RBT.foldi (\lambda s. s = True) (\lambda k v s. s \wedge P k) t val = fold-keys$   
 $(\lambda k s. s \wedge P k) t val$   
*<proof>* **including** *rbt.lifting* *<proof>*

### 135.4 foldi and Bex

**lemma** *Bex-True*:  $RBT-Impl.fold (\lambda k v s. s \vee P k) t True = True$   
*<proof>*

**lemma** *rbt-foldi-fold-disj*:

$RBT-Impl.foldi (\lambda s. s = False) (\lambda k v s. s \vee P k) t val = RBT-Impl.fold (\lambda k v s.$   
 $s \vee P k) t val$   
*<proof>*

**lemma** *foldi-fold-disj*:  $RBT.foldi (\lambda s. s = False) (\lambda k v s. s \vee P k) t val = fold-keys$   
 $(\lambda k s. s \vee P k) t val$   
*<proof>* **including** *rbt.lifting* *<proof>*

### 135.5 folding over non empty trees and selecting the minimal and maximal element

#### 135.5.1 concrete

The concrete part is here because it’s probably not general enough to be moved to *RBT-Impl*

**definition** *rbt-fold1-keys* ::  $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a::linorder, 'b) RBT-Impl.rbt \Rightarrow 'a$   
**where**  $rbt-fold1-keys f t = List.fold f (tl(RBT-Impl.keys t)) (hd(RBT-Impl.keys t))$

**minimum definition** *rbt-min* ::  $('a::linorder, unit) RBT-Impl.rbt \Rightarrow 'a$   
**where**  $rbt-min t = rbt-fold1-keys min t$

**lemma** *key-le-right*:  $rbt-sorted (Branch c lt k v rt) \implies (\bigwedge x. x \in set (RBT-Impl.keys rt) \implies k \leq x)$   
*<proof>*

**lemma** *left-le-key*:  $rbt-sorted (Branch c lt k v rt) \implies (\bigwedge x. x \in set (RBT-Impl.keys lt) \implies x \leq k)$   
*<proof>*

**lemma** *fold-min-triv*:

**fixes**  $k :: - :: linorder$   
**shows**  $(\forall x \in set xs. k \leq x) \implies List.fold min xs k = k$   
*<proof>*

**lemma** *rbt-min-simps*:

*is-rbt* (*Branch c RBT-Impl.Empty k v rt*)  $\implies$  *rbt-min* (*Branch c RBT-Impl.Empty k v rt*) = *k*  
 ⟨*proof*⟩

**fun** *rbt-min-opt* **where**

*rbt-min-opt* (*Branch c RBT-Impl.Empty k v rt*) = *k* |  
*rbt-min-opt* (*Branch c (Branch lc llc lk lv lrt) k v rt*) = *rbt-min-opt* (*Branch lc llc lk lv lrt*)

**lemma** *rbt-min-opt-Branch*:

*t1*  $\neq$  *rbt.Empty*  $\implies$  *rbt-min-opt* (*Branch c t1 k () t2*) = *rbt-min-opt t1*  
 ⟨*proof*⟩

**lemma** *rbt-min-opt-induct* [*case-names empty left-empty left-non-empty*]:

**fixes** *t* :: ('*a* :: *linorder*, *unit*) *RBT-Impl.rbt*  
**assumes** *P rbt.Empty*  
**assumes**  $\bigwedge$  *color t1 a b t2*. *P t1*  $\implies$  *P t2*  $\implies$  *t1* = *rbt.Empty*  $\implies$  *P (Branch color t1 a b t2)*  
**assumes**  $\bigwedge$  *color t1 a b t2*. *P t1*  $\implies$  *P t2*  $\implies$  *t1*  $\neq$  *rbt.Empty*  $\implies$  *P (Branch color t1 a b t2)*  
**shows** *P t*  
 ⟨*proof*⟩

**lemma** *rbt-min-opt-in-set*:

**fixes** *t* :: ('*a* :: *linorder*, *unit*) *RBT-Impl.rbt*  
**assumes** *t*  $\neq$  *rbt.Empty*  
**shows** *rbt-min-opt t*  $\in$  *set (RBT-Impl.keys t)*  
 ⟨*proof*⟩

**lemma** *rbt-min-opt-is-min*:

**fixes** *t* :: ('*a* :: *linorder*, *unit*) *RBT-Impl.rbt*  
**assumes** *rbt-sorted t*  
**assumes** *t*  $\neq$  *rbt.Empty*  
**shows**  $\bigwedge$  *y*. *y*  $\in$  *set (RBT-Impl.keys t)*  $\implies$  *y*  $\geq$  *rbt-min-opt t*  
 ⟨*proof*⟩

**lemma** *rbt-min-eq-rbt-min-opt*:

**assumes** *t*  $\neq$  *RBT-Impl.Empty*  
**assumes** *is-rbt t*  
**shows** *rbt-min t* = *rbt-min-opt t*  
 ⟨*proof*⟩

**maximum definition** *rbt-max* :: ('*a*::*linorder*, *unit*) *RBT-Impl.rbt*  $\Rightarrow$  '*a*

**where** *rbt-max t* = *rbt-fold1-keys max t*

**lemma** *fold-max-triv*:

**fixes** *k* :: - :: *linorder*  
**shows** ( $\forall x \in \text{set } xs. x \leq k$ )  $\implies$  *List.fold max xs k* = *k*  
 ⟨*proof*⟩

**lemma** *fold-max-rev-eq*:

**fixes**  $xs :: ('a :: \text{linorder}) \text{ list}$

**assumes**  $xs \neq []$

**shows**  $\text{List.fold max (tl xs) (hd xs)} = \text{List.fold max (tl (rev xs)) (hd (rev xs))}$

*<proof>*

**lemma** *rbt-max-simps*:

**assumes**  $\text{is-rbt (Branch c lt k v RBT-Impl.Empty)}$

**shows**  $\text{rbt-max (Branch c lt k v RBT-Impl.Empty)} = k$

*<proof>*

**fun** *rbt-max-opt* **where**

$\text{rbt-max-opt (Branch c lt k v RBT-Impl.Empty)} = k \mid$

$\text{rbt-max-opt (Branch c lt k v (Branch rc rlc rk rv rrt))} = \text{rbt-max-opt (Branch rc rlc rk rv rrt)}$

**lemma** *rbt-max-opt-Branch*:

$t2 \neq \text{rbt.Empty} \implies \text{rbt-max-opt (Branch c t1 k () t2)} = \text{rbt-max-opt t2}$

*<proof>*

**lemma** *rbt-max-opt-induct* [*case-names empty right-empty right-non-empty*]:

**fixes**  $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$

**assumes**  $P \text{ rbt.Empty}$

**assumes**  $\bigwedge \text{color } t1 \ a \ b \ t2. P \ t1 \implies P \ t2 \implies t2 = \text{rbt.Empty} \implies P \ (\text{Branch color } t1 \ a \ b \ t2)$

**assumes**  $\bigwedge \text{color } t1 \ a \ b \ t2. P \ t1 \implies P \ t2 \implies t2 \neq \text{rbt.Empty} \implies P \ (\text{Branch color } t1 \ a \ b \ t2)$

**shows**  $P \ t$

*<proof>*

**lemma** *rbt-max-opt-in-set*:

**fixes**  $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$

**assumes**  $t \neq \text{rbt.Empty}$

**shows**  $\text{rbt-max-opt } t \in \text{set (RBT-Impl.keys } t)$

*<proof>*

**lemma** *rbt-max-opt-is-max*:

**fixes**  $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$

**assumes**  $\text{rbt-sorted } t$

**assumes**  $t \neq \text{rbt.Empty}$

**shows**  $\bigwedge y. y \in \text{set (RBT-Impl.keys } t) \implies y \leq \text{rbt-max-opt } t$

*<proof>*

**lemma** *rbt-max-eq-rbt-max-opt*:

**assumes**  $t \neq \text{RBT-Impl.Empty}$

**assumes**  $\text{is-rbt } t$

**shows**  $\text{rbt-max } t = \text{rbt-max-opt } t$

*<proof>*

**135.5.2 abstract****context includes** *rbt.lifting* **begin****lift-definition** *fold1-keys* :: ('a ⇒ 'a ⇒ 'a) ⇒ ('a::linorder, 'b) rbt ⇒ 'a  
is *rbt-fold1-keys* ⟨proof⟩**lemma** *fold1-keys-def-alt*:*fold1-keys* f t = *List.fold* f (tl (RBT.keys t)) (hd (RBT.keys t))  
⟨proof⟩**lemma** *finite-fold1-fold1-keys*:**assumes** *semilattice* f**assumes** ¬ *RBT.is-empty* t**shows** *semilattice-set.F* f (Set t) = *fold1-keys* f t

⟨proof⟩

**minimum lift-definition** *r-min* :: ('a :: linorder, unit) rbt ⇒ 'a **is** *rbt-min*

⟨proof⟩

**lift-definition** *r-min-opt* :: ('a :: linorder, unit) rbt ⇒ 'a **is** *rbt-min-opt* ⟨proof⟩**lemma** *r-min-alt-def*: *r-min* t = *fold1-keys* min t

⟨proof⟩

**lemma** *r-min-eq-r-min-opt*:**assumes** ¬ (*RBT.is-empty* t)**shows** *r-min* t = *r-min-opt* t

⟨proof⟩

**lemma** *fold-keys-min-top-eq*:**fixes** t :: ('a::{linorder,bounded-lattice-top}, unit) rbt**assumes** ¬ (*RBT.is-empty* t)**shows** *fold-keys* min t top = *fold1-keys* min t

⟨proof⟩

**maximum lift-definition** *r-max* :: ('a :: linorder, unit) rbt ⇒ 'a **is** *rbt-max*

⟨proof⟩

**lift-definition** *r-max-opt* :: ('a :: linorder, unit) rbt ⇒ 'a **is** *rbt-max-opt* ⟨proof⟩**lemma** *r-max-alt-def*: *r-max* t = *fold1-keys* max t

⟨proof⟩

**lemma** *r-max-eq-r-max-opt*:**assumes** ¬ (*RBT.is-empty* t)**shows** *r-max* t = *r-max-opt* t

⟨proof⟩

**lemma** *fold-keys-max-bot-eq*:**fixes** t :: ('a::{linorder,bounded-lattice-bot}, unit) rbt

**assumes**  $\neg (RBT.is-empty\ t)$   
**shows**  $fold-keys\ max\ t\ bot = fold1-keys\ max\ t$   
 $\langle proof \rangle$

**end**

## 136 Code equations

**code-datatype** *Set Coset*

**declare** *list.set[code]*

**lemma** *empty-Set [code]:*  
 $Set.empty = Set\ RBT.empty$   
 $\langle proof \rangle$

**lemma** *UNIV-Coset [code]:*  
 $UNIV = Coset\ RBT.empty$   
 $\langle proof \rangle$

**lemma** *is-empty-Set [code]:*  
 $Set.is-empty\ (Set\ t) = RBT.is-empty\ t$   
 $\langle proof \rangle$

**lemma** *compl-code [code]:*  
 $- Set\ xs = Coset\ xs$   
 $- Coset\ xs = Set\ xs$   
 $\langle proof \rangle$

**lemma** *member-code [code]:*  
 $x \in (Set\ t) = (RBT.lookup\ t\ x = Some\ ())$   
 $x \in (Coset\ t) = (RBT.lookup\ t\ x = None)$   
 $\langle proof \rangle$

**lemma** *insert-code [code]:*  
 $Set.insert\ x\ (Set\ t) = Set\ (RBT.insert\ x\ ()\ t)$   
 $Set.insert\ x\ (Coset\ t) = Coset\ (RBT.delete\ x\ t)$   
 $\langle proof \rangle$

**lemma** *remove-code [code]:*  
 $Set.remove\ x\ (Set\ t) = Set\ (RBT.delete\ x\ t)$   
 $Set.remove\ x\ (Coset\ t) = Coset\ (RBT.insert\ x\ ()\ t)$   
 $\langle proof \rangle$

**lemma** *union-Set [code]:*  
 $Set\ t \cup A = fold-keys\ Set.insert\ t\ A$   
 $\langle proof \rangle$

**lemma** *inter-Set [code]:*

$A \cap \text{Set } t = \text{rbt-filter } (\lambda k. k \in A) t$   
 ⟨proof⟩

**lemma** *minus-Set* [code]:  
 $A - \text{Set } t = \text{fold-keys } \text{Set.remove } t A$   
 ⟨proof⟩

**lemma** *union-Coset* [code]:  
 $\text{Coset } t \cup A = - \text{rbt-filter } (\lambda k. k \notin A) t$   
 ⟨proof⟩

**lemma** *union-Set-Set* [code]:  
 $\text{Set } t1 \cup \text{Set } t2 = \text{Set } (\text{RBT.union } t1 t2)$   
 ⟨proof⟩

**lemma** *inter-Coset* [code]:  
 $A \cap \text{Coset } t = \text{fold-keys } \text{Set.remove } t A$   
 ⟨proof⟩

**lemma** *inter-Coset-Coset* [code]:  
 $\text{Coset } t1 \cap \text{Coset } t2 = \text{Coset } (\text{RBT.union } t1 t2)$   
 ⟨proof⟩

**lemma** *minus-Coset* [code]:  
 $A - \text{Coset } t = \text{rbt-filter } (\lambda k. k \in A) t$   
 ⟨proof⟩

**lemma** *filter-Set* [code]:  
 $\text{Set.filter } P (\text{Set } t) = (\text{rbt-filter } P) t$   
 ⟨proof⟩

**lemma** *image-Set* [code]:  
 $\text{image } f (\text{Set } t) = \text{fold-keys } (\lambda k A. \text{Set.insert } (f k) A) t \{\}$   
 ⟨proof⟩

**lemma** *Ball-Set* [code]:  
 $\text{Ball } (\text{Set } t) P \longleftrightarrow \text{RBT.foldi } (\lambda s. s = \text{True}) (\lambda k v s. s \wedge P k) t \text{True}$   
 ⟨proof⟩

**lemma** *Bex-Set* [code]:  
 $\text{Bex } (\text{Set } t) P \longleftrightarrow \text{RBT.foldi } (\lambda s. s = \text{False}) (\lambda k v s. s \vee P k) t \text{False}$   
 ⟨proof⟩

**lemma** *subset-code* [code]:  
 $\text{Set } t \leq B \longleftrightarrow (\forall x \in \text{Set } t. x \in B)$   
 $A \leq \text{Coset } t \longleftrightarrow (\forall y \in \text{Set } t. y \notin A)$   
 ⟨proof⟩

**lemma** *subset-Coset-empty-Set-empty* [code]:



$$\text{Coset } t1 \leq \text{Set } t2 \iff (\text{case } (\text{RBT.impl-of } t1, \text{RBT.impl-of } t2) \text{ of}$$

$$(\text{rbt.Empty}, \text{rbt.Empty}) \Rightarrow \text{False} \mid$$

$$(-, -) \Rightarrow \text{Code.abort } (\text{STR "non-empty-trees"}) (\lambda-. \text{Coset } t1 \leq \text{Set } t2))$$
 <proof>

A frequent case – avoid intermediate sets

**lemma** [code-unfold]:

$$\text{Set } t1 \subseteq \text{Set } t2 \iff \text{RBT.foldi } (\lambda s. s = \text{True}) (\lambda k v s. s \wedge k \in \text{Set } t2) t1 \text{ True}$$
 <proof>

**lemma** card-Set [code]:

$$\text{card } (\text{Set } t) = \text{fold-keys } (\lambda-. n. n + 1) t 0$$
 <proof>

**lemma** sum-Set [code]:

$$\text{sum } f (\text{Set } xs) = \text{fold-keys } (\text{plus } \circ f) xs 0$$
 <proof>

**lemma** the-elem-set [code]:

**fixes**  $t :: ('a :: \text{linorder}, \text{unit}) \text{rbt}$   
**shows**  $\text{the-elem } (\text{Set } t) = (\text{case } \text{RBT.impl-of } t \text{ of}$   
 $(\text{Branch } \text{RBT-Impl.B } \text{RBT-Impl.Empty } x \ () \ \text{RBT-Impl.Empty}) \Rightarrow x$   
 $\mid - \Rightarrow \text{Code.abort } (\text{STR "not-a-singleton-tree"}) (\lambda-. \text{the-elem } (\text{Set } t))$ )  
 <proof>

**lemma** Pow-Set [code]:  $\text{Pow } (\text{Set } t) = \text{fold-keys } (\lambda x A. A \cup \text{Set.insert } x \ 'A) t \{\{\}\}$   
 <proof>

**lemma** product-Set [code]:

$$\text{Product-Type.product } (\text{Set } t1) (\text{Set } t2) =$$

$$\text{fold-keys } (\lambda x A. \text{fold-keys } (\lambda y. \text{Set.insert } (x, y)) t2 A) t1 \{\}$$
 <proof>

**lemma** Id-on-Set [code]:  $\text{Id-on } (\text{Set } t) = \text{fold-keys } (\lambda x. \text{Set.insert } (x, x)) t \{\}$   
 <proof>

**lemma** Image-Set [code]:

$$(\text{Set } t) \text{ “ } S = \text{fold-keys } (\lambda(x,y) A. \text{if } x \in S \text{ then } \text{Set.insert } y A \text{ else } A) t \{\}$$
 <proof>

**lemma** trancl-set-ntrancl [code]:

$$\text{trancl } (\text{Set } t) = \text{ntrancl } (\text{card } (\text{Set } t) - 1) (\text{Set } t)$$
 <proof>

**lemma** relcomp-Set [code]:

$$(\text{Set } t1) \text{ O } (\text{Set } t2) = \text{fold-keys}$$

$$(\lambda(x,y) A. \text{fold-keys } (\lambda(w,z) A'. \text{if } y = w \text{ then } \text{Set.insert } (x,z) A' \text{ else } A') t2 A)$$

$$t1 \{\}$$

⟨proof⟩

**lemma** *wf-set*:  $wf (Set t) = acyclic (Set t)$   
 ⟨proof⟩

**lemma** *wf-code-set*[code]:  $wf-code (Set t) = acyclic (Set t)$   
 ⟨proof⟩

**lemma** *Min-fin-set-fold* [code]:  
 $Min (Set t) =$   
 (if *RBT.is-empty* *t*  
 then *Code.abort* (*STR* "not-non-empty-tree") ( $\lambda\cdot. Min (Set t)$ )  
 else *r-min-opt* *t*)  
 ⟨proof⟩

**lemma** *Inf-fin-set-fold* [code]:  
 $Inf-fin (Set t) = Min (Set t)$   
 ⟨proof⟩

**lemma** *Inf-Set-fold*:  
**fixes**  $t :: ('a :: \{linorder, complete-lattice\}, unit) rbt$   
**shows**  $Inf (Set t) = (if RBT.is-empty t then top else r-min-opt t)$   
 ⟨proof⟩

**lemma** *Max-fin-set-fold* [code]:  
 $Max (Set t) =$   
 (if *RBT.is-empty* *t*  
 then *Code.abort* (*STR* "not-non-empty-tree") ( $\lambda\cdot. Max (Set t)$ )  
 else *r-max-opt* *t*)  
 ⟨proof⟩

**lemma** *Sup-fin-set-fold* [code]:  
 $Sup-fin (Set t) = Max (Set t)$   
 ⟨proof⟩

**lemma** *Sup-Set-fold*:  
**fixes**  $t :: ('a :: \{linorder, complete-lattice\}, unit) rbt$   
**shows**  $Sup (Set t) = (if RBT.is-empty t then bot else r-max-opt t)$   
 ⟨proof⟩

**context**  
**begin**

**declare** [[code drop: *Gcd-fin* *Lcm-fin* ⟨*Gcd* :: - ⇒ *nat*⟩ ⟨*Gcd* :: - ⇒ *int*⟩ ⟨*Lcm* :: -  
 ⇒ *nat*⟩ ⟨*Lcm* :: - ⇒ *int*⟩]]

**lemma** [code]:  
 $Gcd_{fin} (Set t) = fold-keys gcd t (0::'a::\{semiring-gcd, linorder\})$   
 ⟨proof⟩

```

lemma [code]:
  Gcd (Set t) = (Gcdfin (Set t) :: nat)
  ⟨proof⟩

lemma [code]:
  Gcd (Set t) = (Gcdfin (Set t) :: int)
  ⟨proof⟩

lemma [code]:
  Lcmfin (Set t) = fold-keys lcm t (1::'a::{semiring-gcd, linorder})
  ⟨proof⟩

lemma [code drop: Lcm :: - ⇒ nat, code]:
  Lcm (Set t) = (Lcmfin (Set t) :: nat)
  ⟨proof⟩

lemma [code drop: Lcm :: - ⇒ int, code]:
  Lcm (Set t) = (Lcmfin (Set t) :: int)
  ⟨proof⟩ definition Inf' :: 'a :: {linorder, complete-lattice} set ⇒ 'a
  where [code-abbrev]: Inf' = Inf

lemma Inf'-Set-fold [code]:
  Inf' (Set t) = (if RBT.is-empty t then top else r-min-opt t)
  ⟨proof⟩ definition Sup' :: 'a :: {linorder, complete-lattice} set ⇒ 'a
  where [code-abbrev]: Sup' = Sup

lemma Sup'-Set-fold [code]:
  Sup' (Set t) = (if RBT.is-empty t then bot else r-max-opt t)
  ⟨proof⟩

end

lemma sorted-list-set[code]: sorted-list-of-set (Set t) = RBT.keys t
  ⟨proof⟩

lemma Bleast-code [code]:
  Bleast (Set t) P =
    (case List.filter P (RBT.keys t) of
     x # xs ⇒ x
     | [] ⇒ abort-Bleast (Set t) P)
  ⟨proof⟩

hide-const (open) RBT-Set.Set RBT-Set.Coset

end

⟨ML⟩

```

```

theory Suc-Notation
imports Main
begin

```

Nested *Suc* terms of depth  $2 \leq n \leq 9$  are abbreviated with new notations *Suc*<sup>*n*</sup>:

```

abbreviation (input) Suc2 where Suc2 n  $\equiv$  Suc (Suc n)
abbreviation (input) Suc3 where Suc3 n  $\equiv$  Suc (Suc2 n)
abbreviation (input) Suc4 where Suc4 n  $\equiv$  Suc (Suc3 n)
abbreviation (input) Suc5 where Suc5 n  $\equiv$  Suc (Suc4 n)
abbreviation (input) Suc6 where Suc6 n  $\equiv$  Suc (Suc5 n)
abbreviation (input) Suc7 where Suc7 n  $\equiv$  Suc (Suc6 n)
abbreviation (input) Suc8 where Suc8 n  $\equiv$  Suc (Suc7 n)
abbreviation (input) Suc9 where Suc9 n  $\equiv$  Suc (Suc8 n)

```

```

notation Suc2 (Suc2)
notation Suc3 (Suc3)
notation Suc4 (Suc4)
notation Suc5 (Suc5)
notation Suc6 (Suc6)
notation Suc7 (Suc7)
notation Suc8 (Suc8)
notation Suc9 (Suc9)

```

Beyond 9, the syntax *Suc*<sup>*n*</sup> kicks in:

```

syntax
  -Suc-tower :: num-token  $\Rightarrow$  nat  $\Rightarrow$  nat (Suc-)

```

$\langle ML \rangle$

```

end

```

```

theory Predicate-Compile-Alternative-Defs
imports Main
begin

```

## 137 Common constants

```

declare HOL.if-bool-eq-disj[code-pred-inline]

declare bool-diff-def[code-pred-inline]
declare inf-bool-def[abs-def, code-pred-inline]
declare less-bool-def[abs-def, code-pred-inline]
declare le-bool-def[abs-def, code-pred-inline]

```

**lemma** *min-bool-eq* [*code-pred-inline*]:  $(\text{min} :: \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}) == (\wedge)$   
 <proof>

**lemma** [*code-pred-inline*]:  
 $((A :: \text{bool}) \neq (B :: \text{bool})) = ((A \wedge \neg B) \vee (B \wedge \neg A))$   
 <proof>

<ML>

## 138 Pairs

<ML>

## 139 Filters

<ML>

## 140 Bounded quantifiers

**declare** *Ball-def* [*code-pred-inline*]  
**declare** *Bex-def* [*code-pred-inline*]

## 141 Operations on Predicates

**lemma** *Diff* [*code-pred-inline*]:  
 $(A - B) = (\%x. A x \wedge \neg B x)$   
 <proof>

**lemma** *subset-eq* [*code-pred-inline*]:  
 $(P :: 'a \Rightarrow \text{bool}) < (Q :: 'a \Rightarrow \text{bool}) \equiv ((\exists x. Q x \wedge (\neg P x)) \wedge (\forall x. P x \longrightarrow Q x))$   
 <proof>

**lemma** *set-equality* [*code-pred-inline*]:  
 $A = B \longleftrightarrow (\forall x. A x \longrightarrow B x) \wedge (\forall x. B x \longrightarrow A x)$   
 <proof>

## 142 Setup for Numerals

<ML>

## 143 Arithmetic operations

### 143.1 Arithmetic on naturals and integers

**definition** *plus-eq-nat* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$

**where**

$plus\text{-}eq\text{-}nat\ x\ y\ z = (x + y = z)$

**definition**  $minus\text{-}eq\text{-}nat :: nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$

**where**

$minus\text{-}eq\text{-}nat\ x\ y\ z = (x - y = z)$

**definition**  $plus\text{-}eq\text{-}int :: int \Rightarrow int \Rightarrow int \Rightarrow bool$

**where**

$plus\text{-}eq\text{-}int\ x\ y\ z = (x + y = z)$

**definition**  $minus\text{-}eq\text{-}int :: int \Rightarrow int \Rightarrow int \Rightarrow bool$

**where**

$minus\text{-}eq\text{-}int\ x\ y\ z = (x - y = z)$

**definition**  $subtract$

**where**

$[code\text{-}unfold]:\ subtract\ x\ y = y - x$

$\langle ML \rangle$

## 143.2 Inductive definitions for ordering on naturals

**inductive**  $less\text{-}nat$

**where**

$less\text{-}nat\ 0\ (Suc\ y)$

$| less\text{-}nat\ x\ y \implies less\text{-}nat\ (Suc\ x)\ (Suc\ y)$

**lemma**  $less\text{-}nat[code\text{-}pred\text{-}inline]:$

$x < y = less\text{-}nat\ x\ y$

$\langle proof \rangle$

**inductive**  $less\text{-}eq\text{-}nat$

**where**

$less\text{-}eq\text{-}nat\ 0\ y$

$| less\text{-}eq\text{-}nat\ x\ y \implies less\text{-}eq\text{-}nat\ (Suc\ x)\ (Suc\ y)$

**lemma**  $[code\text{-}pred\text{-}inline]:$

$x \leq y = less\text{-}eq\text{-}nat\ x\ y$

$\langle proof \rangle$

## 144 Alternative list definitions

### 144.1 Alternative rules for $length$

**definition**  $size\text{-}list' :: 'a\ list \Rightarrow nat$

**where**  $size\text{-}list' = size$

**lemma**  $size\text{-}list'\text{-}simps:$

```

size-list' [] = 0
size-list' (x # xs) = Suc (size-list' xs)
⟨proof⟩

```

```

declare size-list'-simps[code-pred-def]
declare size-list'-def[symmetric, code-pred-inline]

```

### 144.2 Alternative rules for *list-all2*

```

lemma list-all2-NilI [code-pred-intro]: list-all2 P [] []
⟨proof⟩

```

```

lemma list-all2-ConsI [code-pred-intro]: list-all2 P xs ys ==> P x y ==> list-all2
P (x#xs) (y#ys)
⟨proof⟩

```

```

code-pred [skip-proof] list-all2
⟨proof⟩

```

### 144.3 Alternative rules for membership in lists

```

declare in-set-member[code-pred-inline]

```

```

lemma member-intros [code-pred-intro]:
  List.member (x#xs) x
  List.member xs x ==> List.member (y#xs) x
⟨proof⟩

```

```

code-pred List.member
⟨proof⟩

```

```

code-identifier constant member-i-i
  -> (SML) List.member-i-i
  and (OCaml) List.member-i-i
  and (Haskell) List.member-i-i
  and (Scala) List.member-i-i

```

```

code-identifier constant member-i-o
  -> (SML) List.member-i-o
  and (OCaml) List.member-i-o
  and (Haskell) List.member-i-o
  and (Scala) List.member-i-o

```

## 145 Setup for *String.literal*

```

⟨ML⟩

```

## 146 Simplification rules for optimisation

**lemma** *[code-pred-simp]:  $\neg \text{False} == \text{True}$*   
*<proof>*

**lemma** *[code-pred-simp]:  $\neg \text{True} == \text{False}$*   
*<proof>*

**lemma** *less-nat-k-0 [code-pred-simp]:  $\text{less-nat } k \ 0 == \text{False}$*   
*<proof>*

**end**

## 147 A Prototype of Quickcheck based on the Predicate Compiler

**theory** *Predicate-Compile-Quickcheck*  
**imports** *Predicate-Compile-Alternative-Defs*  
**begin**

*<ML>*

**end**

## 148 TFL: recursive function definitions

**theory** *Old-Recdef*  
**imports** *Main*  
**keywords**  
*recdef :: thy-defn and*  
*permissive congs hints*  
**begin**

### 148.1 Lemmas for TFL

**lemma** *tfl-wf-induct:  $\forall R. \text{wf } R \longrightarrow$*   
 $(\forall P. (\forall x. (\forall y. (y,x) \in R \longrightarrow P \ y) \longrightarrow P \ x) \longrightarrow (\forall x. P \ x))$   
*<proof>*

**lemma** *tfl-cut-def:  $\text{cut } f \ r \ x \equiv (\lambda y. \text{if } (y,x) \in r \text{ then } f \ y \text{ else undefined})$*   
*<proof>*

**lemma** *tfl-cut-apply:  $\forall f \ R. (x,a) \in R \longrightarrow (\text{cut } f \ R \ a)(x) = f(x)$*   
*<proof>*

**lemma** *tfl-wfrec:*  
 $\forall M \ R \ f. (f = \text{wfrec } R \ M) \longrightarrow \text{wf } R \longrightarrow (\forall x. f \ x = M (\text{cut } f \ R \ x) \ x)$   
*<proof>*



**lemma** *tfl-eq-True*:  $(x = \text{True}) \longrightarrow x$   
 ⟨*proof*⟩

**lemma** *tfl-rev-eq-mp*:  $(x = y) \longrightarrow y \longrightarrow x$   
 ⟨*proof*⟩

**lemma** *tfl-simp-thm*:  $(x \longrightarrow y) \longrightarrow (x = x') \longrightarrow (x' \longrightarrow y)$   
 ⟨*proof*⟩

**lemma** *tfl-P-imp-P-iff-True*:  $P \Longrightarrow P = \text{True}$   
 ⟨*proof*⟩

**lemma** *tfl-imp-trans*:  $(A \longrightarrow B) \Longrightarrow (B \longrightarrow C) \Longrightarrow (A \longrightarrow C)$   
 ⟨*proof*⟩

**lemma** *tfl-disj-assoc*:  $(a \vee b) \vee c \equiv a \vee (b \vee c)$   
 ⟨*proof*⟩

**lemma** *tfl-disjE*:  $P \vee Q \Longrightarrow P \longrightarrow R \Longrightarrow Q \longrightarrow R \Longrightarrow R$   
 ⟨*proof*⟩

**lemma** *tfl-exE*:  $\exists x. P x \Longrightarrow \forall x. P x \longrightarrow Q \Longrightarrow Q$   
 ⟨*proof*⟩

⟨*ML*⟩

## 148.2 Rule setup

**lemmas** [*recdef-simp*] =  
*inv-image-def*  
*measure-def*  
*lex-prod-def*  
*same-fst-def*  
*less-Suc-eq* [*THEN iffD2*]

**lemmas** [*recdef-cong*] =  
*if-cong* *let-cong* *image-cong* *INF-cong* *SUP-cong* *bex-cong* *ball-cong* *imp-cong*  
*map-cong* *filter-cong* *takeWhile-cong* *dropWhile-cong* *foldl-cong* *foldr-cong*

**lemmas** [*recdef-wf*] =  
*wf-trancl*  
*wf-less-than*  
*wf-lex-prod*  
*wf-inv-image*  
*wf-measure*  
*wf-measures*  
*wf-pred-nat*  
*wf-same-fst*

```
wf-empty
```

```
end
```

## 149 Program extraction from proofs involving datatypes and inductive predicates

```
theory Realizers
imports Main
begin
```

```
<ML>
```

```
end
```

## 150 Refute

```
theory Refute
imports Main
keywords
  refute :: diag and
  refute-params :: thy-decl
begin
```

```
<ML>
```

```
refute-params
[itself = 1,
 minsize = 1,
 maxsize = 8,
 maxvars = 10000,
 maxtime = 60,
 satsolver = auto,
 no-assms = false]
```

```
(* ----- *)
(* REFUTE *)
(* *)
(* We use a SAT solver to search for a (finite) model that refutes a given *)
(* HOL formula. *)
(* ----- *)

(* ----- *)
(* NOTE *)
(* *)
(* I strongly recommend that you install a stand-alone SAT solver if you *)
(* want to use 'refute'. For details see 'HOL/Tools/sat_solver.ML'. If you *)
```

```

(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME' *)
(* in 'etc/settings'. *)
(* ----- *)

(* ----- *)
(* USAGE *)
(* *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples. The supported *)
(* parameters are explained below. *)
(* ----- *)

(* ----- *)
(* CURRENT LIMITATIONS *)
(* *)
(* 'refute' currently accepts formulas of higher-order predicate logic (with *)
(* equality), including free/bound/schematic variables, lambda abstractions, *)
(* sets and set membership, "arbitrary", "The", "Eps", records and *)
(* inductively defined sets. Constants are unfolded automatically, and sort *)
(* axioms are added as well. Other, user-asserted axioms however are *)
(* ignored. Inductive datatypes and recursive functions are supported, but *)
(* may lead to spurious countermodels. *)
(* *)
(* The (space) complexity of the algorithm is non-elementary. *)
(* *)
(* Schematic type variables are not supported. *)
(* ----- *)

(* ----- *)
(* PARAMETERS *)
(* *)
(* The following global parameters are currently supported (and required, *)
(* except for "expect"): *)
(* *)
(* Name          Type      Description *)
(* *)
(* "minsize"     int        Only search for models with size at least *)
(*                'minsize'. *)
(* *)
(* "maxsize"     int        If >0, only search for models with size at most *)
(*                'maxsize'. *)
(* *)
(* "maxvars"     int        If >0, use at most 'maxvars' boolean variables *)
(*                when transforming the term into a propositional *)
(*                formula. *)
(* *)
(* "maxtime"     int        If >0, terminate after at most 'maxtime' seconds. *)
(*                This value is ignored under some ML compilers. *)
(* *)
(* "satsolver"   string     Name of the SAT solver to be used. *)
(* *)
(* "no_assms"    bool       If "true", assumptions in structured proofs are *)
(*                not considered. *)
(* *)
(* "expect"      string     Expected result ("genuine", "potential", "none", or *)
(*                "unknown"). *)

```

```

(*)
(*) The size of particular types can be specified in the form type=size
(*) (where 'type' is a string, and 'size' is an int).  Examples:
(*) "'a"=1
(*) "List.list"=2
(*) -----
(*) -----
(*) FILES
(*)
(*) HOL/Tools/prop_logic.ML      Propositional logic
(*) HOL/Tools/sat_solver.ML     SAT solvers
(*) HOL/Tools/refute.ML         Translation HOL -> propositional logic and
(*)                             Boolean assignment -> HOL model
(*) HOL/Refute.thy              This file: loads the ML files, basic setup,
(*)                             documentation
(*) HOL/SAT.thy                 Sets default parameters
(*) HOL/ex/Refute_Examples.thy  Examples
(*) -----

```

end

## 151 Misc lemmas on division, to be sorted out finally

```

theory Divides
imports Main
begin

```

```

class unique-euclidean-semiring-numeral = linordered-euclidean-semiring + dis-
crete-linordered-semidom +
  assumes div-less [no-atp]:  $0 \leq a \implies a < b \implies a \text{ div } b = 0$ 
  and mod-less [no-atp]:  $0 \leq a \implies a < b \implies a \text{ mod } b = a$ 
  and div-positive [no-atp]:  $0 < b \implies b \leq a \implies a \text{ div } b > 0$ 
  and mod-less-eq-dividend [no-atp]:  $0 \leq a \implies a \text{ mod } b \leq a$ 
  and pos-mod-bound [no-atp]:  $0 < b \implies a \text{ mod } b < b$ 
  and pos-mod-sign [no-atp]:  $0 < b \implies 0 \leq a \text{ mod } b$ 
  and mod-mult2-eq [no-atp]:  $0 \leq c \implies a \text{ mod } (b * c) = b * (a \text{ div } b \text{ mod } c) +$ 
 $a \text{ mod } b$ 
  and div-mult2-eq [no-atp]:  $0 \leq c \implies a \text{ div } (b * c) = a \text{ div } b \text{ div } c$ 

```

```

hide-fact (open) div-less mod-less div-positive mod-less-eq-dividend pos-mod-bound
pos-mod-sign mod-mult2-eq div-mult2-eq

```

```

context unique-euclidean-semiring-numeral
begin

```

```

context

```

**begin**

**qualified lemma** *discrete* [no-atp]:

$$a < b \longleftrightarrow a + 1 \leq b$$

*<proof>* **lemma** *divmod-digit-1* [no-atp]:

**assumes**  $0 \leq a < b$  **and**  $b \leq a \bmod (2 * b)$

**shows**  $2 * (a \operatorname{div} (2 * b)) + 1 = a \operatorname{div} b$  (**is** ?P)

**and**  $a \bmod (2 * b) - b = a \bmod b$  (**is** ?Q)

*<proof>* **lemma** *divmod-digit-0* [no-atp]:

**assumes**  $0 < b$  **and**  $a \bmod (2 * b) < b$

**shows**  $2 * (a \operatorname{div} (2 * b)) = a \operatorname{div} b$  (**is** ?P)

**and**  $a \bmod (2 * b) = a \bmod b$  (**is** ?Q)

*<proof>* **lemma** *mod-double-modulus* [no-atp]:

**assumes**  $m > 0$   $x \geq 0$

**shows**  $x \bmod (2 * m) = x \bmod m \vee x \bmod (2 * m) = x \bmod m + m$

*<proof>*

**end**

**end**

**instance** *nat* :: *unique-euclidean-semiring-numeral*

*<proof>*

**instance** *int* :: *unique-euclidean-semiring-numeral*

*<proof>*

**context**

**begin**

**qualified lemma** *zmod-eq-0D*:  $\exists q. m = d * q$  **if**  $m \bmod d = 0$  **for**  $m d$  :: *int*

*<proof>* **lemma** *div-geq* [no-atp]:  $m \operatorname{div} n = \operatorname{Suc} ((m - n) \operatorname{div} n)$  **if**  $0 < n$  **and**  
 $\neg m < n$  **for**  $m n$  :: *nat*

*<proof>* **lemma** *mod-geq* [no-atp]:  $m \bmod n = (m - n) \bmod n$  **if**  $\neg m < n$  **for**  
 $m n$  :: *nat*

*<proof>* **lemma** *mod-eq-0D* [no-atp]:  $\exists q. m = d * q$  **if**  $m \bmod d = 0$  **for**  $m d$  ::  
*nat*

*<proof>* **lemma** *pos-mod-conj* [no-atp]:  $0 < b \implies 0 \leq a \bmod b \wedge a \bmod b < b$   
**for**  $a b$  :: *int*

*<proof>* **lemma** *neg-mod-conj* [no-atp]:  $b < 0 \implies a \bmod b \leq 0 \wedge b < a \bmod b$   
**for**  $a b$  :: *int*

*<proof>* **lemma** *zmod-eq-0-iff* [no-atp]:  $m \bmod d = 0 \longleftrightarrow (\exists q. m = d * q)$  **for**  
 $m d$  :: *int*

*<proof>* **lemma** *div-positive-int* [no-atp]:

$k \operatorname{div} l > 0$  **if**  $k \geq l$  **and**  $l > 0$  **for**  $k l$  :: *int*

*<proof>*

**end**

**code-identifier**

**code-module** *Divides*  $\rightarrow$  (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

**end**

## References

- [1] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming: 10th International Symposium: FLOPS 2010*, volume 6009, 2010.
- [2] D. Leijen. Division and modulus for computer scientists. 2001.
- [3] A. Lochbihler and P. Stoop. Lazy algebraic types in Isabelle/HOL. In *Isabelle Workshop 2018*, 2018.
- [4] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.