The Supplemental Isabelle/HOL Library

June 8, 2008

Contents

1 GCD: The Greatest Common Divisor **9**
- 1.1 Specification of GCD on nats . . . 9
- 1.2 GCD on nat by Euclid's algorithm . . . 9
- 1.3 Derived laws for GCD . . . 10
- 1.4 LCM defined by GCD . . . 12
- 1.5 GCD and LCM on integers . . . 15

2 Abstract-Rat: Abstract rational numbers **18**

3 AssocList: Map operations implemented on association lists **29**
- 3.1 *delete* . . . 31
- 3.2 *clearjunk* . . . 32
- 3.3 *dom* and *ran* . . . 33
- 3.4 *update* . . . 34
- 3.5 *updates* . . . 36
- 3.6 *map-ran* . . . 37
- 3.7 *merge* . . . 38
- 3.8 *compose* . . . 39
- 3.9 *restrict* . . . 42

4 SetsAndFunctions: Operations on sets and functions **45**
- 4.1 Basic definitions . . . 45
- 4.2 Basic properties . . . 48

5 BigO: Big O notation **53**
- 5.1 Definitions . . . 53
- 5.2 Setsum . . . 64
- 5.3 Misc useful stuff . . . 66
- 5.4 Less than or equal to . . . 67

6 Binomial: Binomial Coefficients **70**
- 6.1 Theorems about *choose* . . . 72

7 Boolean-Algebra: Boolean Algebras 74
7.1 Complement . . . 75
7.2 Conjunction . . . 75
7.3 Disjunction . . . 76
7.4 De Morgan's Laws . . . 77
7.5 Symmetric Difference . . . 77

8 Product-ord: Order on product types 79

9 Char-nat: Mapping between characters and natural numbers 80

10 Char-ord: Order on characters 85

11 Code-Char: Code generation of pretty characters (and strings) 87

12 Code-Integer: Pretty integer literals for code generation 88

13 Code-Char-chr: Code generation of pretty characters with character codes 90

14 Code-Index: Type of indices 90
14.1 Datatype of indices . . . 91
14.2 Indices as datatype of ints . . . 93
14.3 Basic arithmetic . . . 93
14.4 ML interface . . . 95
14.5 Specialized *op −*, *op div* and *op mod* operations . . . 96
14.6 Code serialization . . . 96

15 Code-Message: Monolithic strings (message strings) for code generation 97
15.1 Datatype of messages . . . 97
15.2 ML interface . . . 98
15.3 Code serialization . . . 98

16 Coinductive-List: Potentially infinite lists as greatest fixed-point 99
16.1 List constructors over the datatype universe . . . 99
16.2 Corecursive lists . . . 99
16.3 Abstract type definition . . . 101
16.4 Equality as greatest fixed-point – the bisimulation principle . 104
16.5 Derived operations – both on the set and abstract type . . . 109
16.5.1 *Lconst* . . . 109
16.5.2 *Lmap* and *lmap* . . . 110
16.5.3 *Lappend* . . . 112

16.6 iterates . . . 113
16.7 A rather complex proof about iterates – cf. Andy Pitts . . . 114

17 Parity: Even and Odd for int and nat 115
17.1 Even and odd are mutually exclusive . . . 116
17.2 Behavior under integer arithmetic operations . . . 116
17.3 Equivalent definitions . . . 117
17.4 even and odd for nats . . . 118
17.5 Equivalent definitions . . . 118
17.6 Parity and powers . . . 119
17.7 General Lemmas About Division . . . 122
17.8 More Even/Odd Results . . . 122
17.9 An Equivalence for $0 \leq a\hat{}n$ . . . 124
17.10 Miscellaneous . . . 124

18 Commutative-Ring: Proving equalities in commutative rings 125

19 Continuity: Continuity and iterations (of set transformers) 131
19.1 Continuity for complete lattices . . . 131
19.2 Chains . . . 133
19.3 Continuity . . . 134
19.4 Iteration . . . 135

20 Countable: Encoding (almost) everything into natural numbers 137
20.1 The class of countable types . . . 137
20.2 Conversion functions . . . 137
20.3 Countable types . . . 138

21 Dense-Linear-Order: Dense linear order without endpoints and a quantifier elimination procedure in Ferrante and Rackoff style 141

22 The classical QE after Langford for dense linear orders 146

23 Contructive dense linear orders yield QE for linear arithmetic over ordered Fields – see *Arith-Tools.thy* 147
23.1 Ferrante and Rackoff algorithm over ordered fields . . . 153

24 Efficient-Nat: Implementation of natural numbers by target-language integers 160
24.1 Basic arithmetic . . . 160
24.2 Case analysis . . . 161
24.3 Preprocessors . . . 162
24.4 Target language setup . . . 162

25 Enum: Finite types as explicit enumerations **166**
25.1 Class *enum* . . . 166
25.2 Equality and order on functions . . . 166
25.3 Quantifiers . . . 167
25.4 Default instances . . . 167

26 RType: Reflecting Pure types into HOL **173**

27 Eval: A simple term evaluation mechanism **175**
27.1 Term representation . . . 176
27.1.1 Terms and class *term-of* . . . 176
27.1.2 *term-of* instances . . . 176
27.1.3 Code generator setup . . . 178
27.1.4 Syntax . . . 179
27.2 Evaluation setup . . . 180

28 Eval-Witness: Evaluation Oracle with ML witnesses **181**
28.1 Toy Examples . . . 183
28.2 Discussion . . . 183
28.2.1 Conflicts . . . 183
28.2.2 Haskell . . . 183

29 Executable-Set: Implementation of finite sets by lists **184**
29.1 Definitional rewrites . . . 184
29.2 Operations on lists . . . 184
29.2.1 Basic definitions . . . 184
29.2.2 Derived definitions . . . 185
29.3 Isomorphism proofs . . . 187
29.4 code generator setup . . . 188
29.4.1 const serializations . . . 188

30 FuncSet: Pi and Function Sets **189**
30.1 Basic Properties of *Pi* . . . 190
30.2 Composition With a Restricted Domain: *compose* . . . 190
30.3 Bounded Abstraction: *restrict* . . . 191
30.4 Bijections Between Sets . . . 191
30.5 Extensionality . . . 192
30.6 Cardinality . . . 193

31 Heap: A polymorphic heap based on cantor encodings **193**
31.1 Representable types . . . 193
31.2 A polymorphic heap with dynamic arrays and references . . . 195
31.3 Imperative references and arrays . . . 196
31.3.1 Primitive operations . . . 196
31.3.2 Interface operations . . . 197

31.3.3 Reference equality . . . 197
31.3.4 Properties of heap containers . . . 198

32 Heap-Monad: A monad with a polymorphic heap 202
32.1 The monad . . . 202
32.1.1 Monad combinators . . . 202
32.1.2 do-syntax . . . 204
32.1.3 Plain evaluation . . . 205
32.2 Monad properties . . . 205
32.2.1 Superfluous runs . . . 205
32.2.2 Monad laws . . . 206
32.3 Generic combinators . . . 206
32.4 Code generator setup . . . 207
32.4.1 Logical intermediate layer . . . 207
32.4.2 SML . . . 208
32.4.3 OCaml . . . 208
32.4.4 Haskell . . . 208

33 Array: Monadic arrays 209
33.1 Primitives . . . 209
33.2 Derivates . . . 210
33.3 Properties . . . 211
33.4 Code generator setup . . . 212
33.4.1 Logical intermediate layer . . . 212
33.4.2 SML . . . 213
33.4.3 OCaml . . . 213
33.4.4 Haskell . . . 213

34 Ref: Monadic references 213
34.1 Primitives . . . 214
34.2 Derivates . . . 214
34.3 Properties . . . 214
34.4 Code generator setup . . . 214
34.4.1 SML . . . 214
34.4.2 OCaml . . . 215
34.4.3 Haskell . . . 215

35 Imperative-HOL: Entry point into monadic imperative HOL 215

36 Infinite-Set: Infinite Sets and Related Concepts 215
36.1 Infinite Sets . . . 215
36.2 Infinitely Many and Almost All . . . 222
36.3 Enumeration of an Infinite Set . . . 223
36.4 Miscellaneous . . . 224

37 ListVector: Lists as vectors **224**
37.1 + and − . . . 225
37.2 Inner product . . . 226

38 Multiset: Multisets **227**
38.1 The type of multisets . . . 227
38.2 Algebraic properties . . . 229
38.2.1 Union . . . 229
38.2.2 Difference . . . 230
38.2.3 Count of elements . . . 230
38.2.4 Set of elements . . . 230
38.2.5 Size . . . 231
38.2.6 Equality of multisets . . . 232
38.2.7 Intersection . . . 234
38.2.8 Comprehension (filter) . . . 234
38.3 Induction and case splits . . . 235
38.4 Orderings . . . 237
38.4.1 Well-foundedness . . . 237
38.4.2 Closure-free presentation . . . 239
38.4.3 Partial-order properties . . . 241
38.4.4 Monotonicity of multiset union . . . 242
38.5 Link with lists . . . 244
38.6 Pointwise ordering induced by count . . . 246
38.7 Strong induction and subset induction for multisets . . . 250
38.8 The fold combinator . . . 252
38.9 Image . . . 255

39 NatPair: Pairs of Natural Numbers **256**

40 Nat-Infinity: Natural numbers with infinity **258**
40.1 Definitions . . . 258
40.2 Constructors . . . 259
40.3 Ordering relations . . . 260
40.4 Well-ordering . . . 262

41 Nested-Environment: Nested environments **263**
41.1 The lookup operation . . . 264
41.2 The update operation . . . 268

42 Numeral-Type: Numeral Syntax for Types **274**
42.1 Preliminary lemmas . . . 274
42.2 Cardinalities of types . . . 275
42.3 Numeral Types . . . 276
42.4 Syntax . . . 277

42.5 Classes with at least 1 and 2 . . . 278
42.6 Examples . . . 279

43 Option-ord: Canonical order on option type 279

44 Order-Relation: Orders as Relations 280
44.1 Orders on a set . . . 281
44.2 Orders on the field . . . 282
44.3 Orders on a type . . . 282

45 Permutation: Permutations 282
45.1 Some examples of rule induction on permutations . . . 283
45.2 Ways of making new permutations . . . 283
45.3 Further results . . . 284
45.4 Removing elements . . . 284

46 Primes: Primality on nat 286

47 Quicksort: Quicksort 308

48 Quotient: Quotient types 309
48.1 Equivalence relations and quotient types . . . 309
48.2 Equality on quotients . . . 310
48.3 Picking representing elements . . . 311

49 Ramsey: Ramsey's Theorem 313
49.1 Preliminaries . . . 313
49.1.1 "Axiom" of Dependent Choice . . . 313
49.1.2 Partitions of a Set . . . 313
49.2 Ramsey's Theorem: Infinitary Version . . . 314
49.3 Disjunctive Well-Foundedness . . . 317

50 RBT: Red-Black Trees 319
50.1 Data type and invariant . . . 319
50.2 Operations . . . 320
50.3 Invariant preservation . . . 320
50.4 Map Semantics . . . 321

51 State-Monad: Combinators syntax for generic, open state monads (single threaded monads) 321
51.1 Motivation . . . 321
51.2 State transformations and combinators . . . 322
51.3 Obsolete runs . . . 323
51.4 Monad laws . . . 323
51.5 ML abstract operations . . . 324

51.6 Syntax . . . 324
51.7 Combinators . . . 325

52 Univ-Poly: Univariate Polynomials 327
52.1 Arithmetic Operations on Polynomials . . . 327
52.2 Key Property: if $f\ a = (0::'a)$ then $x - a$ divides $p\ x$ . . . 331
52.3 Polynomial length . . . 332

53 While-Combinator: A general "while" combinator 349

54 Word: Binary Words 351
54.1 Auxilary Lemmas . . . 351
54.2 Bits . . . 351
54.3 Bit Vectors . . . 353
54.4 Unsigned Arithmetic Operations . . . 366
54.5 Signed Vectors . . . 368
54.6 Signed Arithmetic Operations . . . 380
54.6.1 Conversion from unsigned to signed . . . 380
54.6.2 Unary minus . . . 380
54.7 Structural operations . . . 392

55 Zorn: Zorn's Lemma 400
55.1 Mathematical Preamble . . . 401
55.2 Hausdorff's Theorem: Every Set Contains a Maximal Chain. 403
55.3 Zorn's Lemma: If All Chains Have Upper Bounds Then There Is a Maximal Element . . . 404
55.4 Alternative version of Zorn's Lemma . . . 405

56 List-Prefix: List prefixes and postfixes 410
56.1 Prefix order on lists . . . 410
56.2 Basic properties of prefixes . . . 411
56.3 Parallel lists . . . 414
56.4 Postfix order on lists . . . 415
56.5 Executable code . . . 418

57 List-lexord: Lexicographic order on lists 418

58 Sublist-Order: Sublist Ordering 420
58.1 Definitions and basic lemmas . . . 420
58.2 Appending elements . . . 423
58.3 Relation to standard list operations . . . 423

1 GCD: The Greatest Common Divisor

theory *GCD*
imports *ATP-Linkup*
begin

See [3].

1.1 Specification of GCD on nats

definition
is-gcd :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where** — *gcd* as a relation
is-gcd p m n ⟷ *p dvd m* ∧ *p dvd n* ∧
(∀ *d*. *d dvd m* ⟶ *d dvd n* ⟶ *d dvd p*)

Uniqueness

lemma *is-gcd-unique*: *is-gcd m a b* ⟹ *is-gcd n a b* ⟹ *m* = *n*
by (*simp add*: *is-gcd-def*) (*blast intro*: *dvd-anti-sym*)

Connection to divides relation

lemma *is-gcd-dvd*: *is-gcd m a b* ⟹ *k dvd a* ⟹ *k dvd b* ⟹ *k dvd m*
by (*auto simp add*: *is-gcd-def*)

Commutativity

lemma *is-gcd-commute*: *is-gcd k m n* = *is-gcd k n m*
by (*auto simp add*: *is-gcd-def*)

1.2 GCD on nat by Euclid's algorithm

fun
gcd :: *nat* × *nat* => *nat*
where
gcd (*m*, *n*) = (*if n* = *0 then m else gcd* (*n*, *m mod n*))

lemma *gcd-induct*:
fixes *m n* :: *nat*
assumes ⋀*m*. *P m 0*
and ⋀*m n*. *0* < *n* ⟹ *P n* (*m mod n*) ⟹ *P m n*
shows *P m n*
apply (*rule gcd.induct* [*of split P* (*m*, *n*), *unfolded Product-Type.split*])
apply (*case-tac n* = *0*)
apply *simp-all*
using *assms* **apply** *simp-all*
done

lemma *gcd-0* [*simp*]: *gcd* (*m*, *0*) = *m*
by *simp*

lemma *gcd-0-left* [*simp*]: *gcd* (*0*, *m*) = *m*

by *simp*

lemma *gcd-non-0*: $n > 0 \Longrightarrow gcd\ (m,\ n) = gcd\ (n,\ m\ mod\ n)$
by *simp*

lemma *gcd-1* [*simp*]: $gcd\ (m,\ Suc\ 0) = 1$
by *simp*

declare *gcd.simps* [*simp del*]

$gcd\ (m,\ n)$ divides m and n. The conjunctions don't seem provable separately.

lemma *gcd-dvd1* [*iff*]: $gcd\ (m,\ n)\ dvd\ m$
and *gcd-dvd2* [*iff*]: $gcd\ (m,\ n)\ dvd\ n$
apply (*induct m n rule*: *gcd-induct*)
apply (*simp-all add*: *gcd-non-0*)
apply (*blast dest*: *dvd-mod-imp-dvd*)
done

Maximality: for all m, n, k naturals, if k divides m and k divides n then k divides $gcd\ (m,\ n)$.

lemma *gcd-greatest*: $k\ dvd\ m \Longrightarrow k\ dvd\ n \Longrightarrow k\ dvd\ gcd\ (m,\ n)$
by (*induct m n rule*: *gcd-induct*) (*simp-all add*: *gcd-non-0 dvd-mod*)

Function gcd yields the Greatest Common Divisor.

lemma *is-gcd*: *is-gcd* $(gcd\ (m,\ n))\ m\ n$
by (*simp add*: *is-gcd-def gcd-greatest*)

1.3 Derived laws for GCD

lemma *gcd-greatest-iff* [*iff*]: $k\ dvd\ gcd\ (m,\ n) \longleftrightarrow k\ dvd\ m \wedge k\ dvd\ n$
by (*blast intro*!: *gcd-greatest intro*: *dvd-trans*)

lemma *gcd-zero*: $gcd\ (m,\ n) = 0 \longleftrightarrow m = 0 \wedge n = 0$
by (*simp only*: *dvd-0-left-iff* [*symmetric*] *gcd-greatest-iff*)

lemma *gcd-commute*: $gcd\ (m,\ n) = gcd\ (n,\ m)$
apply (*rule is-gcd-unique*)
apply (*rule is-gcd*)
apply (*subst is-gcd-commute*)
apply (*simp add*: *is-gcd*)
done

lemma *gcd-assoc*: $gcd\ (gcd\ (k,\ m),\ n) = gcd\ (k,\ gcd\ (m,\ n))$
apply (*rule is-gcd-unique*)
apply (*rule is-gcd*)
apply (*simp add*: *is-gcd-def*)
apply (*blast intro*: *dvd-trans*)

done

lemma *gcd-1-left* [*simp*]: *gcd* (*Suc 0*, *m*) = *1*
by (*simp add*: *gcd-commute*)

Multiplication laws

lemma *gcd-mult-distrib2*: *k* ∗ *gcd* (*m*, *n*) = *gcd* (*k* ∗ *m*, *k* ∗ *n*)
— [3, page 27]
apply (*induct m n rule*: *gcd-induct*)
apply *simp*
apply (*case-tac k* = *0*)
apply (*simp-all add*: *mod-geq gcd-non-0 mod-mult-distrib2*)
done

lemma *gcd-mult* [*simp*]: *gcd* (*k*, *k* ∗ *n*) = *k*
apply (*rule gcd-mult-distrib2* [*of k 1 n*, *simplified*, *symmetric*])
done

lemma *gcd-self* [*simp*]: *gcd* (*k*, *k*) = *k*
apply (*rule gcd-mult* [*of k 1*, *simplified*])
done

lemma *relprime-dvd-mult*: *gcd* (*k*, *n*) = *1* ==> *k dvd m* ∗ *n* ==> *k dvd m*
apply (*insert gcd-mult-distrib2* [*of m k n*])
apply *simp*
apply (*erule-tac t* = *m* **in** *ssubst*)
apply *simp*
done

lemma *relprime-dvd-mult-iff*: *gcd* (*k*, *n*) = *1* ==> (*k dvd m* ∗ *n*) = (*k dvd m*)
apply (*blast intro*: *relprime-dvd-mult dvd-trans*)
done

lemma *gcd-mult-cancel*: *gcd* (*k*, *n*) = *1* ==> *gcd* (*k* ∗ *m*, *n*) = *gcd* (*m*, *n*)
apply (*rule dvd-anti-sym*)
apply (*rule gcd-greatest*)
apply (*rule-tac n* = *k* **in** *relprime-dvd-mult*)
apply (*simp add*: *gcd-assoc*)
apply (*simp add*: *gcd-commute*)
apply (*simp-all add*: *mult-commute*)
apply (*blast intro*: *dvd-trans*)
done

Addition laws

lemma *gcd-add1* [*simp*]: *gcd* (*m* + *n*, *n*) = *gcd* (*m*, *n*)
apply (*case-tac n* = *0*)
apply (*simp-all add*: *gcd-non-0*)
done

lemma *gcd-add2* [*simp*]: *gcd* (*m*, *m* + *n*) = *gcd* (*m*, *n*)
proof −
have *gcd* (*m*, *m* + *n*) = *gcd* (*m* + *n*, *m*) **by** (*rule gcd-commute*)
also have ... = *gcd* (*n* + *m*, *m*) **by** (*simp add*: *add-commute*)
also have ... = *gcd* (*n*, *m*) **by** *simp*
also have ... = *gcd* (*m*, *n*) **by** (*rule gcd-commute*)
finally show *?thesis* **.**
qed

lemma *gcd-add2'* [*simp*]: *gcd* (*m*, *n* + *m*) = *gcd* (*m*, *n*)
apply (*subst add-commute*)
apply (*rule gcd-add2*)
done

lemma *gcd-add-mult*: *gcd* (*m*, *k* ∗ *m* + *n*) = *gcd* (*m*, *n*)
by (*induct k*) (*simp-all add*: *add-assoc*)

lemma *gcd-dvd-prod*: *gcd* (*m*, *n*) *dvd m* ∗ *n*
using *mult-dvd-mono* [*of 1*] **by** *auto*

Division by gcd yields rrelatively primes.

lemma *div-gcd-relprime*:
assumes *nz*: *a* ≠ *0* ∨ *b* ≠ *0*
shows *gcd* (*a div gcd*(*a*,*b*), *b div gcd*(*a*,*b*)) = *1*
proof −
let *?g* = *gcd* (*a*, *b*)
let *?a'* = *a div ?g*
let *?b'* = *b div ?g*
let *?g'* = *gcd* (*?a'*, *?b'*)
have *dvdg*: *?g dvd a ?g dvd b* **by** *simp-all*
have *dvdg'*: *?g' dvd ?a' ?g' dvd ?b'* **by** *simp-all*
from *dvdg dvdg'* **obtain** *ka kb ka' kb'* **where**
kab: *a* = *?g* ∗ *ka b* = *?g* ∗ *kb ?a'* = *?g'* ∗ *ka' ?b'* = *?g'* ∗ *kb'*
unfolding *dvd-def* **by** *blast*
then have *?g* ∗ *?a'* = (*?g* ∗ *?g'*) ∗ *ka' ?g* ∗ *?b'* = (*?g* ∗ *?g'*) ∗ *kb'* **by** *simp-all*
then have *dvdgg'*:*?g* ∗ *?g' dvd a ?g*∗ *?g' dvd b*
by (*auto simp add*: *dvd-mult-div-cancel* [*OF dvdg*(*1*)]
dvd-mult-div-cancel [*OF dvdg*(*2*)] *dvd-def*)
have *?g* ≠ *0* **using** *nz* **by** (*simp add*: *gcd-zero*)
then have *gp*: *?g* > *0* **by** *simp*
from *gcd-greatest* [*OF dvdgg'*] **have** *?g* ∗ *?g' dvd ?g* **.**
with *dvd-mult-cancel1* [*OF gp*] **show** *?g'* = *1* **by** *simp*
qed

1.4 LCM defined by GCD

definition
lcm :: *nat* × *nat* ⇒ *nat*

where
lcm-prim-def: $lcm = (\lambda(m, n).\ m * n\ div\ gcd\ (m, n))$

lemma *lcm-def*:
$lcm\ (m, n) = m * n\ div\ gcd\ (m, n)$
unfolding *lcm-prim-def* **by** *simp*

lemma *prod-gcd-lcm*:
$m * n = gcd\ (m, n) * lcm\ (m, n)$
unfolding *lcm-def* **by** (*simp add*: *dvd-mult-div-cancel* [*OF gcd-dvd-prod*])

lemma *lcm-0* [*simp*]: $lcm\ (m, 0) = 0$
unfolding *lcm-def* **by** *simp*

lemma *lcm-1* [*simp*]: $lcm\ (m, 1) = m$
unfolding *lcm-def* **by** *simp*

lemma *lcm-0-left* [*simp*]: $lcm\ (0, n) = 0$
unfolding *lcm-def* **by** *simp*

lemma *lcm-1-left* [*simp*]: $lcm\ (1, m) = m$
unfolding *lcm-def* **by** *simp*

lemma *dvd-pos*:
fixes $n\ m :: nat$
assumes $n > 0$ **and** $m\ dvd\ n$
shows $m > 0$
using *assms* **by** (*cases m*) *auto*

lemma *lcm-least*:
assumes $m\ dvd\ k$ **and** $n\ dvd\ k$
shows $lcm\ (m, n)\ dvd\ k$
proof (*cases k*)
case *0* **then show** *?thesis* **by** *auto*
next
case (*Suc* -) **then have** *pos-k*: $k > 0$ **by** *auto*
from *assms dvd-pos* [*OF this*] **have** *pos-mn*: $m > 0\ n > 0$ **by** *auto*
with *gcd-zero* [*of m n*] **have** *pos-gcd*: $gcd\ (m, n) > 0$ **by** *simp*
from *assms* **obtain** *p* **where** *k-m*: $k = m * p$ **using** *dvd-def* **by** *blast*
from *assms* **obtain** *q* **where** *k-n*: $k = n * q$ **using** *dvd-def* **by** *blast*
from *pos-k k-m* **have** *pos-p*: $p > 0$ **by** *auto*
from *pos-k k-n* **have** *pos-q*: $q > 0$ **by** *auto*
have $k * k * gcd\ (q, p) = k * gcd\ (k * q, k * p)$
by (*simp add*: *mult-ac gcd-mult-distrib2*)
also have $\ldots = k * gcd\ (m * p * q, n * q * p)$
by (*simp add*: *k-m* [*symmetric*] *k-n* [*symmetric*])
also have $\ldots = k * p * q * gcd\ (m, n)$
by (*simp add*: *mult-ac gcd-mult-distrib2*)
finally have $(m * p) * (n * q) * gcd\ (q, p) = k * p * q * gcd\ (m, n)$

by (*simp only*: *k-m* [*symmetric*] *k-n* [*symmetric*])
then have $p * q * m * n * gcd\ (q,\ p) = p * q * k * gcd\ (m,\ n)$
by (*simp add*: *mult-ac*)
with *pos-p pos-q* **have** $m * n * gcd\ (q,\ p) = k * gcd\ (m,\ n)$
by *simp*
with *prod-gcd-lcm* [*of m n*]
have $lcm\ (m,\ n) * gcd\ (q,\ p) * gcd\ (m,\ n) = k * gcd\ (m,\ n)$
by (*simp add*: *mult-ac*)
with *pos-gcd* **have** $lcm\ (m,\ n) * gcd\ (q,\ p) = k$ **by** *simp*
then show *?thesis* **using** *dvd-def* **by** *auto*
qed

lemma *lcm-dvd1* [*iff*]:
$m\ dvd\ lcm\ (m,\ n)$
proof (*cases m*)
case *0* **then show** *?thesis* **by** *simp*
next
case (*Suc* -)
then have *mpos*: $m > 0$ **by** *simp*
show *?thesis*
proof (*cases n*)
case *0* **then show** *?thesis* **by** *simp*
next
case (*Suc* -)
then have *npos*: $n > 0$ **by** *simp*
have $gcd\ (m,\ n)\ dvd\ n$ **by** *simp*
then obtain *k* **where** $n = gcd\ (m,\ n) * k$ **using** *dvd-def* **by** *auto*
then have $m * n\ div\ gcd\ (m,\ n) = m * (gcd\ (m,\ n) * k)\ div\ gcd\ (m,\ n)$ **by** (*simp add*: *mult-ac*)
also have $\ldots = m * k$ **using** *mpos npos gcd-zero* **by** *simp*
finally show *?thesis* **by** (*simp add*: *lcm-def*)
qed
qed

lemma *lcm-dvd2* [*iff*]:
$n\ dvd\ lcm\ (m,\ n)$
proof (*cases n*)
case *0* **then show** *?thesis* **by** *simp*
next
case (*Suc* -)
then have *npos*: $n > 0$ **by** *simp*
show *?thesis*
proof (*cases m*)
case *0* **then show** *?thesis* **by** *simp*
next
case (*Suc* -)
then have *mpos*: $m > 0$ **by** *simp*
have $gcd\ (m,\ n)\ dvd\ m$ **by** *simp*
then obtain *k* **where** $m = gcd\ (m,\ n) * k$ **using** *dvd-def* **by** *auto*

then have $m * n$ *div gcd* $(m, n) = (gcd\ (m, n) * k) * n$ *div gcd* (m, n) **by** (*simp add*: *mult-ac*)
also have $\ldots = n * k$ **using** *mpos npos gcd-zero* **by** *simp*
finally show *?thesis* **by** (*simp add*: *lcm-def*)
qed
qed

1.5 GCD and LCM on integers

definition
igcd :: *int* ⇒ *int* ⇒ *int* **where**
igcd i j = *int* (*gcd* (*nat* (*abs i*), *nat* (*abs j*)))

lemma *igcd-dvd1* [*simp*]: *igcd i j dvd i*
by (*simp add*: *igcd-def int-dvd-iff*)

lemma *igcd-dvd2* [*simp*]: *igcd i j dvd j*
by (*simp add*: *igcd-def int-dvd-iff*)

lemma *igcd-pos*: *igcd i j* ≥ *0*
by (*simp add*: *igcd-def*)

lemma *igcd0* [*simp*]: (*igcd i j* = *0*) = (*i* = *0* ∧ *j* = *0*)
by (*simp add*: *igcd-def gcd-zero*) *arith*

lemma *igcd-commute*: *igcd i j* = *igcd j i*
unfolding *igcd-def* **by** (*simp add*: *gcd-commute*)

lemma *igcd-neg1* [*simp*]: *igcd* (− *i*) *j* = *igcd i j*
unfolding *igcd-def* **by** *simp*

lemma *igcd-neg2* [*simp*]: *igcd i* (− *j*) = *igcd i j*
unfolding *igcd-def* **by** *simp*

lemma *zrelprime-dvd-mult*: *igcd i j* = *1* ⟹ *i dvd k* * *j* ⟹ *i dvd k*
unfolding *igcd-def*
proof −
assume *int* (*gcd* (*nat* |*i*|, *nat* |*j*|)) = *1 i dvd k* * *j*
then have *g*: *gcd* (*nat* |*i*|, *nat* |*j*|) = *1* **by** *simp*
from ‹*i dvd k* * *j*› **obtain** *h* **where** *h*: *k***j* = *i***h* **unfolding** *dvd-def* **by** *blast*
have *th*: *nat* |*i*| *dvd nat* |*k*| * *nat* |*j*|
unfolding *dvd-def*
by (*rule-tac x*= *nat* |*h*| **in** *exI*, *simp add*: *h nat-abs-mult-distrib* [*symmetric*])
from *relprime-dvd-mult* [*OF g th*] **obtain** *h′* **where** *h′*: *nat* |*k*| = *nat* |*i*| * *h′*
unfolding *dvd-def* **by** *blast*
from *h′* **have** *int* (*nat* |*k*|) = *int* (*nat* |*i*| * *h′*) **by** *simp*
then have |*k*| = |*i*| * *int h′* **by** (*simp add*: *int-mult*)
then show *?thesis*
apply (*subst zdvd-abs1* [*symmetric*])

```
    apply (subst zdvd-abs2 [symmetric])
    apply (unfold dvd-def)
    apply (rule-tac x = int h' in exI, simp)
    done
qed

lemma int-nat-abs: int (nat (abs x)) = abs x  by arith

lemma igcd-greatest:
  assumes k dvd m and k dvd n
  shows k dvd igcd m n
proof −
  let ?k' = nat |k|
  let ?m' = nat |m|
  let ?n' = nat |n|
  from ‹k dvd m› and ‹k dvd n› have dvd': ?k' dvd ?m' ?k' dvd ?n'
    unfolding zdvd-int by (simp-all only: int-nat-abs zdvd-abs1 zdvd-abs2)
  from gcd-greatest [OF dvd'] have int (nat |k|) dvd igcd m n
    unfolding igcd-def by (simp only: zdvd-int)
  then have |k| dvd igcd m n by (simp only: int-nat-abs)
  then show k dvd igcd m n by (simp add: zdvd-abs1)
qed

lemma div-igcd-relprime:
  assumes nz: a ≠ 0 ∨ b ≠ 0
  shows igcd (a div (igcd a b)) (b div (igcd a b)) = 1
proof −
  from nz have nz': nat |a| ≠ 0 ∨ nat |b| ≠ 0 by arith
  let ?g = igcd a b
  let ?a' = a div ?g
  let ?b' = b div ?g
  let ?g' = igcd ?a' ?b'
  have dvdg: ?g dvd a ?g dvd b by (simp-all add: igcd-dvd1 igcd-dvd2)
  have dvdg': ?g' dvd ?a' ?g' dvd ?b' by (simp-all add: igcd-dvd1 igcd-dvd2)
  from dvdg dvdg' obtain ka kb ka' kb' where
   kab: a = ?g*ka b = ?g*kb ?a' = ?g'*ka' ?b' = ?g' * kb'
    unfolding dvd-def by blast
  then have ?g* ?a' = (?g * ?g') * ka' ?g* ?b' = (?g * ?g') * kb' by simp-all
  then have dvdgg':?g * ?g' dvd a ?g* ?g' dvd b
    by (auto simp add: zdvd-mult-div-cancel [OF dvdg(1)]
      zdvd-mult-div-cancel [OF dvdg(2)] dvd-def)
  have ?g ≠ 0 using nz by simp
  then have gp: ?g ≠ 0 using igcd-pos[where i=a and j=b] by arith
  from igcd-greatest [OF dvdgg'] have ?g * ?g' dvd ?g .
  with zdvd-mult-cancel1 [OF gp] have |?g'| = 1 by simp
  with igcd-pos show ?g' = 1 by simp
qed

definition ilcm = (λi j. int (lcm(nat(abs i),nat(abs j))))
```

lemma *dvd-ilcm-self1*[*simp*]: *i dvd ilcm i j*
by(*simp add*:*ilcm-def dvd-int-iff*)

lemma *dvd-ilcm-self2*[*simp*]: *j dvd ilcm i j*
by(*simp add*:*ilcm-def dvd-int-iff*)

lemma *dvd-imp-dvd-ilcm1*:
assumes *k dvd i* **shows** *k dvd* (*ilcm i j*)
proof −
have *nat*(*abs k*) *dvd nat*(*abs i*) **using** ‹*k dvd i*›
by(*simp add*:*int-dvd-iff*[*symmetric*] *dvd-int-iff*[*symmetric*] *zdvd-abs1*)
thus *?thesis* **by**(*simp add*:*ilcm-def dvd-int-iff*)(*blast intro*: *dvd-trans*)
qed

lemma *dvd-imp-dvd-ilcm2*:
assumes *k dvd j* **shows** *k dvd* (*ilcm i j*)
proof −
have *nat*(*abs k*) *dvd nat*(*abs j*) **using** ‹*k dvd j*›
by(*simp add*:*int-dvd-iff*[*symmetric*] *dvd-int-iff*[*symmetric*] *zdvd-abs1*)
thus *?thesis* **by**(*simp add*:*ilcm-def dvd-int-iff*)(*blast intro*: *dvd-trans*)
qed

lemma *zdvd-self-abs1*: (*d*::*int*) *dvd* (*abs d*)
by (*case-tac d <0*, *simp-all*)

lemma *zdvd-self-abs2*: (*abs* (*d*::*int*)) *dvd d*
by (*case-tac d<0*, *simp-all*)

lemma *lcm-pos*:
assumes *mpos*: *m* > *0*
and *npos*: *n>0*
shows *lcm* (*m*,*n*) > *0*
proof(*rule ccontr*, *simp add*: *lcm-def gcd-zero*)
assume *h*:*m*∗*n div gcd*(*m*,*n*) = *0*
from *mpos npos* **have** *gcd* (*m*,*n*) ≠ *0* **using** *gcd-zero* **by** *simp*
hence *gcdp*: *gcd*(*m*,*n*) > *0* **by** *simp*
with *h*
have *m*∗*n* < *gcd*(*m*,*n*)
by (*cases m* ∗ *n* < *gcd* (*m*, *n*)) (*auto simp add*: *div-if*[*OF gcdp*, **where** *m*=*m*∗*n*])
moreover
have *gcd*(*m*,*n*) *dvd m* **by** *simp*
with *mpos dvd-imp-le* **have** *t1*:*gcd*(*m*,*n*) ≤ *m* **by** *simp*
with *npos* **have** *t1*:*gcd*(*m*,*n*)∗*n* ≤ *m*∗*n* **by** *simp*
have *gcd*(*m*,*n*) ≤ *gcd*(*m*,*n*)∗*n* **using** *npos* **by** *simp*

with *t1* **have** $gcd(m,n) \leq m*n$ **by** *arith*
ultimately show *False* **by** *simp*
qed

lemma *ilcm-pos*:
assumes *anz*: $a \neq 0$
and *bnz*: $b \neq 0$
shows $0 < ilcm\ a\ b$
proof−
let *?na* = $nat\ (abs\ a)$
let *?nb* = $nat\ (abs\ b)$
have *nap*: *?na* >*0* **using** *anz* **by** *simp*
have *nbp*: *?nb* >*0* **using** *bnz* **by** *simp*
have $0 < lcm\ (?na,?nb)$ **by** (*rule lcm-pos*[*OF nap nbp*])
thus *?thesis* **by** (*simp add*: *ilcm-def*)
qed

end

2 Abstract-Rat: Abstract rational numbers

theory *Abstract-Rat*
imports *GCD*
begin

types $Num = int \times int$

abbreviation
Num0-syn :: *Num* (0_N)
where $0_N \equiv (0,\ 0)$

abbreviation
Numi-syn :: $int \Rightarrow Num$ (-_N)
where $i_N \equiv (i,\ 1)$

definition
isnormNum :: $Num \Rightarrow bool$
where
$isnormNum = (\lambda(a,b).\ (if\ a = 0\ then\ b = 0\ else\ b > 0 \wedge igcd\ a\ b = 1))$

definition
normNum :: $Num \Rightarrow Num$
where
$normNum = (\lambda(a,b).\ (if\ a=0 \vee b = 0\ then\ (0,0)\ else$
(*let* $g = igcd\ a\ b$
in if $b > 0$ *then* $(a\ div\ g,\ b\ div\ g)$ *else* $(-\ (a\ div\ g), -\ (b\ div\ g)))))$

lemma *normNum-isnormNum* [*simp*]: $isnormNum\ (normNum\ x)$

proof −
have ∃ *a b*. *x* = (*a*,*b*) **by** *auto*
then obtain *a b* **where** *x*[*simp*]: *x* = (*a*,*b*) **by** *blast*
{assume *a*=*0* ∨ *b* = *0* **hence** *?thesis* **by** (*simp add*: *normNum-def isnormNum-def*)**}**

moreover
{assume *anz*: *a* ≠ *0* **and** *bnz*: *b* ≠ *0*
let *?g* = *igcd a b*
let *?a'* = *a div ?g*
let *?b'* = *b div ?g*
let *?g'* = *igcd ?a' ?b'*
from *anz bnz* **have** *?g* ≠ *0* **by** *simp* **with** *igcd-pos*[*of a b*]
have *gpos*: *?g* > *0* **by** *arith*
have *gdvd*: *?g dvd a ?g dvd b* **by** (*simp-all add*: *igcd-dvd1 igcd-dvd2*)
from *zdvd-mult-div-cancel*[*OF gdvd*(*1*)] *zdvd-mult-div-cancel*[*OF gdvd*(*2*)]
anz bnz
have *nz'*:*?a'* ≠ *0 ?b'* ≠ *0*
by − (*rule notI*,*simp add*:*igcd-def*)+
from *anz bnz* **have** *stupid*: *a* ≠ *0* ∨ *b* ≠ *0* **by** *blast*
from *div-igcd-relprime*[*OF stupid*] **have** *gp1*: *?g'* = *1* **.**
from *bnz* **have** *b* < *0* ∨ *b* > *0* **by** *arith*
moreover
{assume *b*: *b* > *0*
from *pos-imp-zdiv-nonneg-iff*[*OF gpos*] *b*
have *?b'* ≥ *0* **by** *simp*
with *nz'* **have** *b'*: *?b'* > *0* **by** *simp*
from *b b' anz bnz nz' gp1* **have** *?thesis*
by (*simp add*: *isnormNum-def normNum-def Let-def split-def fst-conv snd-conv*)**}**
moreover {assume *b*: *b* < *0*
{assume *b'*: *?b'* ≥ *0*
from *gpos* **have** *th*: *?g* ≥ *0* **by** *arith*
from *mult-nonneg-nonneg*[*OF th b'*] *zdvd-mult-div-cancel*[*OF gdvd*(*2*)]
have *False* **using** *b* **by** *simp* **}**
hence *b'*: *?b'* < *0* **by** (*presburger add*: *linorder-not-le*[*symmetric*])
from *anz bnz nz' b b' gp1* **have** *?thesis*
by (*simp add*: *isnormNum-def normNum-def Let-def split-def fst-conv snd-conv*)**}**
ultimately have *?thesis* **by** *blast*
}
ultimately show *?thesis* **by** *blast*
qed

Arithmetic over Num

definition
Nadd :: *Num* ⇒ *Num* ⇒ *Num* (**infixl** +$_N$ *60*)
where
Nadd = (λ(*a*,*b*) (*a'*,*b'*). *if a* = *0* ∨ *b* = *0 then normNum*(*a'*,*b'*)
else if a'=*0* ∨ *b'* = *0 then normNum*(*a*,*b*)

else normNum$(a*b' + b*a', b*b')$)

definition
Nmul :: *Num* ⇒ *Num* ⇒ *Num* (**infixl** $*_N$ *60*)
where
Nmul = (λ(*a*,*b*) (*a'*,*b'*). *let g* = *igcd* (*a****a'*) (*b****b'*)
in (*a****a' div g*, *b****b' div g*))

definition
Nneg :: *Num* ⇒ *Num* ($^\sim_N$)
where
Nneg ≡ (λ(*a*,*b*). (−*a*,*b*))

definition
Nsub :: *Num* ⇒ *Num* ⇒ *Num* (**infixl** $-_N$ *60*)
where
Nsub = (λ*a b*. *a* $+_N$ $^\sim_N$ *b*)

definition
Ninv :: *Num* ⇒ *Num*
where
Ninv ≡ λ(*a*,*b*). *if a* < *0 then* (−*b*, |*a*|) *else* (*b*,*a*)

definition
Ndiv :: *Num* ⇒ *Num* ⇒ *Num* (**infixl** $\div_N$ *60*)
where
Ndiv ≡ λ*a b*. *a* $*_N$ *Ninv b*

lemma *Nneg-normN*[*simp*]: *isnormNum x* ⟹ *isnormNum* ($^\sim_N$ *x*)
by(*simp add*: *isnormNum-def Nneg-def split-def*)
lemma *Nadd-normN*[*simp*]: *isnormNum* (*x* $+_N$ *y*)
by (*simp add*: *Nadd-def split-def*)
lemma *Nsub-normN*[*simp*]: ⟦ *isnormNum y*⟧ ⟹ *isnormNum* (*x* $-_N$ *y*)
by (*simp add*: *Nsub-def split-def*)
lemma *Nmul-normN*[*simp*]: **assumes** *xn*:*isnormNum x* **and** *yn*: *isnormNum y*
shows *isnormNum* (*x* $*_N$ *y*)
proof−
have ∃ *a b*. *x* = (*a*,*b*) **and** ∃ *a' b'*. *y* = (*a'*,*b'*) **by** *auto*
then obtain *a b a' b'* **where** *ab*: *x* = (*a*,*b*) **and** *ab'*: *y* = (*a'*,*b'*) **by** *blast*
{assume *a* = *0*
hence *?thesis* **using** *xn ab ab'*
by (*simp add*: *igcd-def isnormNum-def Let-def Nmul-def split-def*)**}**
moreover
{assume *a'* = *0*
hence *?thesis* **using** *yn ab ab'*
by (*simp add*: *igcd-def isnormNum-def Let-def Nmul-def split-def*)**}**
moreover
{assume *a*: *a* ≠*0* **and** *a'*: *a'*≠*0*
hence *bp*: *b* > *0 b'* > *0* **using** *xn yn ab ab'* **by** (*simp-all add*: *isnormNum-def*)

from *mult-pos-pos*[*OF bp*] **have** $x *_N y = normNum\ (a*a', b*b')$
using *ab ab' a a' bp* **by** (*simp add*: *Nmul-def Let-def split-def normNum-def*)
hence *?thesis* **by** *simp*}
ultimately show *?thesis* **by** *blast*
qed

lemma *Ninv-normN*[*simp*]: $isnormNum\ x \Longrightarrow isnormNum\ (Ninv\ x)$
by (*simp add*: *Ninv-def isnormNum-def split-def*)
(*cases fst x = 0*, *auto simp add*: *igcd-commute*)

lemma *isnormNum-int*[*simp*]:
$isnormNum\ 0_N\ isnormNum\ (1::int)_N\ i \neq 0 \Longrightarrow isnormNum\ i_N$
by (*simp-all add*: *isnormNum-def igcd-def*)

Relations over Num

definition
$Nlt0:: Num \Rightarrow bool\ (0>_N)$
where
$Nlt0 = (\lambda(a,b).\ a < 0)$

definition
$Nle0:: Num \Rightarrow bool\ (0\geq_N)$
where
$Nle0 = (\lambda(a,b).\ a \leq 0)$

definition
$Ngt0:: Num \Rightarrow bool\ (0<_N)$
where
$Ngt0 = (\lambda(a,b).\ a > 0)$

definition
$Nge0:: Num \Rightarrow bool\ (0\leq_N)$
where
$Nge0 = (\lambda(a,b).\ a \geq 0)$

definition
$Nlt :: Num \Rightarrow Num \Rightarrow bool\ (\textbf{infix}\ <_N\ 55)$
where
$Nlt = (\lambda a\ b.\ 0>_N\ (a\ -_N\ b))$

definition
$Nle :: Num \Rightarrow Num \Rightarrow bool\ (\textbf{infix}\ \leq_N\ 55)$
where
$Nle = (\lambda a\ b.\ 0\geq_N\ (a\ -_N\ b))$

definition
$INum = (\lambda(a,b).\ of\text{-}int\ a\ /\ of\text{-}int\ b)$

lemma *INum-int* [*simp*]: $INum\ i_N = ((of\text{-}int\ i) ::'a::field)\ INum\ 0_N = (0::'a::field)$

by (*simp-all add*: *INum-def*)

lemma *isnormNum-unique*[*simp*]:
assumes *na*: *isnormNum x* **and** *nb*: *isnormNum y*
shows ((*INum x* ::'*a*::{*ring-char-0*,*field*, *division-by-zero*}) = *INum y*) = (*x* = *y*) (**is** *?lhs* = *?rhs*)
proof
have ∃ *a b a' b'*. *x* = (*a*,*b*) ∧ *y* = (*a'*,*b'*) **by** *auto*
then obtain *a b a' b'* **where** *xy*[*simp*]: *x* = (*a*,*b*) *y*=(*a'*,*b'*) **by** *blast*
assume *H*: *?lhs*
{assume *a* = *0* ∨ *b* = *0* ∨ *a'* = *0* ∨ *b'* = *0* **hence** *?rhs*
using *na nb H*
apply (*simp add*: *INum-def split-def isnormNum-def*)
apply (*cases a* = *0*, *simp-all*)
apply (*cases b* = *0*, *simp-all*)
apply (*cases a'* = *0*, *simp-all*)
apply (*cases a'* = *0*, *simp-all add*: *of-int-eq-0-iff*)
done}
moreover
{ assume *az*: *a* ≠ *0* **and** *bz*: *b* ≠ *0* **and** *a'z*: *a'*≠*0* **and** *b'z*: *b'*≠*0*
from *az bz a'z b'z na nb* **have** *pos*: *b* > *0 b'* > *0* **by** (*simp-all add*: *isnormNum-def*)
from *prems* **have** *eq*:*a* ∗ *b'* = *a'*∗*b*
by (*simp add*: *INum-def eq-divide-eq divide-eq-eq of-int-mult*[*symmetric*] *del*: *of-int-mult*)
from *prems* **have** *gcd1*: *igcd a b* = *1 igcd b a* = *1 igcd a' b'* = *1 igcd b' a'* = *1*
by (*simp-all add*: *isnormNum-def add*: *igcd-commute*)
from *eq* **have** *raw-dvd*: *a dvd a'*∗*b b dvd b'*∗*a a' dvd a*∗*b' b' dvd b*∗*a'*
apply(*unfold dvd-def*)
apply (*rule-tac x*=*b'* **in** *exI*, *simp add*: *mult-ac*)
apply (*rule-tac x*=*a'* **in** *exI*, *simp add*: *mult-ac*)
apply (*rule-tac x*=*b* **in** *exI*, *simp add*: *mult-ac*)
apply (*rule-tac x*=*a* **in** *exI*, *simp add*: *mult-ac*)
done
from *zdvd-dvd-eq*[*OF bz zrelprime-dvd-mult*[*OF gcd1*(*2*) *raw-dvd*(*2*)]
zrelprime-dvd-mult[*OF gcd1*(*4*) *raw-dvd*(*4*)]]
have *eq1*: *b* = *b'* **using** *pos* **by** *simp-all*
with *eq* **have** *a* = *a'* **using** *pos* **by** *simp*
with *eq1* **have** *?rhs* **by** *simp***}**
ultimately show *?rhs* **by** *blast*
next
assume *?rhs* **thus** *?lhs* **by** *simp*
qed

lemma *isnormNum0*[*simp*]: *isnormNum x* ⟹ (*INum x* = (*0*::'*a*::{*ring-char-0*, *field*,*division-by-zero*})) = (*x* = 0_N)
unfolding *INum-int*(*2*)[*symmetric*]
by (*rule isnormNum-unique*, *simp-all*)

lemma *of-int-div-aux*: $d \sim= 0 \Longrightarrow ((\textit{of-int}\ x)::'a::\{\textit{field}, \textit{ring-char-0}\})\ /\ (\textit{of-int}\ d) =$
$\quad \textit{of-int}\ (x\ \textit{div}\ d) + (\textit{of-int}\ (x\ \textit{mod}\ d))\ /\ ((\textit{of-int}\ d)::'a)$
proof −
assume $d \sim= 0$
hence *dz*: *of-int* $d \neq (0::'a)$ **by** (*simp add*: *of-int-eq-0-iff*)
let *?t* = *of-int* $(x\ \textit{div}\ d) * ((\textit{of-int}\ d)::'a) + \textit{of-int}(x\ \textit{mod}\ d)$
let *?f* = $\lambda x.\ x\ /\ \textit{of-int}\ d$
have $x = (x\ \textit{div}\ d) * d + x\ \textit{mod}\ d$
by *auto*
then have *eq*: *of-int* x = *?t*
by (*simp only*: *of-int-mult*[*symmetric*] *of-int-add* [*symmetric*])
then have *of-int* x / *of-int* d = *?t* / *of-int* d
using *cong*[*OF refl*[*of ?f*] *eq*] **by** *simp*
then show *?thesis* **by** (*simp add*: *add-divide-distrib ring-simps prems*)
qed

lemma *of-int-div*: $(d::int) \sim= 0 \Longrightarrow d\ dvd\ n \Longrightarrow$
$\quad (\textit{of-int}(n\ \textit{div}\ d)::'a::\{\textit{field}, \textit{ring-char-0}\}) = \textit{of-int}\ n\ /\ \textit{of-int}\ d$
apply (*frule of-int-div-aux* [*of d n*, **where** *?'a* = *'a*])
apply *simp*
apply (*simp add*: *zdvd-iff-zmod-eq-0*)
done

lemma *normNum*[*simp*]: *INum* (*normNum x*) = (*INum x* :: *'a*::{*ring-char-0*,*field*, *division-by-zero*})
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *auto*
then obtain *a b* **where** *x*[*simp*]: *x* = (*a*,*b*) **by** *blast*
{assume *a*=*0* ∨ *b* = *0* **hence** *?thesis*
by (*simp add*: *INum-def normNum-def split-def Let-def*)**}**
moreover
{assume *a*: *a*≠*0* **and** *b*: *b*≠*0*
let *?g* = *igcd a b*
from *a b* **have** *g*: *?g* ≠ *0***by** *simp*
from *of-int-div*[*OF g*, **where** *?'a* = *'a*]
have *?thesis* **by** (*auto simp add*: *INum-def normNum-def split-def Let-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *INum-normNum-iff*: (*INum x* ::*'a*::{*field*, *division-by-zero*, *ring-char-0*}) = *INum y* ⟷ *normNum x* = *normNum y* (**is** *?lhs* = *?rhs*)
proof −
have *normNum x* = *normNum y* ⟷ (*INum* (*normNum x*) :: *'a*) = *INum* (*normNum y*)
by (*simp del*: *normNum*)
also have ... = *?lhs* **by** *simp*

finally show *?thesis* **by** *simp*
qed

lemma *Nadd*[*simp*]: *INum* $(x +_N y) =$ *INum* $x +$ (*INum* $y ::$ *'a* :: {*ring-char-0*,*division-by-zero*,*field*})
proof−
let *?z* = *0*:: *'a*
have ∃ *a b*. $x = (a,b)$ ∃ *a' b'*. $y = (a',b')$ **by** *auto*
then obtain *a b a' b'* **where** *x*[*simp*]: $x = (a,b)$
and *y*[*simp*]: $y = (a',b')$ **by** *blast*
{assume *a=0* ∨ *a'= 0* ∨ *b =0* ∨ *b' = 0* **hence** *?thesis*
apply (*cases a=0*,*simp-all add*: *Nadd-def*)
apply (*cases b= 0*,*simp-all add*: *INum-def*)
apply (*cases a'= 0*,*simp-all*)
apply (*cases b'= 0*,*simp-all*)
done }
moreover
{assume *aa'*:*a* ≠ *0 a'*≠ *0* **and** *bb'*: *b* ≠ *0 b'* ≠ *0*
{assume *z*: *a* ∗ *b'* + *b* ∗ *a'* = *0*
hence *of-int* (*a*∗*b'* + *b*∗*a'*) / (*of-int b*∗ *of-int b'*) = *?z* **by** *simp*
hence *of-int b'* ∗ *of-int a* / (*of-int b* ∗ *of-int b'*) + *of-int b* ∗ *of-int a'* / (*of-int b* ∗ *of-int b'*) = *?z* **by** (*simp add*:*add-divide-distrib*)
hence *th*: *of-int a* / *of-int b* + *of-int a'* / *of-int b'* = *?z* **using** *bb' aa'* **by** *simp*
from *z aa' bb'* **have** *?thesis*
by (*simp add*: *th Nadd-def normNum-def INum-def split-def*)**}**
moreover {assume *z*: *a* ∗ *b'* + *b* ∗ *a'* ≠ *0*
let *?g* = *igcd* (*a* ∗ *b'* + *b* ∗ *a'*) (*b*∗*b'*)
have *gz*: *?g* ≠ *0* **using** *z* **by** *simp*
have *?thesis* **using** *aa' bb' z gz*
of-int-div[**where** *?'a* = *'a*,
OF gz igcd-dvd1[**where** *i*=*a* ∗ *b'* + *b* ∗ *a'* **and** *j*=*b*∗*b'*]]
of-int-div[**where** *?'a* = *'a*,
OF gz igcd-dvd2[**where** *i*=*a* ∗ *b'* + *b* ∗ *a'* **and** *j*=*b*∗*b'*]]
by (*simp add*: *x y Nadd-def INum-def normNum-def Let-def add-divide-distrib*)**}**
ultimately have *?thesis* **using** *aa' bb'*
by (*simp add*: *Nadd-def INum-def normNum-def x y Let-def*) **}**
ultimately show *?thesis* **by** *blast*
qed

lemma *Nmul*[*simp*]: *INum* $(x *_N y) =$ *INum* $x *$ (*INum* $y::$ *'a* :: {*ring-char-0*,*division-by-zero*,*field*})

proof−
let *?z* = *0*::*'a*
have ∃ *a b*. $x = (a,b)$ ∃ *a' b'*. $y = (a',b')$ **by** *auto*
then obtain *a b a' b'* **where** *x*: $x = (a,b)$ **and** *y*: $y = (a',b')$ **by** *blast*
{assume *a=0* ∨ *a'= 0* ∨ *b = 0* ∨ *b' = 0* **hence** *?thesis*
apply (*cases a=0*,*simp-all add*: *x y Nmul-def INum-def Let-def*)
apply (*cases b=0*,*simp-all*)
apply (*cases a'=0*,*simp-all*)

```
    done }
  moreover
  {assume z: a ≠ 0 a' ≠ 0 b ≠ 0 b' ≠ 0
   let ?g=igcd (a*a') (b*b')
   have gz: ?g ≠ 0 using z by simp
  from z of-int-div[where ?'a = 'a, OF gz igcd-dvd1[where i=a*a' and j=b*b']]

     of-int-div[where ?'a = 'a , OF gz igcd-dvd2[where i=a*a' and j=b*b']]
   have ?thesis by (simp add: Nmul-def x y Let-def INum-def)}
  ultimately show ?thesis by blast
qed

lemma Nneg[simp]: INum (~N x) = - (INum x ::'a:: field)
  by (simp add: Nneg-def split-def INum-def)

lemma Nsub[simp]: shows INum (x -N y) = INum x - (INum y:: 'a :: {ring-char-0,division-by-zero,field})
by (simp add: Nsub-def split-def)

lemma Ninv[simp]: INum (Ninv x) = (1::'a :: {division-by-zero,field}) / (INum
x)
  by (simp add: Ninv-def INum-def split-def)

lemma Ndiv[simp]: INum (x ÷N y) = INum x / (INum y ::'a :: {ring-char-0,
division-by-zero,field}) by (simp add: Ndiv-def)

lemma Nlt0-iff[simp]: assumes nx: isnormNum x
  shows ((INum x :: 'a :: {ring-char-0,division-by-zero,ordered-field})< 0) = 0>N
x
proof−
  have ∃ a b. x = (a,b) by simp
  then obtain a b where x[simp]:x = (a,b) by blast
  {assume a = 0 hence ?thesis by (simp add: Nlt0-def INum-def) }
  moreover
  {assume a: a≠0 hence b: (of-int b::'a) > 0 using nx by (simp add: isnormNum-def)
    from pos-divide-less-eq[OF b, where b=of-int a and a=0::'a]
    have ?thesis by (simp add: Nlt0-def INum-def)}
  ultimately show ?thesis by blast
qed

lemma Nle0-iff[simp]:assumes nx: isnormNum x
  shows ((INum x :: 'a :: {ring-char-0,division-by-zero,ordered-field}) ≤ 0) = 0≥N
x
proof−
  have ∃ a b. x = (a,b) by simp
  then obtain a b where x[simp]:x = (a,b) by blast
  {assume a = 0 hence ?thesis by (simp add: Nle0-def INum-def) }
  moreover
  {assume a: a≠0 hence b: (of-int b :: 'a) > 0 using nx by (simp add: isnormNum-def)
    from pos-divide-le-eq[OF b, where b=of-int a and a=0::'a]
```

have *?thesis* **by** (*simp add*: *Nle0-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *Ngt0-iff*[*simp*]:**assumes** *nx*: *isnormNum x* **shows** ((*INum x* :: *'a* :: {*ring-char-0*,*division-by-zero*,*ordere*
0) = *0*<$_N$ *x*
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *simp*
then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{assume *a* = *0* **hence** *?thesis* **by** (*simp add*: *Ngt0-def INum-def*) **}**
moreover
{assume *a*: *a*≠*0* **hence** *b*: (*of-int b*::*'a*) > *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-less-divide-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::*'a*]
have *?thesis* **by** (*simp add*: *Ngt0-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed
lemma *Nge0-iff*[*simp*]:**assumes** *nx*: *isnormNum x*
shows ((*INum x* :: *'a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) ≥ *0*) = *0*≤$_N$
x
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *simp*
then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{assume *a* = *0* **hence** *?thesis* **by** (*simp add*: *Nge0-def INum-def*) **}**
moreover
{assume *a*: *a*≠*0* **hence** *b*: (*of-int b*::*'a*) > *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-le-divide-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::*'a*]
have *?thesis* **by** (*simp add*: *Nge0-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *Nlt-iff*[*simp*]: **assumes** *nx*: *isnormNum x* **and** *ny*: *isnormNum y*
shows ((*INum x* :: *'a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) < *INum y*)
= (*x* <$_N$ *y*)
proof−
let *?z* = *0*::*'a*
have ((*INum x* ::*'a*) < *INum y*) = (*INum* (*x* −$_N$ *y*) < *?z*) **using** *nx ny* **by** *simp*
also have ... = (*0*>$_N$ (*x* −$_N$ *y*)) **using** *Nlt0-iff*[*OF Nsub-normN*[*OF ny*]] **by**
simp
finally show *?thesis* **by** (*simp add*: *Nlt-def*)
qed

lemma *Nle-iff*[*simp*]: **assumes** *nx*: *isnormNum x* **and** *ny*: *isnormNum y*
shows ((*INum x* :: *'a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*})≤ *INum y*)
= (*x* ≤$_N$ *y*)
proof−
have ((*INum x* ::*'a*) ≤ *INum y*) = (*INum* (*x* −$_N$ *y*) ≤ (*0*::*'a*)) **using** *nx ny* **by**
simp
also have ... = (*0*≥$_N$ (*x* −$_N$ *y*)) **using** *Nle0-iff*[*OF Nsub-normN*[*OF ny*]] **by**
simp

finally show *?thesis* **by** (*simp add*: *Nle-def*)
qed

lemma *Nadd-commute*: $x +_N y = y +_N x$
proof−
have *n*: *isnormNum* $(x +_N y)$ *isnormNum* $(y +_N x)$ **by** *simp-all*
have (*INum* $(x +_N y)$::$'a$ :: {*ring-char-0*,*division-by-zero*,*field*}) = *INum* $(y +_N x)$ **by** *simp*
with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

lemma[*simp*]: $(0, b) +_N y$ = *normNum* y $(a, 0) +_N y$ = *normNum* y
$x +_N (0, b)$ = *normNum* x $x +_N (a, 0)$ = *normNum* x
apply (*simp add*: *Nadd-def split-def*, *simp add*: *Nadd-def split-def*)
apply (*subst Nadd-commute*,*simp add*: *Nadd-def split-def*)
apply (*subst Nadd-commute*,*simp add*: *Nadd-def split-def*)
done

lemma *normNum-nilpotent-aux*[*simp*]: **assumes** *nx*: *isnormNum x*
shows *normNum x* = *x*
proof−
let *?a* = *normNum x*
have *n*: *isnormNum ?a* **by** *simp*
have *th*:*INum ?a* = (*INum x* ::$'a$ :: {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
with *isnormNum-unique*[*OF n nx*]
show *?thesis* **by** *simp*
qed

lemma *normNum-nilpotent*[*simp*]: *normNum* (*normNum x*) = *normNum x*
by *simp*
lemma *normNum0*[*simp*]: *normNum* (*0*,*b*) = 0_N *normNum* (*a*,*0*) = 0_N
by (*simp-all add*: *normNum-def*)
lemma *normNum-Nadd*: *normNum* $(x +_N y) = x +_N y$ **by** *simp*
lemma *Nadd-normNum1*[*simp*]: *normNum* $x +_N y = x +_N y$
proof−
have *n*: *isnormNum* (*normNum* $x +_N y$) *isnormNum* $(x +_N y)$ **by** *simp-all*
have *INum* (*normNum* $x +_N y$) = *INum* x + (*INum* y :: $'a$ :: {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
also have ... = *INum* $(x +_N y)$ **by** *simp*
finally show *?thesis* **using** *isnormNum-unique*[*OF n*] **by** *simp*
qed
lemma *Nadd-normNum2*[*simp*]: $x +_N$ *normNum* $y = x +_N y$
proof−
have *n*: *isnormNum* ($x +_N$ *normNum* y) *isnormNum* $(x +_N y)$ **by** *simp-all*
have *INum* ($x +_N$ *normNum* y) = *INum* x + (*INum* y :: $'a$ :: {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
also have ... = *INum* $(x +_N y)$ **by** *simp*
finally show *?thesis* **using** *isnormNum-unique*[*OF n*] **by** *simp*
qed

lemma *Nadd-assoc*: $x +_N y +_N z = x +_N (y +_N z)$
proof−
have *n*: *isnormNum* $(x +_N y +_N z)$ *isnormNum* $(x +_N (y +_N z))$ **by** *simp-all*
have *INum* $(x +_N y +_N z) =$ (*INum* $(x +_N (y +_N z)) :: 'a ::$ {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

lemma *Nmul-commute*: *isnormNum* $x \Longrightarrow$ *isnormNum* $y \Longrightarrow x *_N y = y *_N x$
by (*simp add*: *Nmul-def split-def Let-def igcd-commute mult-commute*)

lemma *Nmul-assoc*: **assumes** *nx*: *isnormNum x* **and** *ny*:*isnormNum y* **and** *nz*:*isnormNum z*
shows $x *_N y *_N z = x *_N (y *_N z)$
proof−
from *nx ny nz* **have** *n*: *isnormNum* $(x *_N y *_N z)$ *isnormNum* $(x *_N (y *_N z))$

by *simp-all*
have *INum* $(x +_N y +_N z) =$ (*INum* $(x +_N (y +_N z)) :: 'a ::$ {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

lemma *Nsub0*: **assumes** *x*: *isnormNum x* **and** *y*:*isnormNum y* **shows** $(x -_N y = 0_N) = (x = y)$
proof−
{fix $h :: 'a ::$ {*ring-char-0*,*division-by-zero*,*ordered-field*}
from *isnormNum-unique*[**where** $?'a = 'a$, *OF Nsub-normN*[*OF y*], **where** $y=0_N$]
have $(x -_N y = 0_N) =$ (*INum* $(x -_N y) =$ (*INum* $0_N :: 'a$)) **by** *simp*
also have $\ldots =$ (*INum x* = (*INum y*:: $'a$)) **by** *simp*
also have $\ldots = (x = y)$ **using** *x y* **by** *simp*
finally show *?thesis* **.}**
qed

lemma *Nmul0*[*simp*]: $c *_N 0_N = 0_N \quad 0_N *_N c = 0_N$
by (*simp-all add*: *Nmul-def Let-def split-def*)

lemma *Nmul-eq0*[*simp*]: **assumes** *nx*:*isnormNum x* **and** *ny*: *isnormNum y*
shows $(x *_N y = 0_N) = (x = 0_N \vee y = 0_N)$
proof−
{fix $h :: 'a ::$ {*ring-char-0*,*division-by-zero*,*ordered-field*}
have $\exists\ a\ b\ a'\ b'.\ x = (a,b) \wedge y = (a',b')$ **by** *auto*
then obtain *a b a' b'* **where** *xy*[*simp*]: $x = (a,b)$ $y = (a',b')$ **by** *blast*
have *n0*: *isnormNum* 0_N **by** *simp*
show *?thesis* **using** *nx ny*
apply (*simp only*: *isnormNum-unique*[**where** $?'a = 'a$, *OF Nmul-normN*[*OF nx ny*] *n0*, *symmetric*] *Nmul*[**where** $?'a = 'a$])

apply (*simp add*: *INum-def split-def isnormNum-def fst-conv snd-conv*)
apply (*cases a=0*,*simp-all*)
apply (*cases a'=0*,*simp-all*)
done }
qed
lemma *Nneg-Nneg*[*simp*]: $\sim_N\ (\sim_N\ c) = c$
by (*simp add*: *Nneg-def split-def*)

lemma *Nmul1*[*simp*]:
$isnormNum\ c \Longrightarrow 1_N *_N c = c$
$isnormNum\ c \Longrightarrow c *_N 1_N = c$
apply (*simp-all add*: *Nmul-def Let-def split-def isnormNum-def*)
by (*cases fst c = 0*, *simp-all*,*cases c*, *simp-all*)+

end

3 AssocList: Map operations implemented on association lists

theory *AssocList*
imports *Map*
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.

primrec
delete :: *'key* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where
delete k [] = []
| *delete k* (*p*#*ps*) = (*if fst p = k then delete k ps else p* # *delete k ps*)

primrec
update :: *'key* ⇒ *'val* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'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*)

primrec
updates :: *'key list* ⇒ *'val list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where
updates [] *vs ps* = *ps*
| *updates* (*k*#*ks*) *vs ps* = (*case vs*
of [] ⇒ *ps*
| (*v*#*vs'*) ⇒ *updates ks vs'* (*update k v ps*))

primrec
merge :: (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*

where
  merge qs [] = *qs*
| *merge qs* (*p*#*ps*) = *update* (*fst p*) (*snd p*) (*merge qs ps*)

lemma *length-delete-le*: *length* (*delete k al*) ≤ *length al*
proof (*induct al*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons a al*)
  note *length-filter-le* [*of* λ*p*. *fst p* ≠ *fst a al*]
  also have ⋀*n*. *n* ≤ *Suc n*
    by *simp*
  finally have *length* [*p*←*al* . *fst p* ≠ *fst a*] ≤ *Suc* (*length al*) **.**
  with *Cons* **show** *?case*
    by *auto*
qed

lemma *compose-hint* [*simp*]:
  length (*delete k al*) < *Suc* (*length al*)
proof −
  note *length-delete-le*
  also have ⋀*n*. *n* < *Suc n*
    by *simp*
  finally show *?thesis* **.**
qed

fun
  compose :: ('*key* × '*a*) *list* ⇒ ('*a* × '*b*) *list* ⇒ ('*key* × '*b*) *list*
where
  compose [] *ys* = []
| *compose* (*x*#*xs*) *ys* = (*case map-of ys* (*snd x*)
    of None ⇒ *compose* (*delete* (*fst x*) *xs*) *ys*
     | *Some v* ⇒ (*fst x*, *v*) # *compose xs ys*)

primrec
  restrict :: '*key set* ⇒ ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list*
where
  restrict A [] = []
| *restrict A* (*p*#*ps*) = (*if fst p* ∈ *A then p*#*restrict A ps else restrict A ps*)

primrec
  map-ran :: ('*key* ⇒ '*val* ⇒ '*val*) ⇒ ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list*
where
  map-ran f [] = []
| *map-ran f* (*p*#*ps*) = (*fst p*, *f* (*fst p*) (*snd p*)) # *map-ran f ps*

fun
  clearjunk :: ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list*
where

clearjunk [] = []
| *clearjunk* (*p*#*ps*) = *p* # *clearjunk* (*delete* (*fst p*) *ps*)

lemmas [*simp del*] = *compose-hint*

3.1 *delete*

lemma *delete-eq*:
delete k xs = *filter* (λ*p*. *fst p* ≠ *k*) *xs*
by (*induct xs*) *auto*

lemma *delete-id* [*simp*]: *k* ∉ *fst* ‘ *set al* ⟹ *delete k al* = *al*
by (*induct al*) *auto*

lemma *delete-conv*: *map-of* (*delete k al*) *k'* = ((*map-of al*)(*k* := *None*)) *k'*
by (*induct al*) *auto*

lemma *delete-conv'*: *map-of* (*delete k al*) = ((*map-of al*)(*k* := *None*))
by (*rule ext*) (*rule delete-conv*)

lemma *delete-idem*: *delete k* (*delete k al*) = *delete k al*
by (*induct al*) *auto*

lemma *map-of-delete* [*simp*]:
k' ≠ *k* ⟹ *map-of* (*delete k al*) *k'* = *map-of al k'*
by (*induct al*) *auto*

lemma *delete-notin-dom*: *k* ∉ *fst* ‘ *set* (*delete k al*)
by (*induct al*) *auto*

lemma *dom-delete-subset*: *fst* ‘ *set* (*delete k al*) ⊆ *fst* ‘ *set al*
by (*induct al*) *auto*

lemma *distinct-delete*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*delete k al*))
using *assms*
proof (*induct al*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a al*)
from *Cons.prems* **obtain**
a-notin-al: *fst a* ∉ *fst* ‘ *set al* **and**
dist-al: *distinct* (*map fst al*)
by *auto*
show *?case*
proof (*cases fst a* = *k*)
case *True*
with *Cons dist-al* **show** *?thesis* **by** *simp*

```
  next
    case False
    from dist-al
    have distinct (map fst (delete k al))
      by (rule Cons.hyps)
    moreover from a-notin-al dom-delete-subset [of k al]
    have fst a ∉ fst ‘ set (delete k al)
      by blast
    ultimately show ?thesis using False by simp
  qed
qed

lemma delete-twist: delete x (delete y al) = delete y (delete x al)
  by (induct al) auto

lemma clearjunk-delete: clearjunk (delete x al) = delete x (clearjunk al)
  by (induct al rule: clearjunk.induct) (auto simp add: delete-idem delete-twist)
```

3.2 *clearjunk*

```
lemma insert-fst-filter:
  insert a(fst ‘ {x ∈ set ps. fst x ≠ a}) = insert a (fst ‘ set ps)
  by (induct ps) auto

lemma dom-clearjunk: fst ‘ set (clearjunk al) = fst ‘ set al
  by (induct al rule: clearjunk.induct) (simp-all add: insert-fst-filter delete-eq)

lemma notin-filter-fst: a ∉ fst ‘ {x ∈ set ps. fst x ≠ a}
  by (induct ps) auto

lemma distinct-clearjunk [simp]: distinct (map fst (clearjunk al))
  by (induct al rule: clearjunk.induct)
     (simp-all add: dom-clearjunk notin-filter-fst delete-eq)

lemma map-of-filter: k ≠ a ⟹ map-of [q←ps . fst q ≠ a] k = map-of ps k
  by (induct ps) auto

lemma map-of-clearjunk: map-of (clearjunk al) = map-of al
  apply (rule ext)
  apply (induct al rule: clearjunk.induct)
  apply  simp
  apply (simp add: map-of-filter)
  done

lemma length-clearjunk: length (clearjunk al) ≤ length al
proof (induct al rule: clearjunk.induct [case-names Nil Cons])
  case Nil thus ?case by simp
next
  case (Cons p ps)
```

from *Cons* **have** *length* (*clearjunk* [*q*←*ps* . *fst q* ≠ *fst p*]) ≤ *length* [*q*←*ps* . *fst q* ≠ *fst p*]
by (*simp add*: *delete-eq*)
also have ... ≤ *length ps*
by *simp*
finally show *?case*
by (*simp add*: *delete-eq*)
qed

lemma *notin-fst-filter*: *a* ∉ *fst* ‘ *set ps* ⟹ [*q*←*ps* . *fst q* ≠ *a*] = *ps*
by (*induct ps*) *auto*

lemma *distinct-clearjunk-id* [*simp*]: *distinct* (*map fst al*) ⟹ *clearjunk al* = *al*
by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *notin-fst-filter*)

lemma *clearjunk-idem*: *clearjunk* (*clearjunk al*) = *clearjunk al*
by *simp*

3.3 *dom* and *ran*

lemma *dom-map-of'*: *fst* ‘ *set al* = *dom* (*map-of al*)
by (*induct al*) *auto*

lemmas *dom-map-of* = *dom-map-of'* [*symmetric*]

lemma *ran-clearjunk*: *ran* (*map-of* (*clearjunk al*)) = *ran* (*map-of al*)
by (*simp add*: *map-of-clearjunk*)

lemma *ran-distinct*:
assumes *dist*: *distinct* (*map fst al*)
shows *ran* (*map-of al*) = *snd* ‘ *set al*
using *dist*
proof (*induct al*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons a al*)
hence *hyp*: *snd* ‘ *set al* = *ran* (*map-of al*)
by *simp*

have *ran* (*map-of* (*a* # *al*)) = {*snd a*} ∪ *ran* (*map-of al*)
proof
show *ran* (*map-of* (*a* # *al*)) ⊆ {*snd a*} ∪ *ran* (*map-of al*)
proof
fix *v*
assume *v* ∈ *ran* (*map-of* (*a*#*al*))
then obtain *x* **where** *map-of* (*a*#*al*) *x* = *Some v*
by (*auto simp add*: *ran-def*)
then show *v* ∈ {*snd a*} ∪ *ran* (*map-of al*)

```
      by (auto split: split-if-asm simp add: ran-def)
    qed
  next
    show {snd a} ∪ ran (map-of al) ⊆ ran (map-of (a # al))
    proof
      fix v
      assume v-in: v ∈ {snd a} ∪ ran (map-of al)
      show v ∈ ran (map-of (a#al))
      proof (cases v=snd a)
        case True
        with v-in show ?thesis
          by (auto simp add: ran-def)
      next
        case False
        with v-in have v ∈ ran (map-of al) by auto
        then obtain x where al-x: map-of al x = Some v
          by (auto simp add: ran-def)
        from map-of-SomeD [OF this]
        have x ∈ fst ` set al
          by (force simp add: image-def)
        with Cons.prems have x≠fst a
          by − (rule ccontr,simp)
        with al-x
        show ?thesis
          by (auto simp add: ran-def)
      qed
    qed
  qed
  with hyp show ?case
    by (simp only:) auto
qed

lemma ran-map-of: ran (map-of al) = snd ` set (clearjunk al)
proof −
  have ran (map-of al) = ran (map-of (clearjunk al))
    by (simp add: ran-clearjunk)
  also have ... = snd ` set (clearjunk al)
    by (simp add: ran-distinct)
  finally show ?thesis .
qed
```

3.4 *update*

```
lemma update-conv: map-of (update k v al) k' = ((map-of al)(k↦v)) k'
  by (induct al) auto

lemma update-conv': map-of (update k v al)  = ((map-of al)(k↦v))
  by (rule ext) (rule update-conv)
```

lemma *dom-update*: *fst ‘ set* (*update k v al*) = {*k*} ∪ *fst ‘ set al*
by (*induct al*) *auto*

lemma *distinct-update*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*update k v al*))
using *assms*
proof (*induct al*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a al*)
from *Cons.prems* **obtain**
a-notin-al: *fst a* ∉ *fst ‘ set al* **and**
dist-al: *distinct* (*map fst al*)
by *auto*
show *?case*
proof (*cases fst a* = *k*)
case *True*
from *True dist-al a-notin-al* **show** *?thesis* **by** *simp*
next
case *False*
from *dist-al*
have *distinct* (*map fst* (*update k v al*))
by (*rule Cons.hyps*)
with *False a-notin-al* **show** *?thesis* **by** (*simp add*: *dom-update*)
qed
qed

lemma *update-filter*:
a≠*k* ⟹ *update k v* [*q*←*ps* . *fst q* ≠ *a*] = [*q*←*update k v ps* . *fst q* ≠ *a*]
by (*induct ps*) *auto*

lemma *clearjunk-update*: *clearjunk* (*update k v al*) = *update k v* (*clearjunk al*)
by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *update-filter delete-eq*)

lemma *update-triv*: *map-of al k* = *Some v* ⟹ *update k v al* = *al*
by (*induct al*) *auto*

lemma *update-nonempty* [*simp*]: *update k v al* ≠ []
by (*induct al*) *auto*

lemma *update-eqD*: *update k v al* = *update k v′ al′* ⟹ *v*=*v′*
proof (*induct al arbitrary*: *al′*)
case *Nil* **thus** *?case*
by (*cases al′*) (*auto split*: *split-if-asm*)
next
case *Cons* **thus** *?case*
by (*cases al′*) (*auto split*: *split-if-asm*)
qed

lemma *update-last* [*simp*]: *update k v* (*update k v′ al*) = *update k v al*
by (*induct al*) *auto*

Note that the lists are not necessarily the same: *update k v* (*update k′ v′* []) = [(*k′*, *v′*), (*k*, *v*)] and *update k′ v′* (*update k v* []) = [(*k*, *v*), (*k′*, *v′*)].

lemma *update-swap*: *k*≠*k′*
⟹ *map-of* (*update k v* (*update k′ v′ al*)) = *map-of* (*update k′ v′* (*update k v al*))
by (*auto simp add*: *update-conv′ intro*: *ext*)

lemma *update-Some-unfold*:
(*map-of* (*update k v al*) *x* = *Some y*) =
(*x* = *k* ∧ *v* = *y* ∨ *x* ≠ *k* ∧ *map-of al x* = *Some y*)
by (*simp add*: *update-conv′ map-upd-Some-unfold*)

lemma *image-update*[*simp*]: *x* ∉ *A* ⟹ *map-of* (*update x y al*) ‘ *A* = *map-of al* ‘ *A*
by (*simp add*: *update-conv′ image-map-upd*)

3.5 *updates*

lemma *updates-conv*: *map-of* (*updates ks vs al*) *k* = ((*map-of al*)(*ks*[↦]*vs*)) *k*
proof (*induct ks arbitrary*: *vs al*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons k ks*)
show *?case*
proof (*cases vs*)
case *Nil*
with *Cons* **show** *?thesis* **by** *simp*
next
case (*Cons k ks′*)
with *Cons.hyps* **show** *?thesis*
by (*simp add*: *update-conv fun-upd-def*)
qed
qed

lemma *updates-conv′*: *map-of* (*updates ks vs al*) = ((*map-of al*)(*ks*[↦]*vs*))
by (*rule ext*) (*rule updates-conv*)

lemma *distinct-updates*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*updates ks vs al*))
using *assms*
by (*induct ks arbitrary*: *vs al*)
(*auto simp add*: *distinct-update split*: *list.splits*)

lemma *clearjunk-updates*:
clearjunk (*updates ks vs al*) = *updates ks vs* (*clearjunk al*)
by (*induct ks arbitrary*: *vs al*) (*auto simp add*: *clearjunk-update split*: *list.splits*)

lemma *updates-empty*[*simp*]: *updates vs* [] *al* = *al*
by (*induct vs*) *auto*

lemma *updates-Cons*: *updates* (*k*#*ks*) (*v*#*vs*) *al* = *updates ks vs* (*update k v al*)
by *simp*

lemma *updates-append1*[*simp*]: *size ks* < *size vs* ⟹
updates (*ks*@[*k*]) *vs al* = *update k* (*vs*!*size ks*) (*updates ks vs al*)
by (*induct ks arbitrary*: *vs al*) (*auto split*: *list.splits*)

lemma *updates-list-update-drop*[*simp*]:
⟦*size ks* ≤ *i*; *i* < *size vs*⟧
⟹ *updates ks* (*vs*[*i*:=*v*]) *al* = *updates ks vs al*
by (*induct ks arbitrary*: *al vs i*) (*auto split*:*list.splits nat.splits*)

lemma *update-updates-conv-if*:
map-of (*updates xs ys* (*update x y al*)) =
map-of (*if x* ∈ *set*(*take* (*length ys*) *xs*) *then updates xs ys al*
else (*update x y* (*updates xs ys al*)))
by (*simp add*: *updates-conv' update-conv' map-upd-upds-conv-if*)

lemma *updates-twist* [*simp*]:
k ∉ *set ks* ⟹
map-of (*updates ks vs* (*update k v al*)) = *map-of* (*update k v* (*updates ks vs al*))
by (*simp add*: *updates-conv' update-conv' map-upds-twist*)

lemma *updates-apply-notin*[*simp*]:
k ∉ *set ks* ==> *map-of* (*updates ks vs al*) *k* = *map-of al k*
by (*simp add*: *updates-conv*)

lemma *updates-append-drop*[*simp*]:
size xs = *size ys* ⟹ *updates* (*xs*@*zs*) *ys al* = *updates xs ys al*
by (*induct xs arbitrary*: *ys al*) (*auto split*: *list.splits*)

lemma *updates-append2-drop*[*simp*]:
size xs = *size ys* ⟹ *updates xs* (*ys*@*zs*) *al* = *updates xs ys al*
by (*induct xs arbitrary*: *ys al*) (*auto split*: *list.splits*)

3.6 *map-ran*

lemma *map-ran-conv*: *map-of* (*map-ran f al*) *k* = *option-map* (*f k*) (*map-of al k*)
by (*induct al*) *auto*

lemma *dom-map-ran*: *fst* ‘ *set* (*map-ran f al*) = *fst* ‘ *set al*
by (*induct al*) *auto*

lemma *distinct-map-ran*: *distinct* (*map fst al*) $\Longrightarrow$ *distinct* (*map fst* (*map-ran f al*))
by (*induct al*) (*auto simp add*: *dom-map-ran*)

lemma *map-ran-filter*: *map-ran f* [*p*←*ps*. *fst p* $\neq$ *a*] = [*p*←*map-ran f ps*. *fst p* $\neq$ *a*]
by (*induct ps*) *auto*

lemma *clearjunk-map-ran*: *clearjunk* (*map-ran f al*) = *map-ran f* (*clearjunk al*)
by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *delete-eq map-ran-filter*)

3.7 *merge*

lemma *dom-merge*: *fst* ' *set* (*merge xs ys*) = *fst* ' *set xs* $\cup$ *fst* ' *set ys*
by (*induct ys arbitrary*: *xs*) (*auto simp add*: *dom-update*)

lemma *distinct-merge*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*merge xs ys*))
using *assms*
by (*induct ys arbitrary*: *xs*) (*auto simp add*: *dom-merge distinct-update*)

lemma *clearjunk-merge*:
clearjunk (*merge xs ys*) = *merge* (*clearjunk xs*) *ys*
by (*induct ys*) (*auto simp add*: *clearjunk-update*)

lemma *merge-conv*: *map-of* (*merge xs ys*) *k* = (*map-of xs* ++ *map-of ys*) *k*
proof (*induct ys*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons y ys*)
show *?case*
proof (*cases k* = *fst y*)
case *True*
from *True* **show** *?thesis*
by (*simp add*: *update-conv*)
next
case *False*
from *False* **show** *?thesis*
by (*auto simp add*: *update-conv Cons.hyps map-add-def*)
qed
qed

lemma *merge-conv'*: *map-of* (*merge xs ys*) = (*map-of xs* ++ *map-of ys*)
by (*rule ext*) (*rule merge-conv*)

lemma *merge-emty*: *map-of* (*merge* [] *ys*) = *map-of ys*
by (*simp add*: *merge-conv'*)

lemma *merge-assoc*[*simp*]: *map-of* (*merge m1* (*merge m2 m3*)) =
map-of (*merge* (*merge m1 m2*) *m3*)
by (*simp add*: *merge-conv'*)

lemma *merge-Some-iff*:
(*map-of* (*merge m n*) *k* = *Some x*) =
(*map-of n k* = *Some x* ∨ *map-of n k* = *None* ∧ *map-of m k* = *Some x*)
by (*simp add*: *merge-conv' map-add-Some-iff*)

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

lemma *merge-find-right*[*simp*]: *map-of n k* = *Some v* ⟹ *map-of* (*merge m n*) *k* = *Some v*
by (*simp add*: *merge-conv'*)

lemma *merge-None* [*iff*]:
(*map-of* (*merge m n*) *k* = *None*) = (*map-of n k* = *None* ∧ *map-of m k* = *None*)
by (*simp add*: *merge-conv'*)

lemma *merge-upd*[*simp*]:
map-of (*merge m* (*update k v n*)) = *map-of* (*update k v* (*merge m n*))
by (*simp add*: *update-conv' merge-conv'*)

lemma *merge-updatess*[*simp*]:
map-of (*merge m* (*updates xs ys n*)) = *map-of* (*updates xs ys* (*merge m n*))
by (*simp add*: *updates-conv' merge-conv'*)

lemma *merge-append*: *map-of* (*xs*@*ys*) = *map-of* (*merge ys xs*)
by (*simp add*: *merge-conv'*)

3.8 *compose*

lemma *compose-first-None* [*simp*]:
assumes *map-of xs k* = *None*
shows *map-of* (*compose xs ys*) *k* = *None*
using *assms* **by** (*induct xs ys rule*: *compose.induct*)
(*auto split*: *option.splits split-if-asm*)

lemma *compose-conv*:
shows *map-of* (*compose xs ys*) *k* = (*map-of ys* ∘$_m$ *map-of xs*) *k*
proof (*induct xs ys rule*: *compose.induct*)
case *1* **then show** *?case* **by** *simp*
next
case (*2 x xs ys*) **show** *?case*
proof (*cases map-of ys* (*snd x*))
case *None* **with** *2*
have *hyp*: *map-of* (*compose* (*delete* (*fst x*) *xs*) *ys*) *k* =

```
          (map-of ys ∘m map-of (delete (fst x) xs)) k
    by simp
  show ?thesis
  proof (cases fst x = k)
    case True
    from True delete-notin-dom [of k xs]
    have map-of (delete (fst x) xs) k = None
      by (simp add: map-of-eq-None-iff)
    with hyp show ?thesis
      using True None
      by simp
  next
    case False
    from False have map-of (delete (fst x) xs) k = map-of xs k
      by simp
    with hyp show ?thesis
      using False None
      by (simp add: map-comp-def)
  qed
 next
  case (Some v)
  with 2
  have map-of (compose xs ys) k = (map-of ys ∘m map-of xs) k
    by simp
  with Some show ?thesis
    by (auto simp add: map-comp-def)
 qed
qed

lemma compose-conv':
 shows map-of (compose xs ys) = (map-of ys ∘m map-of xs)
 by (rule ext) (rule compose-conv)

lemma compose-first-Some [simp]:
 assumes map-of xs k = Some v
 shows map-of (compose xs ys) k = map-of ys v
using assms by (simp add: compose-conv)

lemma dom-compose: fst ‘ set (compose xs ys) ⊆ fst ‘ set xs
proof (induct xs ys rule: compose.induct)
 case 1 thus ?case by simp
next
 case (2 x xs ys)
 show ?case
 proof (cases map-of ys (snd x))
   case None
   with 2.hyps
   have fst ‘ set (compose (delete (fst x) xs) ys) ⊆ fst ‘ set (delete (fst x) xs)
     by simp
```

```
    also
    have ... ⊆ fst ` set xs
      by (rule dom-delete-subset)
    finally show ?thesis
      using None
      by auto
  next
    case (Some v)
    with 2.hyps
    have fst ` set (compose xs ys) ⊆ fst ` set xs
      by simp
    with Some show ?thesis
      by auto
  qed
qed

lemma distinct-compose:
 assumes distinct (map fst xs)
 shows distinct (map fst (compose xs ys))
using assms
proof (induct xs ys rule: compose.induct)
  case 1 thus ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2 show ?thesis by simp
  next
    case (Some v)
    with 2 dom-compose [of xs ys] show ?thesis
      by (auto)
  qed
qed

lemma compose-delete-twist: (compose (delete k xs) ys) = delete k (compose xs
ys)
proof (induct xs ys rule: compose.induct)
  case 1 thus ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2 have
      hyp: compose (delete k (delete (fst x) xs)) ys =
           delete k (compose (delete (fst x) xs) ys)
      by simp
    show ?thesis
```

```
    proof (cases fst x = k)
      case True
      with None hyp
      show ?thesis
        by (simp add: delete-idem)
    next
      case False
      from None False hyp
      show ?thesis
        by (simp add: delete-twist)
    qed
  next
    case (Some v)
    with 2 have hyp: compose (delete k xs) ys = delete k (compose xs ys) by simp
    with Some show ?thesis
      by simp
  qed
qed

lemma compose-clearjunk: compose xs (clearjunk ys) = compose xs ys
  by (induct xs ys rule: compose.induct)
     (auto simp add: map-of-clearjunk split: option.splits)

lemma clearjunk-compose: clearjunk (compose xs ys) = compose (clearjunk xs) ys
  by (induct xs rule: clearjunk.induct)
     (auto split: option.splits simp add: clearjunk-delete delete-idem
           compose-delete-twist)

lemma compose-empty [simp]:
 compose xs [] = []
  by (induct xs) (auto simp add: compose-delete-twist)

lemma compose-Some-iff:
  (map-of (compose xs ys) k = Some v) =
     (∃k'. map-of xs k = Some k' ∧ map-of ys k' = Some v)
  by (simp add: compose-conv map-comp-Some-iff)

lemma map-comp-None-iff:
  (map-of (compose xs ys) k = None) =
    (map-of xs k = None ∨ (∃k'. map-of xs k = Some k' ∧ map-of ys k' = None))

  by (simp add: compose-conv map-comp-None-iff)
```

3.9 *restrict*

```
lemma restrict-eq:
  restrict A = filter (λp. fst p ∈ A)
proof
  fix xs
```

show *restrict A xs = filter* (λ*p*. *fst p* ∈ *A*) *xs*
by (*induct xs*) *auto*
qed

lemma *distinct-restr*: *distinct* (*map fst al*) ⟹ *distinct* (*map fst* (*restrict A al*))
by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k*
apply (*induct al*)
apply (*simp add*: *restrict-eq*)
apply (*cases k*∈*A*)
apply (*auto simp add*: *restrict-eq*)
done

lemma *restr-conv′*: *map-of* (*restrict A al*) = ((*map-of al*)|‘ *A*)
by (*rule ext*) (*rule restr-conv*)

lemma *restr-empty* [*simp*]:
restrict {} *al* = []
restrict A [] = []
by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-in* [*simp*]: *x* ∈ *A* ⟹ *map-of* (*restrict A al*) *x* = *map-of al x*
by (*simp add*: *restr-conv′*)

lemma *restr-out* [*simp*]: *x* ∉ *A* ⟹ *map-of* (*restrict A al*) *x* = *None*
by (*simp add*: *restr-conv′*)

lemma *dom-restr* [*simp*]: *fst* ‘ *set* (*restrict A al*) = *fst* ‘ *set al* ∩ *A*
by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-upd-same* [*simp*]: *restrict* (−{*x*}) (*update x y al*) = *restrict* (−{*x*}) *al*
by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-restr* [*simp*]: *restrict A* (*restrict B al*) = *restrict* (*A*∩*B*) *al*
by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-update*[*simp*]:
map-of (*restrict D* (*update x y al*)) =
map-of ((*if x* ∈ *D then* (*update x y* (*restrict* (*D*−{*x*}) *al*)) *else restrict D al*))
by (*simp add*: *restr-conv′ update-conv′*)

lemma *restr-delete* [*simp*]:
(*delete x* (*restrict D al*)) =
(*if x*∈ *D then restrict* (*D* − {*x*}) *al else restrict D al*)
proof (*induct al*)
case *Nil* **thus** *?case* **by** *simp*
next

```
  case (Cons a al)
  show ?case
  proof (cases x ∈ D)
    case True
    note x-D = this
    with Cons have hyp: delete x (restrict D al) = restrict (D − {x}) al
      by simp
    show ?thesis
    proof (cases fst a = x)
      case True
      from Cons.hyps
      show ?thesis
        using x-D True
        by simp
    next
      case False
      note not-fst-a-x = this
      show ?thesis
      proof (cases fst a ∈ D)
        case True
        with not-fst-a-x
        have delete x (restrict D (a#al)) = a#(delete x (restrict D al))
          by (cases a) (simp add: restrict-eq)
        also from not-fst-a-x True hyp have ... = restrict (D − {x}) (a # al)
          by (cases a) (simp add: restrict-eq)
        finally show ?thesis
          using x-D by simp
      next
        case False
        hence delete x (restrict D (a#al)) = delete x (restrict D al)
          by (cases a) (simp add: restrict-eq)
        moreover from False not-fst-a-x
        have restrict (D − {x}) (a # al) = restrict (D − {x}) al
          by (cases a) (simp add: restrict-eq)
        ultimately
        show ?thesis using x-D hyp by simp
      qed
    qed
  next
    case False
    from False Cons show ?thesis
      by simp
  qed
qed

lemma update-restr:
 map-of (update x y (restrict D al)) = map-of (update x y (restrict (D−{x}) al))
  by (simp add: update-conv' restr-conv') (rule fun-upd-restrict)
```

lemma *upate-restr-conv* [*simp*]:
$x \in D \Longrightarrow$
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D*−{*x*}) *al*))
by (*simp add*: *update-conv′ restr-conv′*)

lemma *restr-updates* [*simp*]:
⟦ *length xs* = *length ys*; *set xs* ⊆ *D* ⟧
⟹ *map-of* (*restrict D* (*updates xs ys al*)) =
map-of (*updates xs ys* (*restrict* (*D* − *set xs*) *al*))
by (*simp add*: *updates-conv′ restr-conv′*)

lemma *restr-delete-twist*: (*restrict A* (*delete a ps*)) = *delete a* (*restrict A ps*)
by (*induct ps*) *auto*

lemma *clearjunk-restrict*:
clearjunk (*restrict A al*) = *restrict A* (*clearjunk al*)
by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *restr-delete-twist*)

end

4 SetsAndFunctions: Operations on sets and functions

theory *SetsAndFunctions*
imports *ATP-Linkup*
begin

This library lifts operations like addition and muliplication to sets and functions of appropriate types. It was designed to support asymptotic calculations. See the comments at the top of theory *BigO*.

4.1 Basic definitions

definition
set-plus :: ('*a*::*plus*) *set* => '*a set* => '*a set* (**infixl** ⊕ *65*) **where**
A ⊕ *B* == {*c*. *EX a*:*A*. *EX b*:*B*. *c* = *a* + *b*}

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

definition
func-plus: *f* + *g* == (%*x*. *f x* + *g x*)

instance ..

end

definition
set-times :: ('a::*times*) *set* => 'a *set* => 'a *set* (**infixl** ⊗ *70*) **where**
A ⊗ *B* == {*c*. *EX* *a*:*A*. *EX* *b*:*B*. *c* = *a* * *b*}

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

definition
func-times: *f* * *g* == (%*x*. *f* *x* * *g* *x*)

instance ..

end

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

definition
func-zero: *0*::(('a::*type*) => ('b::*zero*)) == %*x*. *0*

instance ..

end

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

definition
func-one: *1*::(('a::*type*) => ('b::*one*)) == %*x*. *1*

instance ..

end

definition
elt-set-plus :: 'a::*plus* => 'a *set* => 'a *set* (**infixl** +*o* *70*) **where**
a +*o* *B* = {*c*. *EX* *b*:*B*. *c* = *a* + *b*}

definition
elt-set-times :: 'a::*times* => 'a *set* => 'a *set* (**infixl** *o* *80*) **where**
a *o* *B* = {*c*. *EX* *b*:*B*. *c* = *a* * *b*}

abbreviation (*input*)
elt-set-eq :: 'a => 'a *set* => *bool* (**infix** =*o* *50*) **where**
x =*o* *A* == *x* : *A*

instance *fun* :: (*type*,*semigroup-add*)*semigroup-add*
by *default* (*auto simp add*: *func-plus add-assoc*)

instance *fun* :: (*type*,*comm-monoid-add*)*comm-monoid-add*
by *default* (*auto simp add*: *func-zero func-plus add-ac*)

instance *fun* :: (*type*,*ab-group-add*)*ab-group-add*
apply *default*
apply (*simp add*: *fun-Compl-def func-plus func-zero*)
apply (*simp add*: *fun-Compl-def func-plus fun-diff-def diff-minus*)
done

instance *fun* :: (*type*,*semigroup-mult*)*semigroup-mult*
apply *default*
apply (*auto simp add*: *func-times mult-assoc*)
done

instance *fun* :: (*type*,*comm-monoid-mult*)*comm-monoid-mult*
apply *default*
apply (*auto simp add*: *func-one func-times mult-ac*)
done

instance *fun* :: (*type*,*comm-ring-1*)*comm-ring-1*
apply *default*
apply (*auto simp add*: *func-plus func-times fun-Compl-def fun-diff-def ext*
func-one func-zero ring-simps)
apply (*drule fun-cong*)
apply *simp*
done

interpretation *set-semigroup-add*: *semigroup-add* [*op* ⊕ :: ('*a*::*semigroup-add*) *set*
=> '*a set* => '*a set*]
apply *default*
apply (*unfold set-plus-def*)
apply (*force simp add*: *add-assoc*)
done

interpretation *set-semigroup-mult*: *semigroup-mult* [*op* ⊗ :: ('*a*::*semigroup-mult*)
set => '*a set* => '*a set*]
apply *default*
apply (*unfold set-times-def*)
apply (*force simp add*: *mult-assoc*)
done

interpretation *set-comm-monoid-add*: *comm-monoid-add* [{*0*} *op* ⊕ :: ('*a*::*comm-monoid-add*)
set => '*a set* => '*a set*]
apply *default*
apply (*unfold set-plus-def*)
apply (*force simp add*: *add-ac*)
apply *force*
done

interpretation *set-comm-monoid-mult*: *comm-monoid-mult* [{*1*} *op* ⊗ :: ('*a*::*comm-monoid-mult*) *set* => '*a set* => '*a set*]
apply *default*
apply (*unfold set-times-def*)
apply (*force simp add*: *mult-ac*)
apply *force*
done

4.2 Basic properties

lemma *set-plus-intro* [*intro*]: *a* : *C* ==> *b* : *D* ==> *a* + *b* : *C* ⊕ *D*
by (*auto simp add*: *set-plus-def*)

lemma *set-plus-intro2* [*intro*]: *b* : *C* ==> *a* + *b* : *a* +*o* *C*
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-rearrange*: ((*a*::'*a*::*comm-monoid-add*) +*o* *C*) ⊕
(*b* +*o* *D*) = (*a* + *b*) +*o* (*C* ⊕ *D*)
apply (*auto simp add*: *elt-set-plus-def set-plus-def add-ac*)
apply (*rule-tac x* = *ba* + *bb* **in** *exI*)
apply (*auto simp add*: *add-ac*)
apply (*rule-tac x* = *aa* + *a* **in** *exI*)
apply (*auto simp add*: *add-ac*)
done

lemma *set-plus-rearrange2*: (*a*::'*a*::*semigroup-add*) +*o* (*b* +*o* *C*) =
(*a* + *b*) +*o* *C*
by (*auto simp add*: *elt-set-plus-def add-assoc*)

lemma *set-plus-rearrange3*: ((*a*::'*a*::*semigroup-add*) +*o* *B*) ⊕ *C* =
a +*o* (*B* ⊕ *C*)
apply (*auto simp add*: *elt-set-plus-def set-plus-def*)
apply (*blast intro*: *add-ac*)
apply (*rule-tac x* = *a* + *aa* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x* = *aa* **in** *bexI*)
apply *auto*
apply (*rule-tac x* = *ba* **in** *bexI*)
apply (*auto simp add*: *add-ac*)
done

theorem *set-plus-rearrange4*: *C* ⊕ ((*a*::'*a*::*comm-monoid-add*) +*o* *D*) =
a +*o* (*C* ⊕ *D*)
apply (*auto intro*!: *subsetI simp add*: *elt-set-plus-def set-plus-def add-ac*)
apply (*rule-tac x* = *aa* + *ba* **in** *exI*)
apply (*auto simp add*: *add-ac*)
done

theorems *set-plus-rearranges = set-plus-rearrange set-plus-rearrange2 set-plus-rearrange3 set-plus-rearrange4*

lemma *set-plus-mono* [*intro*!]: $C <= D ==> a +o C <= a +o D$
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-mono2* [*intro*]: $(C::('a::plus)\ set) <= D ==> E <= F ==> C \oplus E <= D \oplus F$
by (*auto simp add*: *set-plus-def*)

lemma *set-plus-mono3* [*intro*]: $a : C ==> a +o D <= C \oplus D$
by (*auto simp add*: *elt-set-plus-def set-plus-def*)

lemma *set-plus-mono4* [*intro*]: $(a::'a::comm\text{-}monoid\text{-}add) : C ==> a +o D <= D \oplus C$
by (*auto simp add*: *elt-set-plus-def set-plus-def add-ac*)

lemma *set-plus-mono5*: $a:C ==> B <= D ==> a +o B <= C \oplus D$
apply (*subgoal-tac* $a +o B <= a +o D$)
apply (*erule order-trans*)
apply (*erule set-plus-mono3*)
apply (*erule set-plus-mono*)
done

lemma *set-plus-mono-b*: $C <= D ==> x : a +o C ==> x : a +o D$
apply (*frule set-plus-mono*)
apply *auto*
done

lemma *set-plus-mono2-b*: $C <= D ==> E <= F ==> x : C \oplus E ==> x : D \oplus F$
apply (*frule set-plus-mono2*)
prefer *2*
apply *force*
apply *assumption*
done

lemma *set-plus-mono3-b*: $a : C ==> x : a +o D ==> x : C \oplus D$
apply (*frule set-plus-mono3*)
apply *auto*
done

lemma *set-plus-mono4-b*: $(a::'a::comm\text{-}monoid\text{-}add) : C ==> x : a +o D ==> x : D \oplus C$
apply (*frule set-plus-mono4*)
apply *auto*
done

lemma *set-zero-plus* [*simp*]: $(0::'a::comm\text{-}monoid\text{-}add) +o\ C = C$
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-zero-plus2*: $(0::'a::comm\text{-}monoid\text{-}add) : A ==> B <= A \oplus B$
apply (*auto intro*!: *subsetI simp add*: *set-plus-def*)
apply (*rule-tac* $x = 0$ **in** *bexI*)
apply (*rule-tac* $x = x$ **in** *bexI*)
apply (*auto simp add*: *add-ac*)
done

lemma *set-plus-imp-minus*: $(a::'a::ab\text{-}group\text{-}add) : b +o\ C ==> (a - b) : C$
by (*auto simp add*: *elt-set-plus-def add-ac diff-minus*)

lemma *set-minus-imp-plus*: $(a::'a::ab\text{-}group\text{-}add) - b : C ==> a : b +o\ C$
apply (*auto simp add*: *elt-set-plus-def add-ac diff-minus*)
apply (*subgoal-tac* $a = (a + - b) + b$)
apply (*rule bexI*, *assumption*, *assumption*)
apply (*auto simp add*: *add-ac*)
done

lemma *set-minus-plus*: $((a::'a::ab\text{-}group\text{-}add) - b : C) = (a : b +o\ C)$
by (*rule iffI*, *rule set-minus-imp-plus*, *assumption*, *rule set-plus-imp-minus*, *assumption*)

lemma *set-times-intro* [*intro*]: $a : C ==> b : D ==> a * b : C \otimes D$
by (*auto simp add*: *set-times-def*)

lemma *set-times-intro2* [*intro*!]: $b : C ==> a * b : a *o\ C$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-rearrange*: $((a::'a::comm\text{-}monoid\text{-}mult) *o\ C) \otimes (b *o\ D) = (a * b) *o\ (C \otimes D)$
apply (*auto simp add*: *elt-set-times-def set-times-def*)
apply (*rule-tac* $x = ba * bb$ **in** *exI*)
apply (*auto simp add*: *mult-ac*)
apply (*rule-tac* $x = aa * a$ **in** *exI*)
apply (*auto simp add*: *mult-ac*)
done

lemma *set-times-rearrange2*: $(a::'a::semigroup\text{-}mult) *o\ (b *o\ C) = (a * b) *o\ C$
by (*auto simp add*: *elt-set-times-def mult-assoc*)

lemma *set-times-rearrange3*: $((a::'a::semigroup\text{-}mult) *o\ B) \otimes C = a *o\ (B \otimes C)$
apply (*auto simp add*: *elt-set-times-def set-times-def*)
apply (*blast intro*: *mult-ac*)
apply (*rule-tac* $x = a * aa$ **in** *exI*)
apply (*rule conjI*)

```
 apply (rule-tac x = aa in bexI)
  apply auto
 apply (rule-tac x = ba in bexI)
  apply (auto simp add: mult-ac)
 done

theorem set-times-rearrange4: C ⊗ ((a::'a::comm-monoid-mult) *o D) =
    a *o (C ⊗ D)
 apply (auto intro!: subsetI simp add: elt-set-times-def set-times-def
   mult-ac)
  apply (rule-tac x = aa * ba in exI)
  apply (auto simp add: mult-ac)
 done

theorems set-times-rearranges = set-times-rearrange set-times-rearrange2
  set-times-rearrange3 set-times-rearrange4

lemma set-times-mono [intro]: C <= D ==> a *o C <= a *o D
 by (auto simp add: elt-set-times-def)

lemma set-times-mono2 [intro]: (C::('a::times) set) <= D ==> E <= F ==>
    C ⊗ E <= D ⊗ F
 by (auto simp add: set-times-def)

lemma set-times-mono3 [intro]: a : C ==> a *o D <= C ⊗ D
 by (auto simp add: elt-set-times-def set-times-def)

lemma set-times-mono4 [intro]: (a::'a::comm-monoid-mult) : C ==>
    a *o D <= D ⊗ C
 by (auto simp add: elt-set-times-def set-times-def mult-ac)

lemma set-times-mono5: a:C ==> B <= D ==> a *o B <= C ⊗ D
 apply (subgoal-tac a *o B <= a *o D)
  apply (erule order-trans)
  apply (erule set-times-mono3)
 apply (erule set-times-mono)
 done

lemma set-times-mono-b: C <= D ==> x : a *o C
    ==> x : a *o D
 apply (frule set-times-mono)
 apply auto
 done

lemma set-times-mono2-b: C <= D ==> E <= F ==> x : C ⊗ E ==>
    x : D ⊗ F
 apply (frule set-times-mono2)
  prefer 2
  apply force
```

apply *assumption*
done

lemma *set-times-mono3-b*: $a : C ==> x : a *o D ==> x : C \otimes D$
apply (*frule set-times-mono3*)
apply *auto*
done

lemma *set-times-mono4-b*: (a::'a::*comm-monoid-mult*) : $C ==>$
$x : a *o D ==> x : D \otimes C$
apply (*frule set-times-mono4*)
apply *auto*
done

lemma *set-one-times* [*simp*]: (*1*::'a::*comm-monoid-mult*) $*o$ $C = C$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-plus-distrib*: (a::'a::*semiring*) $*o$ $(b +o C)=$
$(a * b) +o (a *o C)$
by (*auto simp add*: *elt-set-plus-def elt-set-times-def ring-distribs*)

lemma *set-times-plus-distrib2*: (a::'a::*semiring*) $*o$ $(B \oplus C) =$
$(a *o B) \oplus (a *o C)$
apply (*auto simp add*: *set-plus-def elt-set-times-def ring-distribs*)
apply *blast*
apply (*rule-tac* $x = b + bb$ **in** *exI*)
apply (*auto simp add*: *ring-distribs*)
done

lemma *set-times-plus-distrib3*: ((a::'a::*semiring*) $+o$ $C) \otimes D <=$
$a *o D \oplus C \otimes D$
apply (*auto intro*!: *subsetI simp add*:
elt-set-plus-def elt-set-times-def set-times-def
set-plus-def ring-distribs)
apply *auto*
done

theorems *set-times-plus-distribs* =
set-times-plus-distrib
set-times-plus-distrib2

lemma *set-neg-intro*: (a::'a::*ring-1*) : $(- 1) *o C ==>$
$- a : C$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-neg-intro2*: (a::'a::*ring-1*) : $C ==>$
$- a : (- 1) *o C$
by (*auto simp add*: *elt-set-times-def*)

end

5 BigO: Big O notation

theory *BigO*
imports *Main SetsAndFunctions*
begin

This library is designed to support asymptotic "big O" calculations, i.e. reasoning with expressions of the form $f = O(g)$ and $f = g + O(h)$. An earlier version of this library is described in detail in [2].

The main changes in this version are as follows:

- We have eliminated the O operator on sets. (Most uses of this seem to be inessential.)
- We no longer use + as output syntax for $+o$
- Lemmas involving *sumr* have been replaced by more general lemmas involving '*setsum*.
- The library has been expanded, with e.g. support for expressions of the form $f < g + O(h)$.

See `Complex/ex/BigO_Complex.thy` for additional lemmas that require the `HOL-Complex` logic image.

Note also since the Big O library includes rules that demonstrate set inclusion, to use the automated reasoners effectively with the library one should redeclare the theorem *subsetI* as an intro rule, rather than as an *intro*! rule, for example, using **declare** *subsetI* [*del*, *intro*].

5.1 Definitions

definition
bigo :: ('*a* => '*b*::*ordered-idom*) => ('*a* => '*b*) *set* ((*1O'*(-'))) **where**
O(*f*::('*a* => '*b*)) =
{*h*. *EX c*. *ALL x*. *abs* (*h x*) <= *c* * *abs* (*f x*)}

lemma *bigo-pos-const*: (*EX* (*c*::'*a*::*ordered-idom*).
ALL x. (*abs* (*h x*)) <= (*c* * (*abs* (*f x*))))
= (*EX c*. *0* < *c* & (*ALL x*. (*abs*(*h x*)) <= (*c* * (*abs* (*f x*))))))
apply *auto*
apply (*case-tac c* = *0*)
apply *simp*
apply (*rule-tac x* = *1* **in** *exI*)
apply *simp*
apply (*rule-tac x* = *abs c* **in** *exI*)

apply *auto*
apply (*subgoal-tac* $c * abs(f\ x) <= abs\ c * abs\ (f\ x)$)
apply (*erule-tac* $x = x$ **in** *allE*)
apply *force*
apply (*rule mult-right-mono*)
apply (*rule abs-ge-self*)
apply (*rule abs-ge-zero*)
done

lemma *bigo-alt-def*: $O(f) =$
$\{h.\ EX\ c.\ (0 < c\ \&\ (ALL\ x.\ abs\ (h\ x) <= c * abs\ (f\ x)))\}$
by (*auto simp add*: *bigo-def bigo-pos-const*)

lemma *bigo-elt-subset* [*intro*]: $f : O(g) ==> O(f) <= O(g)$
apply (*auto simp add*: *bigo-alt-def*)
apply (*rule-tac* $x = ca * c$ **in** *exI*)
apply (*rule conjI*)
apply (*rule mult-pos-pos*)
apply (*assumption*)+
apply (*rule allI*)
apply (*drule-tac* $x = xa$ **in** *spec*)+
apply (*subgoal-tac* $ca * abs(f\ xa) <= ca * (c * abs(g\ xa))$)
apply (*erule order-trans*)
apply (*simp add*: *mult-ac*)
apply (*rule mult-left-mono*, *assumption*)
apply (*rule order-less-imp-le*, *assumption*)
done

lemma *bigo-refl* [*intro*]: $f : O(f)$
apply(*auto simp add*: *bigo-def*)
apply(*rule-tac* $x = 1$ **in** *exI*)
apply *simp*
done

lemma *bigo-zero*: $0 : O(g)$
apply (*auto simp add*: *bigo-def func-zero*)
apply (*rule-tac* $x = 0$ **in** *exI*)
apply *auto*
done

lemma *bigo-zero2*: $O(\%x.0) = \{\%x.0\}$
apply (*auto simp add*: *bigo-def*)
apply (*rule ext*)
apply *auto*
done

lemma *bigo-plus-self-subset* [*intro*]:
$O(f) \oplus O(f) <= O(f)$
apply (*auto simp add*: *bigo-alt-def set-plus-def*)

apply (*rule-tac* $x = c + ca$ **in** *exI*)
apply *auto*
apply (*simp add*: *ring-distribs func-plus*)
apply (*rule order-trans*)
apply (*rule abs-triangle-ineq*)
apply (*rule add-mono*)
apply *force*
apply *force*
done

lemma *bigo-plus-idemp* [*simp*]: $O(f) \oplus O(f) = O(f)$
apply (*rule equalityI*)
apply (*rule bigo-plus-self-subset*)
apply (*rule set-zero-plus2*)
apply (*rule bigo-zero*)
done

lemma *bigo-plus-subset* [*intro*]: $O(f + g) <= O(f) \oplus O(g)$
apply (*rule subsetI*)
apply (*auto simp add*: *bigo-def bigo-pos-const func-plus set-plus-def*)
apply (*subst bigo-pos-const* [*symmetric*])+
apply (*rule-tac* $x =$
$\%n.$ *if abs* $(g\ n) <= (abs\ (f\ n))$ *then x n else 0* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = c + c$ **in** *exI*)
apply (*clarsimp*)
apply (*auto*)
apply (*subgoal-tac* $c * abs\ (f\ xa + g\ xa) <= (c + c) * abs\ (f\ xa)$)
apply (*erule-tac* $x = xa$ **in** *allE*)
apply (*erule order-trans*)
apply (*simp*)
apply (*subgoal-tac* $c * abs\ (f\ xa + g\ xa) <= c * (abs\ (f\ xa) + abs\ (g\ xa))$)
apply (*erule order-trans*)
apply (*simp add*: *ring-distribs*)
apply (*rule mult-left-mono*)
apply *assumption*
apply (*simp add*: *order-less-le*)
apply (*rule mult-left-mono*)
apply (*simp add*: *abs-triangle-ineq*)
apply (*simp add*: *order-less-le*)
apply (*rule mult-nonneg-nonneg*)
apply (*rule add-nonneg-nonneg*)
apply *auto*
apply (*rule-tac* $x = \%n.$ *if* $(abs\ (f\ n)) <\ abs\ (g\ n)$ *then x n else 0*
in *exI*)
apply (*rule conjI*)
apply (*rule-tac* $x = c + c$ **in** *exI*)
apply *auto*
apply (*subgoal-tac* $c * abs\ (f\ xa + g\ xa) <= (c + c) * abs\ (g\ xa)$)

apply (*erule-tac x = xa* **in** *allE*)
apply (*erule order-trans*)
apply (*simp*)
apply (*subgoal-tac c * abs (f xa + g xa) <= c * (abs (f xa) + abs (g xa))*)
apply (*erule order-trans*)
apply (*simp add*: *ring-distribs*)
apply (*rule mult-left-mono*)
apply (*simp add*: *order-less-le*)
apply (*simp add*: *order-less-le*)
apply (*rule mult-left-mono*)
apply (*rule abs-triangle-ineq*)
apply (*simp add*: *order-less-le*)
apply (*rule mult-nonneg-nonneg*)
apply (*rule add-nonneg-nonneg*)
apply (*erule order-less-imp-le*)+
apply *simp*
apply (*rule ext*)
apply (*auto simp add*: *if-splits linorder-not-le*)
done

lemma *bigo-plus-subset2* [*intro*]: *A <= O(f) ==> B <= O(f) ==> A ⊕ B <= O(f)*
apply (*subgoal-tac A ⊕ B <= O(f) ⊕ O(f)*)
apply (*erule order-trans*)
apply *simp*
apply (*auto del*: *subsetI simp del*: *bigo-plus-idemp*)
done

lemma *bigo-plus-eq*: *ALL x. 0 <= f x ==> ALL x. 0 <= g x ==>*
O(f + g) = O(f) ⊕ O(g)
apply (*rule equalityI*)
apply (*rule bigo-plus-subset*)
apply (*simp add*: *bigo-alt-def set-plus-def func-plus*)
apply *clarify*
apply (*rule-tac x = max c ca* **in** *exI*)
apply (*rule conjI*)
apply (*subgoal-tac c <= max c ca*)
apply (*erule order-less-le-trans*)
apply *assumption*
apply (*rule le-maxI1*)
apply *clarify*
apply (*drule-tac x = xa* **in** *spec*)+
apply (*subgoal-tac 0 <= f xa + g xa*)
apply (*simp add*: *ring-distribs*)
apply (*subgoal-tac abs(a xa + b xa) <= abs(a xa) + abs(b xa)*)
apply (*subgoal-tac abs(a xa) + abs(b xa) <=*
*max c ca * f xa + max c ca * g xa*)
apply (*force*)
apply (*rule add-mono*)

apply (*subgoal-tac c * f xa <= max c ca * f xa*)
apply (*force*)
apply (*rule mult-right-mono*)
apply (*rule le-maxI1*)
apply *assumption*
apply (*subgoal-tac ca * g xa <= max c ca * g xa*)
apply (*force*)
apply (*rule mult-right-mono*)
apply (*rule le-maxI2*)
apply *assumption*
apply (*rule abs-triangle-ineq*)
apply (*rule add-nonneg-nonneg*)
apply *assumption+*
done

lemma *bigo-bounded-alt*: *ALL x. 0 <= f x ==> ALL x. f x <= c * g x ==>*
f : O(g)
apply (*auto simp add*: *bigo-def*)
apply (*rule-tac x = abs c* **in** *exI*)
apply *auto*
apply (*drule-tac x = x* **in** *spec*)+
apply (*simp add*: *abs-mult* [*symmetric*])
done

lemma *bigo-bounded*: *ALL x. 0 <= f x ==> ALL x. f x <= g x ==>*
f : O(g)
apply (*erule bigo-bounded-alt* [*of f 1 g*])
apply *simp*
done

lemma *bigo-bounded2*: *ALL x. lb x <= f x ==> ALL x. f x <= lb x + g x ==>*
f : lb +o O(g)
apply (*rule set-minus-imp-plus*)
apply (*rule bigo-bounded*)
apply (*auto simp add*: *diff-minus fun-Compl-def func-plus*)
apply (*drule-tac x = x* **in** *spec*)+
apply *force*
apply (*drule-tac x = x* **in** *spec*)+
apply *force*
done

lemma *bigo-abs*: *(%x. abs(f x)) =o O(f)*
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac x = 1* **in** *exI*)
apply *auto*
done

lemma *bigo-abs2*: *f =o O(%x. abs(f x))*

apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac x = 1* **in** *exI*)
apply *auto*
done

lemma *bigo-abs3*: $O(f) = O(\%x.\ abs(f\ x))$
apply (*rule equalityI*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-abs2*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-abs*)
done

lemma *bigo-abs4*: $f =o\ g\ +o\ O(h) ==>$
$(\%x.\ abs\ (f\ x)) =o\ (\%x.\ abs\ (g\ x))\ +o\ O(h)$
apply (*drule set-plus-imp-minus*)
apply (*rule set-minus-imp-plus*)
apply (*subst fun-diff-def*)
proof −
assume *a*: $f - g : O(h)$
have $(\%x.\ abs\ (f\ x) - abs\ (g\ x)) =o\ O(\%x.\ abs(abs\ (f\ x) - abs\ (g\ x)))$
by (*rule bigo-abs2*)
also have ... <= $O(\%x.\ abs\ (f\ x - g\ x))$
apply (*rule bigo-elt-subset*)
apply (*rule bigo-bounded*)
apply *force*
apply (*rule allI*)
apply (*rule abs-triangle-ineq3*)
done
also have ... <= $O(f - g)$
apply (*rule bigo-elt-subset*)
apply (*subst fun-diff-def*)
apply (*rule bigo-abs*)
done
also from *a* **have** ... <= $O(h)$
by (*rule bigo-elt-subset*)
finally show $(\%x.\ abs\ (f\ x) - abs\ (g\ x)) : O(h)$**.**
qed

lemma *bigo-abs5*: $f =o\ O(g) ==> (\%x.\ abs(f\ x)) =o\ O(g)$
by (*unfold bigo-def*, *auto*)

lemma *bigo-elt-subset2* [*intro*]: $f : g\ +o\ O(h) ==> O(f) <= O(g) \oplus O(h)$
proof −
assume $f : g\ +o\ O(h)$
also have ... <= $O(g) \oplus O(h)$
by (*auto del*: *subsetI*)
also have ... = $O(\%x.\ abs(g\ x)) \oplus O(\%x.\ abs(h\ x))$

```
    apply (subst bigo-abs3 [symmetric])+
    apply (rule refl)
    done
  also have ... = O((%x. abs(g x)) + (%x. abs(h x)))
    by (rule bigo-plus-eq [symmetric], auto)
  finally have f : ....
  then have O(f) <= ...
    by (elim bigo-elt-subset)
  also have ... = O(%x. abs(g x)) ⊕ O(%x. abs(h x))
    by (rule bigo-plus-eq, auto)
  finally show ?thesis
    by (simp add: bigo-abs3 [symmetric])
qed

lemma bigo-mult [intro]: O(f)⊗O(g) <= O(f * g)
  apply (rule subsetI)
  apply (subst bigo-def)
  apply (auto simp add: bigo-alt-def set-times-def func-times)
  apply (rule-tac x = c * ca in exI)
  apply(rule allI)
  apply(erule-tac x = x in allE)+
  apply(subgoal-tac c * ca * abs(f x * g x) =
      (c * abs(f x)) * (ca * abs(g x)))
  apply(erule ssubst)
  apply (subst abs-mult)
  apply (rule mult-mono)
  apply assumption+
  apply (rule mult-nonneg-nonneg)
  apply auto
  apply (simp add: mult-ac abs-mult)
  done

lemma bigo-mult2 [intro]: f *o O(g) <= O(f * g)
  apply (auto simp add: bigo-def elt-set-times-def func-times abs-mult)
  apply (rule-tac x = c in exI)
  apply auto
  apply (drule-tac x = x in spec)
  apply (subgoal-tac abs(f x) * abs(b x) <= abs(f x) * (c * abs(g x)))
  apply (force simp add: mult-ac)
  apply (rule mult-left-mono, assumption)
  apply (rule abs-ge-zero)
  done

lemma bigo-mult3: f : O(h) ==> g : O(j) ==> f * g : O(h * j)
  apply (rule subsetD)
  apply (rule bigo-mult)
  apply (erule set-times-intro, assumption)
  done
```

lemma *bigo-mult4* [*intro*]:$f : k +o O(h) ==> g * f : (g * k) +o O(g * h)$
apply (*drule set-plus-imp-minus*)
apply (*rule set-minus-imp-plus*)
apply (*drule bigo-mult3* [**where** $g = g$ **and** $j = g$])
apply (*auto simp add*: *ring-simps*)
done

lemma *bigo-mult5*: $ALL\ x.\ f\ x \sim= 0 ==>$
$O(f * g) <= (f::'a => ('b::ordered\text{-}field)) *o O(g)$
proof −
assume $ALL\ x.\ f\ x \sim= 0$
show $O(f * g) <= f *o O(g)$
proof
fix h
assume $h : O(f * g)$
then have $(\%x.\ 1\ /\ (f\ x)) * h : (\%x.\ 1\ /\ f\ x) *o O(f * g)$
by *auto*
also have $... <= O((\%x.\ 1\ /\ f\ x) * (f * g))$
by (*rule bigo-mult2*)
also have $(\%x.\ 1\ /\ f\ x) * (f * g) = g$
apply (*simp add*: *func-times*)
apply (*rule ext*)
apply (*simp add*: *prems nonzero-divide-eq-eq mult-ac*)
done
finally have $(\%x.\ (1::'b)\ /\ f\ x) * h : O(g)$**.**
then have $f * ((\%x.\ (1::'b)\ /\ f\ x) * h) : f *o O(g)$
by *auto*
also have $f * ((\%x.\ (1::'b)\ /\ f\ x) * h) = h$
apply (*simp add*: *func-times*)
apply (*rule ext*)
apply (*simp add*: *prems nonzero-divide-eq-eq mult-ac*)
done
finally show $h : f *o O(g)$**.**
qed
qed

lemma *bigo-mult6*: $ALL\ x.\ f\ x \sim= 0 ==>$
$O(f * g) = (f::'a => ('b::ordered\text{-}field)) *o O(g)$
apply (*rule equalityI*)
apply (*erule bigo-mult5*)
apply (*rule bigo-mult2*)
done

lemma *bigo-mult7*: $ALL\ x.\ f\ x \sim= 0 ==>$
$O(f * g) <= O(f::'a => ('b::ordered\text{-}field)) \otimes O(g)$
apply (*subst bigo-mult6*)
apply *assumption*
apply (*rule set-times-mono3*)
apply (*rule bigo-refl*)

done

lemma *bigo-mult8*: *ALL x. f x ~= 0 ==>*
*O(f * g) = O(f::'a => ('b::ordered-field)) ⊗ O(g)*
apply (*rule equalityI*)
apply (*erule bigo-mult7*)
apply (*rule bigo-mult*)
done

lemma *bigo-minus* [*intro*]: *f : O(g) ==> - f : O(g)*
by (*auto simp add*: *bigo-def fun-Compl-def*)

lemma *bigo-minus2*: *f : g +o O(h) ==> -f : -g +o O(h)*
apply (*rule set-minus-imp-plus*)
apply (*drule set-plus-imp-minus*)
apply (*drule bigo-minus*)
apply (*simp add*: *diff-minus*)
done

lemma *bigo-minus3*: *O(-f) = O(f)*
by (*auto simp add*: *bigo-def fun-Compl-def abs-minus-cancel*)

lemma *bigo-plus-absorb-lemma1*: *f : O(g) ==> f +o O(g) <= O(g)*
proof −
assume *a*: *f : O(g)*
show *f +o O(g) <= O(g)*
proof −
have *f : O(f)* **by** *auto*
then have *f +o O(g) <= O(f) ⊕ O(g)*
by (*auto del*: *subsetI*)
also have *... <= O(g) ⊕ O(g)*
proof −
from *a* **have** *O(f) <= O(g)* **by** (*auto del*: *subsetI*)
thus *?thesis* **by** (*auto del*: *subsetI*)
qed
also have *... <= O(g)* **by** (*simp add*: *bigo-plus-idemp*)
finally show *?thesis* **.**
qed
qed

lemma *bigo-plus-absorb-lemma2*: *f : O(g) ==> O(g) <= f +o O(g)*
proof −
assume *a*: *f : O(g)*
show *O(g) <= f +o O(g)*
proof −
from *a* **have** *-f : O(g)* **by** *auto*
then have *-f +o O(g) <= O(g)* **by** (*elim bigo-plus-absorb-lemma1*)
then have *f +o (-f +o O(g)) <= f +o O(g)* **by** *auto*
also have *f +o (-f +o O(g)) = O(g)*

```
    by (simp add: set-plus-rearranges)
   finally show ?thesis .
  qed
qed

lemma bigo-plus-absorb [simp]: f : O(g) ==> f +o O(g) = O(g)
  apply (rule equalityI)
  apply (erule bigo-plus-absorb-lemma1)
  apply (erule bigo-plus-absorb-lemma2)
  done

lemma bigo-plus-absorb2 [intro]: f : O(g) ==> A <= O(g) ==> f +o A <=
O(g)
  apply (subgoal-tac f +o A <= f +o O(g))
  apply force+
  done

lemma bigo-add-commute-imp: f : g +o O(h) ==> g : f +o O(h)
  apply (subst set-minus-plus [symmetric])
  apply (subgoal-tac g - f = - (f - g))
  apply (erule ssubst)
  apply (rule bigo-minus)
  apply (subst set-minus-plus)
  apply assumption
  apply  (simp add: diff-minus add-ac)
  done

lemma bigo-add-commute: (f : g +o O(h)) = (g : f +o O(h))
  apply (rule iffI)
  apply (erule bigo-add-commute-imp)+
  done

lemma bigo-const1: (%x. c) : O(%x. 1)
  by (auto simp add: bigo-def mult-ac)

lemma bigo-const2 [intro]: O(%x. c) <= O(%x. 1)
  apply (rule bigo-elt-subset)
  apply (rule bigo-const1)
  done

lemma bigo-const3: (c::'a::ordered-field) ~= 0 ==> (%x. 1) : O(%x. c)
  apply (simp add: bigo-def)
  apply (rule-tac x = abs(inverse c) in exI)
  apply (simp add: abs-mult [symmetric])
  done

lemma bigo-const4: (c::'a::ordered-field) ~= 0 ==> O(%x. 1) <= O(%x. c)
  by (rule bigo-elt-subset, rule bigo-const3, assumption)
```

lemma *bigo-const* [*simp*]: (*c*::'*a*::*ordered-field*) ~= *0* ==>
  O(%*x*. *c*) = *O*(%*x*. *1*)
 by (*rule equalityI*, *rule bigo-const2*, *rule bigo-const4*, *assumption*)

lemma *bigo-const-mult1*: (%*x*. *c* * *f x*) : *O*(*f*)
 apply (*simp add*: *bigo-def*)
 apply (*rule-tac x* = *abs*(*c*) **in** *exI*)
 apply (*auto simp add*: *abs-mult* [*symmetric*])
 done

lemma *bigo-const-mult2*: *O*(%*x*. *c* * *f x*) <= *O*(*f*)
 by (*rule bigo-elt-subset*, *rule bigo-const-mult1*)

lemma *bigo-const-mult3*: (*c*::'*a*::*ordered-field*) ~= *0* ==> *f* : *O*(%*x*. *c* * *f x*)
 apply (*simp add*: *bigo-def*)
 apply (*rule-tac x* = *abs*(*inverse c*) **in** *exI*)
 apply (*simp add*: *abs-mult* [*symmetric*] *mult-assoc* [*symmetric*])
 done

lemma *bigo-const-mult4*: (*c*::'*a*::*ordered-field*) ~= *0* ==>
  O(*f*) <= *O*(%*x*. *c* * *f x*)
 by (*rule bigo-elt-subset*, *rule bigo-const-mult3*, *assumption*)

lemma *bigo-const-mult* [*simp*]: (*c*::'*a*::*ordered-field*) ~= *0* ==>
  O(%*x*. *c* * *f x*) = *O*(*f*)
 by (*rule equalityI*, *rule bigo-const-mult2*, *erule bigo-const-mult4*)

lemma *bigo-const-mult5* [*simp*]: (*c*::'*a*::*ordered-field*) ~= *0* ==>
  (%*x*. *c*) *o* *O*(*f*) = *O*(*f*)
 apply (*auto del*: *subsetI*)
 apply (*rule order-trans*)
 apply (*rule bigo-mult2*)
 apply (*simp add*: *func-times*)
 apply (*auto intro*!: *subsetI simp add*: *bigo-def elt-set-times-def func-times*)
 apply (*rule-tac x* = %*y*. *inverse c* * *x y* **in** *exI*)
 apply (*simp add*: *mult-assoc* [*symmetric*] *abs-mult*)
 apply (*rule-tac x* = *abs* (*inverse c*) * *ca* **in** *exI*)
 apply (*rule allI*)
 apply (*subst mult-assoc*)
 apply (*rule mult-left-mono*)
 apply (*erule spec*)
 apply *force*
 done

lemma *bigo-const-mult6* [*intro*]: (%*x*. *c*) *o* *O*(*f*) <= *O*(*f*)
 apply (*auto intro*!: *subsetI*
  simp add: *bigo-def elt-set-times-def func-times*)
 apply (*rule-tac x* = *ca* * (*abs c*) **in** *exI*)
 apply (*rule allI*)

apply (*subgoal-tac ca* $*$ *abs*(*c*) $*$ *abs*(*f x*) = *abs*(*c*) $*$ (*ca* $*$ *abs*(*f x*)))
apply (*erule ssubst*)
apply (*subst abs-mult*)
apply (*rule mult-left-mono*)
apply (*erule spec*)
apply *simp*
apply(*simp add*: *mult-ac*)
done

lemma *bigo-const-mult7* [*intro*]: *f* =*o* *O*(*g*) ==> (%*x*. *c* $*$ *f x*) =*o* *O*(*g*)
proof −
assume *f* =*o* *O*(*g*)
then have (%*x*. *c*) $*$ *f* =*o* (%*x*. *c*) $*o$ *O*(*g*)
by *auto*
also have (%*x*. *c*) $*$ *f* = (%*x*. *c* $*$ *f x*)
by (*simp add*: *func-times*)
also have (%*x*. *c*) $*o$ *O*(*g*) <= *O*(*g*)
by (*auto del*: *subsetI*)
finally show *?thesis* **.**
qed

lemma *bigo-compose1*: *f* =*o* *O*(*g*) ==> (%*x*. *f*(*k x*)) =*o* *O*(%*x*. *g*(*k x*))
by (*unfold bigo-def*, *auto*)

lemma *bigo-compose2*: *f* =*o* *g* +*o* *O*(*h*) ==> (%*x*. *f*(*k x*)) =*o* (%*x*. *g*(*k x*)) +*o*
O(%*x*. *h*(*k x*))
apply (*simp only*: *set-minus-plus* [*symmetric*] *diff-minus fun-Compl-def*
func-plus)
apply (*erule bigo-compose1*)
done

5.2 Setsum

lemma *bigo-setsum-main*: *ALL x*. *ALL y* : *A x*. *0* <= *h x y* ==>
EX c. *ALL x*. *ALL y* : *A x*. *abs*(*f x y*) <= *c* $*$ (*h x y*) ==>
(%*x*. *SUM y* : *A x*. *f x y*) =*o* *O*(%*x*. *SUM y* : *A x*. *h x y*)
apply (*auto simp add*: *bigo-def*)
apply (*rule-tac x* = *abs c* **in** *exI*)
apply (*subst abs-of-nonneg*) **back back**
apply (*rule setsum-nonneg*)
apply *force*
apply (*subst setsum-right-distrib*)
apply (*rule allI*)
apply (*rule order-trans*)
apply (*rule setsum-abs*)
apply (*rule setsum-mono*)
apply (*rule order-trans*)
apply (*drule spec*)+
apply (*drule bspec*)+

apply *assumption*+
apply (*drule bspec*)
apply *assumption*+
apply (*rule mult-right-mono*)
apply (*rule abs-ge-self*)
apply *force*
done

lemma *bigo-setsum1*: *ALL x y. 0 <= h x y ==>*
*EX c. ALL x y. abs(f x y) <= c * (h x y) ==>*
(%x. SUM y : A x. f x y) =o O(%x. SUM y : A x. h x y)
apply (*rule bigo-setsum-main*)
apply *force*
apply *clarsimp*
apply (*rule-tac x = c* **in** *exI*)
apply *force*
done

lemma *bigo-setsum2*: *ALL y. 0 <= h y ==>*
*EX c. ALL y. abs(f y) <= c * (h y) ==>*
(%x. SUM y : A x. f y) =o O(%x. SUM y : A x. h y)
by (*rule bigo-setsum1*, *auto*)

lemma *bigo-setsum3*: *f =o O(h) ==>*
*(%x. SUM y : A x. (l x y) * f(k x y)) =o*
*O(%x. SUM y : A x. abs(l x y * h(k x y)))*
apply (*rule bigo-setsum1*)
apply (*rule allI*)+
apply (*rule abs-ge-zero*)
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac x = c* **in** *exI*)
apply (*rule allI*)+
apply (*subst abs-mult*)+
apply (*subst mult-left-commute*)
apply (*rule mult-left-mono*)
apply (*erule spec*)
apply (*rule abs-ge-zero*)
done

lemma *bigo-setsum4*: *f =o g +o O(h) ==>*
*(%x. SUM y : A x. l x y * f(k x y)) =o*
*(%x. SUM y : A x. l x y * g(k x y)) +o*
*O(%x. SUM y : A x. abs(l x y * h(k x y)))*
apply (*rule set-minus-imp-plus*)
apply (*subst fun-diff-def*)
apply (*subst setsum-subtractf* [*symmetric*])
apply (*subst right-diff-distrib* [*symmetric*])
apply (*rule bigo-setsum3*)

apply (*subst fun-diff-def* [*symmetric*])
apply (*erule set-plus-imp-minus*)
done

lemma *bigo-setsum5*: $f =o O(h) ==> ALL\ x\ y.\ 0 <= l\ x\ y ==>$
$ALL\ x.\ 0 <= h\ x ==>$
$(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * f(k\ x\ y)) =o$
$O(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * h(k\ x\ y))$
apply (*subgoal-tac* $(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * h(k\ x\ y)) =$
$(\%x.\ SUM\ y : A\ x.\ abs((l\ x\ y) * h(k\ x\ y))))$
apply (*erule ssubst*)
apply (*erule bigo-setsum3*)
apply (*rule ext*)
apply (*rule setsum-cong2*)
apply (*subst abs-of-nonneg*)
apply (*rule mult-nonneg-nonneg*)
apply *auto*
done

lemma *bigo-setsum6*: $f =o g +o O(h) ==> ALL\ x\ y.\ 0 <= l\ x\ y ==>$
$ALL\ x.\ 0 <= h\ x ==>$
$(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * f(k\ x\ y)) =o$
$(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * g(k\ x\ y)) +o$
$O(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * h(k\ x\ y))$
apply (*rule set-minus-imp-plus*)
apply (*subst fun-diff-def*)
apply (*subst setsum-subtractf* [*symmetric*])
apply (*subst right-diff-distrib* [*symmetric*])
apply (*rule bigo-setsum5*)
apply (*subst fun-diff-def* [*symmetric*])
apply (*drule set-plus-imp-minus*)
apply *auto*
done

5.3 Misc useful stuff

lemma *bigo-useful-intro*: $A <= O(f) ==> B <= O(f) ==>$
$A \oplus B <= O(f)$
apply (*subst bigo-plus-idemp* [*symmetric*])
apply (*rule set-plus-mono2*)
apply *assumption+*
done

lemma *bigo-useful-add*: $f =o O(h) ==> g =o O(h) ==> f + g =o O(h)$
apply (*subst bigo-plus-idemp* [*symmetric*])
apply (*rule set-plus-intro*)
apply *assumption+*
done

lemma *bigo-useful-const-mult*: $(c::'a::ordered\text{-}field) \sim= 0 ==>$
$(\%x.\ c) * f =o O(h) ==> f =o O(h)$
apply (*rule subsetD*)
apply (*subgoal-tac* $(\%x.\ 1\ /\ c) *o O(h) <= O(h)$)
apply *assumption*
apply (*rule bigo-const-mult6*)
apply (*subgoal-tac* $f = (\%x.\ 1\ /\ c) * ((\%x.\ c) * f)$)
apply (*erule ssubst*)
apply (*erule set-times-intro2*)
apply (*simp add*: *func-times*)
done

lemma *bigo-fix*: $(\%x.\ f\ ((x::nat) + 1)) =o O(\%x.\ h(x + 1)) ==> f\ 0 = 0 ==>$
$f =o O(h)$
apply (*simp add*: *bigo-alt-def*)
apply *auto*
apply (*rule-tac* $x = c$ **in** *exI*)
apply *auto*
apply (*case-tac* $x = 0$)
apply *simp*
apply (*rule mult-nonneg-nonneg*)
apply *force*
apply *force*
apply (*subgoal-tac* $x = Suc\ (x - 1)$)
apply (*erule ssubst*) **back**
apply (*erule spec*)
apply *simp*
done

lemma *bigo-fix2*:
$(\%x.\ f\ ((x::nat) + 1)) =o (\%x.\ g(x + 1)) +o O(\%x.\ h(x + 1)) ==>$
$f\ 0 = g\ 0 ==> f =o g +o O(h)$
apply (*rule set-minus-imp-plus*)
apply (*rule bigo-fix*)
apply (*subst fun-diff-def*)
apply (*subst fun-diff-def* [*symmetric*])
apply (*rule set-plus-imp-minus*)
apply *simp*
apply (*simp add*: *fun-diff-def*)
done

5.4 Less than or equal to

definition
$lesso :: ('a => 'b::ordered\text{-}idom) => ('a => 'b) => ('a => 'b)$
(**infixl** $<o$ *70*) **where**
$f <o g = (\%x.\ max\ (f\ x - g\ x)\ 0)$

lemma *bigo-lesseq1*: $f =o O(h) ==> ALL\ x.\ abs\ (g\ x) <= abs\ (f\ x) ==>$

```
    g =o O(h)
  apply (unfold bigo-def)
  apply clarsimp
  apply (rule-tac x = c in exI)
  apply (rule allI)
  apply (rule order-trans)
  apply (erule spec)+
  done

lemma bigo-lesseq2: f =o O(h) ==> ALL x. abs (g x) <= f x ==>
      g =o O(h)
  apply (erule bigo-lesseq1)
  apply (rule allI)
  apply (drule-tac x = x in spec)
  apply (rule order-trans)
  apply assumption
  apply (rule abs-ge-self)
  done

lemma bigo-lesseq3: f =o O(h) ==> ALL x. 0 <= g x ==> ALL x. g x <= f
x ==>
    g =o O(h)
  apply (erule bigo-lesseq2)
  apply (rule allI)
  apply (subst abs-of-nonneg)
  apply (erule spec)+
  done

lemma bigo-lesseq4: f =o O(h) ==>
    ALL x. 0 <= g x ==> ALL x. g x <= abs (f x) ==>
      g =o O(h)
  apply (erule bigo-lesseq1)
  apply (rule allI)
  apply (subst abs-of-nonneg)
  apply (erule spec)+
  done

lemma bigo-lesso1: ALL x. f x <= g x ==> f <o g =o O(h)
  apply (unfold lesso-def)
  apply (subgoal-tac (%x. max (f x - g x) 0) = 0)
  apply (erule ssubst)
  apply (rule bigo-zero)
  apply (unfold func-zero)
  apply (rule ext)
  apply (simp split: split-max)
  done

lemma bigo-lesso2: f =o g +o O(h) ==>
    ALL x. 0 <= k x ==> ALL x. k x <= f x ==>
```

```
    k <o g =o O(h)
  apply (unfold lesso-def)
  apply (rule bigo-lesseq4)
  apply (erule set-plus-imp-minus)
  apply (rule allI)
  apply (rule le-maxI2)
  apply (rule allI)
  apply (subst fun-diff-def)
  apply (case-tac 0 <= k x - g x)
  apply simp
  apply (subst abs-of-nonneg)
  apply (drule-tac x = x in spec) back
  apply (simp add: compare-rls)
  apply (subst diff-minus)+
  apply (rule add-right-mono)
  apply (erule spec)
  apply (rule order-trans)
  prefer 2
  apply (rule abs-ge-zero)
  apply (simp add: compare-rls)
  done

lemma bigo-lesso3: f =o g +o O(h) ==>
    ALL x. 0 <= k x ==> ALL x. g x <= k x ==>
      f <o k =o O(h)
  apply (unfold lesso-def)
  apply (rule bigo-lesseq4)
  apply (erule set-plus-imp-minus)
  apply (rule allI)
  apply (rule le-maxI2)
  apply (rule allI)
  apply (subst fun-diff-def)
  apply (case-tac 0 <= f x - k x)
  apply simp
  apply (subst abs-of-nonneg)
  apply (drule-tac x = x in spec) back
  apply (simp add: compare-rls)
  apply (subst diff-minus)+
  apply (rule add-left-mono)
  apply (rule le-imp-neg-le)
  apply (erule spec)
  apply (rule order-trans)
  prefer 2
  apply (rule abs-ge-zero)
  apply (simp add: compare-rls)
  done

lemma bigo-lesso4: f <o g =o O(k::'a=>'b::ordered-field) ==>
    g =o h +o O(k) ==> f <o h =o O(k)
```

apply (*unfold lesso-def*)
apply (*drule set-plus-imp-minus*)
apply (*drule bigo-abs5*) **back**
apply (*simp add*: *fun-diff-def*)
apply (*drule bigo-useful-add*)
apply *assumption*
apply (*erule bigo-lesseq2*) **back**
apply (*rule allI*)
apply (*auto simp add*: *func-plus fun-diff-def compare-rls*
split: *split-max abs-split*)
done

lemma *bigo-lesso5*: $f <o g =o O(h) ==>$
$EX\ C.\ ALL\ x.\ f\ x <= g\ x + C * abs(h\ x)$
apply (*simp only*: *lesso-def bigo-alt-def*)
apply *clarsimp*
apply (*rule-tac x = c* **in** *exI*)
apply (*rule allI*)
apply (*drule-tac x = x* **in** *spec*)
apply (*subgoal-tac abs(max (f x − g x) 0) = max (f x − g x) 0*)
apply (*clarsimp simp add*: *compare-rls add-ac*)
apply (*rule abs-of-nonneg*)
apply (*rule le-maxI2*)
done

lemma *lesso-add*: $f <o g =o O(h) ==>$
$k <o l =o O(h) ==> (f + k) <o (g + l) =o O(h)$
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq3*)
apply (*erule bigo-useful-add*)
apply *assumption*
apply (*force split*: *split-max*)
apply (*auto split*: *split-max simp add*: *func-plus*)
done

end

6 Binomial: Binomial Coefficients

theory *Binomial*
imports *ATP-Linkup*
begin

This development is based on the work of Andy Gordon and Florian Kammueller.

consts
binomial :: *nat* ⇒ *nat* ⇒ *nat* (**infixl** *choose 65*)
primrec

binomial-0: $(0 \text{ choose } k) = (\text{if } k = 0 \text{ then } 1 \text{ else } 0)$
binomial-Suc: $(\text{Suc } n \text{ choose } k) =$
$(\text{if } k = 0 \text{ then } 1 \text{ else } (n \text{ choose } (k - 1)) + (n \text{ choose } k))$

lemma *binomial-n-0* [*simp*]: $(n \text{ choose } 0) = 1$
by (*cases n*) *simp-all*

lemma *binomial-0-Suc* [*simp*]: $(0 \text{ choose Suc } k) = 0$
by *simp*

lemma *binomial-Suc-Suc* [*simp*]:
$(\text{Suc } n \text{ choose Suc } k) = (n \text{ choose } k) + (n \text{ choose Suc } k)$
by *simp*

lemma *binomial-eq-0*: $!!k.\ n < k \Longrightarrow (n \text{ choose } k) = 0$
by (*induct n*) *auto*

declare *binomial-0* [*simp del*] *binomial-Suc* [*simp del*]

lemma *binomial-n-n* [*simp*]: $(n \text{ choose } n) = 1$
by (*induct n*) (*simp-all add*: *binomial-eq-0*)

lemma *binomial-Suc-n* [*simp*]: $(\text{Suc } n \text{ choose } n) = \text{Suc } n$
by (*induct n*) *simp-all*

lemma *binomial-1* [*simp*]: $(n \text{ choose Suc } 0) = n$
by (*induct n*) *simp-all*

lemma *zero-less-binomial*: $k \leq n \Longrightarrow (n \text{ choose } k) > 0$
by (*induct n k rule*: *diff-induct*) *simp-all*

lemma *binomial-eq-0-iff*: $(n \text{ choose } k = 0) = (n<k)$
apply (*safe intro*!: *binomial-eq-0*)
apply (*erule contrapos-pp*)
apply (*simp add*: *zero-less-binomial*)
done

lemma *zero-less-binomial-iff*: $(n \text{ choose } k > 0) = (k \leq n)$
by(*simp add*: *linorder-not-less binomial-eq-0-iff neq0-conv*[*symmetric*]
del:*neq0-conv*)

lemma *Suc-times-binomial-eq*:
$!!k.\ k \leq n \Longrightarrow \text{Suc } n * (n \text{ choose } k) = (\text{Suc } n \text{ choose Suc } k) * \text{Suc } k$
apply (*induct n*)
apply (*simp add*: *binomial-0*)
apply (*case-tac k*)
apply (*auto simp add*: *add-mult-distrib add-mult-distrib2 le-Suc-eq*
binomial-eq-0)

done

This is the well-known version, but it's harder to use because of the need to reason about division.

lemma *binomial-Suc-Suc-eq-times*:
$k \leq n \Longrightarrow (Suc\ n\ choose\ Suc\ k) = (Suc\ n * (n\ choose\ k))\ div\ Suc\ k$
by (*simp add*: *Suc-times-binomial-eq div-mult-self-is-m zero-less-Suc*
del: *mult-Suc mult-Suc-right*)

Another version, with -1 instead of Suc.

lemma *times-binomial-minus1-eq*:
$[|k \leq n;\ 0<k|] \Longrightarrow (n\ choose\ k) * k = n * ((n - 1)\ choose\ (k - 1))$
apply (*cut-tac* $n = n - 1$ **and** $k = k - 1$ **in** *Suc-times-binomial-eq*)
apply (*simp split add*: *nat-diff-split*, *auto*)
done

6.1 Theorems about *choose*

Basic theorem about *choose*. By Florian Kammüller, tidied by LCP.

lemma *card-s-0-eq-empty*:
finite $A \Longrightarrow card\ \{B.\ B \subseteq A\ \&\ card\ B = 0\} = 1$
apply (*simp cong add*: *conj-cong add*: *finite-subset* [*THEN card-0-eq*])
apply (*simp cong add*: *rev-conj-cong*)
done

lemma *choose-deconstruct*: *finite* $M \Longrightarrow x \notin M$
$\Longrightarrow \{s.\ s <= insert\ x\ M\ \&\ card(s) = Suc\ k\}$
$= \{s.\ s <= M\ \&\ card(s) = Suc\ k\}\ Un$
$\{s.\ EX\ t.\ t <= M\ \&\ card(t) = k\ \&\ s = insert\ x\ t\}$
apply *safe*
apply (*auto intro*: *finite-subset* [*THEN card-insert-disjoint*])
apply (*drule-tac* $x = xa - \{x\}$ **in** *spec*)
apply (*subgoal-tac* $x \notin xa$, *auto*)
apply (*erule rev-mp*, *subst card-Diff-singleton*)
apply (*auto intro*: *finite-subset*)
done

There are as many subsets of A having cardinality k as there are sets obtained from the former by inserting a fixed element x into each.

lemma *constr-bij*:
$[|$*finite* A; $x \notin A|] \Longrightarrow$
card $\{B.\ EX\ C.\ C <= A\ \&\ card(C) = k\ \&\ B = insert\ x\ C\} =$
card $\{B.\ B <= A\ \&\ card(B) = k\}$
apply (*rule-tac* $f = \%s.\ s - \{x\}$ **and** $g = insert\ x$ **in** *card-bij-eq*)
apply (*auto elim*!: *equalityE simp add*: *inj-on-def*)
apply (*subst Diff-insert0*, *auto*)

finiteness of the two sets

```
  apply (rule-tac [2] B = Pow (A) in finite-subset)
  apply (rule-tac B = Pow (insert x A) in finite-subset)
  apply fast+
 done
```

Main theorem: combinatorial statement about number of subsets of a set.

```
lemma n-sub-lemma:
   !!A. finite A ==> card {B. B <= A & card B = k} = (card A choose k)
 apply (induct k)
  apply (simp add: card-s-0-eq-empty, atomize)
 apply (rotate-tac -1, erule finite-induct)
  apply (simp-all (no-asm-simp) cong add: conj-cong
    add: card-s-0-eq-empty choose-deconstruct)
 apply (subst card-Un-disjoint)
    prefer 4 apply (force simp add: constr-bij)
   prefer 3 apply force
  prefer 2 apply (blast intro: finite-Pow-iff [THEN iffD2]
    finite-subset [of - Pow (insert x F), standard])
 apply (blast intro: finite-Pow-iff [THEN iffD2, THEN [2] finite-subset])
 done

theorem n-subsets:
   finite A ==> card {B. B <= A & card B = k} = (card A choose k)
 by (simp add: n-sub-lemma)
```

The binomial theorem (courtesy of Tobias Nipkow):

```
theorem binomial: (a+b::nat)^n = (∑k=0..n. (n choose k) * a^k * b^(n-k))
proof (induct n)
 case 0 thus ?case by simp
next
 case (Suc n)
 have decomp: {0..n+1} = {0} ∪ {n+1} ∪ {1..n}
   by (auto simp add:atLeastAtMost-def atLeast-def atMost-def)
 have decomp2: {0..n} = {0} ∪ {1..n}
   by (auto simp add:atLeastAtMost-def atLeast-def atMost-def)
 have (a+b::nat)^(n+1) = (a+b) * (∑k=0..n. (n choose k) * a^k * b^(n-k))
   using Suc by simp
 also have ... =  a*(∑k=0..n. (n choose k) * a^k * b^(n-k)) +
                  b*(∑k=0..n. (n choose k) * a^k * b^(n-k))
   by (rule nat-distrib)
 also have ... = (∑k=0..n. (n choose k) * a^(k+1) * b^(n-k)) +
                 (∑k=0..n. (n choose k) * a^k * b^(n-k+1))
   by (simp add: setsum-right-distrib mult-ac)
 also have ... = (∑k=0..n. (n choose k) * a^k * b^(n+1-k)) +
                 (∑k=1..n+1. (n choose (k - 1)) * a^k * b^(n+1-k))
   by (simp add:setsum-shift-bounds-cl-Suc-ivl Suc-diff-le
         del:setsum-cl-ivl-Suc)
 also have ... = a^(n+1) + b^(n+1) +
```

$(\sum k{=}1..n.\ (n\ choose\ (k\ -\ 1))\ *\ a\hat{}k\ *\ b\hat{}(n{+}1{-}k))\ +$
$(\sum k{=}1..n.\ (n\ choose\ k)\ *\ a\hat{}k\ *\ b\hat{}(n{+}1{-}k))$
by (*simp add*: *decomp2*)
also have
$\ldots = a\hat{}(n{+}1) + b\hat{}(n{+}1) + (\sum k{=}1..n.\ (n{+}1\ choose\ k)\ *\ a\hat{}k\ *\ b\hat{}(n{+}1{-}k))$
by (*simp add*: *nat-distrib setsum-addf binomial.simps*)
also have $\ldots = (\sum k{=}0..n{+}1.\ (n{+}1\ choose\ k)\ *\ a\hat{}k\ *\ b\hat{}(n{+}1{-}k))$
using *decomp* **by** *simp*
finally show *?case* **by** *simp*
qed

end

7 Boolean-Algebra: Boolean Algebras

theory *Boolean-Algebra*
imports *ATP-Linkup*
begin

locale *boolean* =
fixes *conj* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\sqcap$ *70*)
fixes *disj* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\sqcup$ *65*)
fixes *compl* :: $'a \Rightarrow 'a$ ($\sim$ - [*81*] *80*)
fixes *zero* :: $'a$ (**0**)
fixes *one* :: $'a$ (**1**)
assumes *conj-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
assumes *disj-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
assumes *conj-commute*: $x \sqcap y = y \sqcap x$
assumes *disj-commute*: $x \sqcup y = y \sqcup x$
assumes *conj-disj-distrib*: $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
assumes *disj-conj-distrib*: $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
assumes *conj-one-right* [*simp*]: $x \sqcap \mathbf{1} = x$
assumes *disj-zero-right* [*simp*]: $x \sqcup \mathbf{0} = x$
assumes *conj-cancel-right* [*simp*]: $x \sqcap \sim x = \mathbf{0}$
assumes *disj-cancel-right* [*simp*]: $x \sqcup \sim x = \mathbf{1}$
begin

lemmas *disj-ac* =
disj-assoc disj-commute
mk-left-commute [**where** $'a = 'a$, *of disj*, *OF disj-assoc disj-commute*]

lemmas *conj-ac* =
conj-assoc conj-commute
mk-left-commute [**where** $'a = 'a$, *of conj*, *OF conj-assoc conj-commute*]

lemma *dual*: *boolean disj conj compl one zero*
apply (*rule boolean.intro*)
apply (*rule disj-assoc*)

apply (*rule conj-assoc*)
apply (*rule disj-commute*)
apply (*rule conj-commute*)
apply (*rule disj-conj-distrib*)
apply (*rule conj-disj-distrib*)
apply (*rule disj-zero-right*)
apply (*rule conj-one-right*)
apply (*rule disj-cancel-right*)
apply (*rule conj-cancel-right*)
done

7.1 Complement

lemma *complement-unique*:
  assumes *1*: $a \sqcap x = \mathbf{0}$
  assumes *2*: $a \sqcup x = \mathbf{1}$
  assumes *3*: $a \sqcap y = \mathbf{0}$
  assumes *4*: $a \sqcup y = \mathbf{1}$
  shows $x = y$
proof −
  have $(a \sqcap x) \sqcup (x \sqcap y) = (a \sqcap y) \sqcup (x \sqcap y)$ **using** *1 3* **by** *simp*
  hence $(x \sqcap a) \sqcup (x \sqcap y) = (y \sqcap a) \sqcup (y \sqcap x)$ **using** *conj-commute* **by** *simp*
  hence $x \sqcap (a \sqcup y) = y \sqcap (a \sqcup x)$ **using** *conj-disj-distrib* **by** *simp*
  hence $x \sqcap \mathbf{1} = y \sqcap \mathbf{1}$ **using** *2 4* **by** *simp*
  thus $x = y$ **using** *conj-one-right* **by** *simp*
qed

lemma *compl-unique*: $[\![x \sqcap y = \mathbf{0};\ x \sqcup y = \mathbf{1}]\!] \Longrightarrow \sim x = y$
by (*rule complement-unique* [*OF conj-cancel-right disj-cancel-right*])

lemma *double-compl* [*simp*]: $\sim (\sim x) = x$
proof (*rule compl-unique*)
  from *conj-cancel-right* **show** $\sim x \sqcap x = \mathbf{0}$ **by** (*simp only*: *conj-commute*)
  from *disj-cancel-right* **show** $\sim x \sqcup x = \mathbf{1}$ **by** (*simp only*: *disj-commute*)
qed

lemma *compl-eq-compl-iff* [*simp*]: $(\sim x = \sim y) = (x = y)$
by (*rule inj-eq* [*OF inj-on-inverseI*], *rule double-compl*)

7.2 Conjunction

lemma *conj-absorb* [*simp*]: $x \sqcap x = x$
proof −
  have $x \sqcap x = (x \sqcap x) \sqcup \mathbf{0}$ **using** *disj-zero-right* **by** *simp*
  also have ... $= (x \sqcap x) \sqcup (x \sqcap \sim x)$ **using** *conj-cancel-right* **by** *simp*
  also have ... $= x \sqcap (x \sqcup \sim x)$ **using** *conj-disj-distrib* **by** (*simp only*:)
  also have ... $= x \sqcap \mathbf{1}$ **using** *disj-cancel-right* **by** *simp*
  also have ... $= x$ **using** *conj-one-right* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *conj-zero-right* [*simp*]: $x \sqcap \mathbf{0} = \mathbf{0}$
proof −
  have $x \sqcap \mathbf{0} = x \sqcap (x \sqcap \sim x)$ **using** *conj-cancel-right* **by** *simp*
  also have ... $= (x \sqcap x) \sqcap \sim x$ **using** *conj-assoc* **by** (*simp only*:)
  also have ... $= x \sqcap \sim x$ **using** *conj-absorb* **by** *simp*
  also have ... $= \mathbf{0}$ **using** *conj-cancel-right* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *compl-one* [*simp*]: $\sim \mathbf{1} = \mathbf{0}$
by (*rule compl-unique* [*OF conj-zero-right disj-zero-right*])

lemma *conj-zero-left* [*simp*]: $\mathbf{0} \sqcap x = \mathbf{0}$
by (*subst conj-commute*) (*rule conj-zero-right*)

lemma *conj-one-left* [*simp*]: $\mathbf{1} \sqcap x = x$
by (*subst conj-commute*) (*rule conj-one-right*)

lemma *conj-cancel-left* [*simp*]: $\sim x \sqcap x = \mathbf{0}$
by (*subst conj-commute*) (*rule conj-cancel-right*)

lemma *conj-left-absorb* [*simp*]: $x \sqcap (x \sqcap y) = x \sqcap y$
by (*simp only*: *conj-assoc* [*symmetric*] *conj-absorb*)

lemma *conj-disj-distrib2*:
  $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
by (*simp only*: *conj-commute conj-disj-distrib*)

lemmas *conj-disj-distribs* =
  conj-disj-distrib conj-disj-distrib2

7.3 Disjunction

lemma *disj-absorb* [*simp*]: $x \sqcup x = x$
by (*rule boolean.conj-absorb* [*OF dual*])

lemma *disj-one-right* [*simp*]: $x \sqcup \mathbf{1} = \mathbf{1}$
by (*rule boolean.conj-zero-right* [*OF dual*])

lemma *compl-zero* [*simp*]: $\sim \mathbf{0} = \mathbf{1}$
by (*rule boolean.compl-one* [*OF dual*])

lemma *disj-zero-left* [*simp*]: $\mathbf{0} \sqcup x = x$
by (*rule boolean.conj-one-left* [*OF dual*])

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$
by (*rule boolean.conj-zero-left* [*OF dual*])

lemma *disj-cancel-left* [*simp*]: $\sim x \sqcup x = \mathbf{1}$
by (*rule boolean.conj-cancel-left* [*OF dual*])

lemma *disj-left-absorb* [*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
by (*rule boolean.conj-left-absorb* [*OF dual*])

lemma *disj-conj-distrib2*:
$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
by (*rule boolean.conj-disj-distrib2* [*OF dual*])

lemmas *disj-conj-distribs* =
disj-conj-distrib disj-conj-distrib2

7.4 De Morgan's Laws

lemma *de-Morgan-conj* [*simp*]: $\sim (x \sqcap y) = \sim x \sqcup \sim y$
proof (*rule compl-unique*)
have $(x \sqcap y) \sqcap (\sim x \sqcup \sim y) = ((x \sqcap y) \sqcap \sim x) \sqcup ((x \sqcap y) \sqcap \sim y)$
by (*rule conj-disj-distrib*)
also have ... $= (y \sqcap (x \sqcap \sim x)) \sqcup (x \sqcap (y \sqcap \sim y))$
by (*simp only*: *conj-ac*)
finally show $(x \sqcap y) \sqcap (\sim x \sqcup \sim y) = \mathbf{0}$
by (*simp only*: *conj-cancel-right conj-zero-right disj-zero-right*)
next
have $(x \sqcap y) \sqcup (\sim x \sqcup \sim y) = (x \sqcup (\sim x \sqcup \sim y)) \sqcap (y \sqcup (\sim x \sqcup \sim y))$
by (*rule disj-conj-distrib2*)
also have ... $= (\sim y \sqcup (x \sqcup \sim x)) \sqcap (\sim x \sqcup (y \sqcup \sim y))$
by (*simp only*: *disj-ac*)
finally show $(x \sqcap y) \sqcup (\sim x \sqcup \sim y) = \mathbf{1}$
by (*simp only*: *disj-cancel-right disj-one-right conj-one-right*)
qed

lemma *de-Morgan-disj* [*simp*]: $\sim (x \sqcup y) = \sim x \sqcap \sim y$
by (*rule boolean.de-Morgan-conj* [*OF dual*])

end

7.5 Symmetric Difference

locale *boolean-xor* = *boolean* +
fixes *xor* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\oplus$ *65*)
assumes *xor-def*: $x \oplus y = (x \sqcap \sim y) \sqcup (\sim x \sqcap y)$
begin

lemma *xor-def2*:
$x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$
by (*simp only*: *xor-def conj-disj-distribs*
disj-ac conj-ac conj-cancel-right disj-zero-left)

lemma *xor-commute*: $x \oplus y = y \oplus x$

by (*simp only*: *xor-def conj-commute disj-commute*)

lemma *xor-assoc*: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
proof −
  let *?t* $= (x \sqcap y \sqcap z) \sqcup (x \sqcap \sim y \sqcap \sim z) \sqcup$
        $(\sim x \sqcap y \sqcap \sim z) \sqcup (\sim x \sqcap \sim y \sqcap z)$
  have *?t* $\sqcup (z \sqcap x \sqcap \sim x) \sqcup (z \sqcap y \sqcap \sim y) =$
      ?t $\sqcup (x \sqcap y \sqcap \sim y) \sqcup (x \sqcap z \sqcap \sim z)$
    by (*simp only*: *conj-cancel-right conj-zero-right*)
  thus $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
    apply (*simp only*: *xor-def de-Morgan-disj de-Morgan-conj double-compl*)
    apply (*simp only*: *conj-disj-distribs conj-ac disj-ac*)
    done
qed

lemmas *xor-ac* =
  xor-assoc xor-commute
  mk-left-commute [**where** $'a = 'a$, *of xor*, *OF xor-assoc xor-commute*]

lemma *xor-zero-right* [*simp*]: $x \oplus \mathbf{0} = x$
by (*simp only*: *xor-def compl-zero conj-one-right conj-zero-right disj-zero-right*)

lemma *xor-zero-left* [*simp*]: $\mathbf{0} \oplus x = x$
by (*subst xor-commute*) (*rule xor-zero-right*)

lemma *xor-one-right* [*simp*]: $x \oplus \mathbf{1} = \sim x$
by (*simp only*: *xor-def compl-one conj-zero-right conj-one-right disj-zero-left*)

lemma *xor-one-left* [*simp*]: $\mathbf{1} \oplus x = \sim x$
by (*subst xor-commute*) (*rule xor-one-right*)

lemma *xor-self* [*simp*]: $x \oplus x = \mathbf{0}$
by (*simp only*: *xor-def conj-cancel-right conj-cancel-left disj-zero-right*)

lemma *xor-left-self* [*simp*]: $x \oplus (x \oplus y) = y$
by (*simp only*: *xor-assoc* [*symmetric*] *xor-self xor-zero-left*)

lemma *xor-compl-left*: $\sim x \oplus y = \sim (x \oplus y)$
apply (*simp only*: *xor-def de-Morgan-disj de-Morgan-conj double-compl*)
apply (*simp only*: *conj-disj-distribs*)
apply (*simp only*: *conj-cancel-right conj-cancel-left*)
apply (*simp only*: *disj-zero-left disj-zero-right*)
apply (*simp only*: *disj-ac conj-ac*)
done

lemma *xor-compl-right*: $x \oplus \sim y = \sim (x \oplus y)$
apply (*simp only*: *xor-def de-Morgan-disj de-Morgan-conj double-compl*)
apply (*simp only*: *conj-disj-distribs*)
apply (*simp only*: *conj-cancel-right conj-cancel-left*)

apply (*simp only*: *disj-zero-left disj-zero-right*)
apply (*simp only*: *disj-ac conj-ac*)
done

lemma *xor-cancel-right* [*simp*]: $x \oplus \sim x = \mathbf{1}$
by (*simp only*: *xor-compl-right xor-self compl-zero*)

lemma *xor-cancel-left* [*simp*]: $\sim x \oplus x = \mathbf{1}$
by (*subst xor-commute*) (*rule xor-cancel-right*)

lemma *conj-xor-distrib*: $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
proof −
  have $(x \sqcap y \sqcap \sim z) \sqcup (x \sqcap \sim y \sqcap z) =$
      $(y \sqcap x \sqcap \sim x) \sqcup (z \sqcap x \sqcap \sim x) \sqcup (x \sqcap y \sqcap \sim z) \sqcup (x \sqcap \sim y \sqcap z)$
    by (*simp only*: *conj-cancel-right conj-zero-right disj-zero-left*)
  thus $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
    by (*simp* (*no-asm-use*) *only*:
      xor-def de-Morgan-disj de-Morgan-conj double-compl
      conj-disj-distribs conj-ac disj-ac)
qed

lemma *conj-xor-distrib2*:
  $(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$
proof −
  have $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
    by (*rule conj-xor-distrib*)
  thus $(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$
    by (*simp only*: *conj-commute*)
qed

lemmas *conj-xor-distribs* =
  conj-xor-distrib conj-xor-distrib2

end

end

8 Product-ord: Order on product types

theory *Product-ord*
imports *ATP-Linkup*
begin

instantiation $*$ :: (*ord*, *ord*) *ord*
begin

definition
  prod-le-def [*code func del*]: $x \leq y \longleftrightarrow fst\ x < fst\ y \vee fst\ x = fst\ y \wedge snd\ x \leq$

snd y

definition
prod-less-def [*code func del*]: $x < y \longleftrightarrow fst\ x < fst\ y \vee fst\ x = fst\ y \wedge snd\ x < snd\ y$

instance ..

end

lemma [*code*, *code func del*]:
$(x1,\ y1) \leq (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 = x2 \wedge y1 \leq y2$
$(x1,\ y1) < (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 = x2 \wedge y1 < y2$
unfolding *prod-le-def prod-less-def* **by** *simp-all*

lemma [*code func*]:
$(x1::'a::\{ord,\ eq\},\ y1) \leq (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 = x2 \wedge y1 \leq y2$
$(x1::'a::\{ord,\ eq\},\ y1) < (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 = x2 \wedge y1 < y2$
unfolding *prod-le-def prod-less-def* **by** *simp-all*

instance $*$ :: (*order*, *order*) *order*
by *default* (*auto simp*: *prod-le-def prod-less-def intro*: *order-less-trans*)

instance $*$ :: (*linorder*, *linorder*) *linorder*
by *default* (*auto simp*: *prod-le-def*)

instantiation $*$ :: (*linorder*, *linorder*) *distrib-lattice*
begin

definition
inf-prod-def: $(inf :: 'a \times 'b \Rightarrow - \Rightarrow -) = min$

definition
sup-prod-def: $(sup :: 'a \times 'b \Rightarrow - \Rightarrow -) = max$

instance
by *intro-classes*
(*auto simp add*: *inf-prod-def sup-prod-def min-max.sup-inf-distrib1*)

end

end

9 Char-nat: Mapping between characters and natural numbers

theory *Char-nat*

imports *List*
begin

Conversions between nibbles and natural numbers in [0..15].

primrec
nat-of-nibble :: *nibble* ⇒ *nat* **where**
nat-of-nibble Nibble0 = 0
| *nat-of-nibble Nibble1 = 1*
| *nat-of-nibble Nibble2 = 2*
| *nat-of-nibble Nibble3 = 3*
| *nat-of-nibble Nibble4 = 4*
| *nat-of-nibble Nibble5 = 5*
| *nat-of-nibble Nibble6 = 6*
| *nat-of-nibble Nibble7 = 7*
| *nat-of-nibble Nibble8 = 8*
| *nat-of-nibble Nibble9 = 9*
| *nat-of-nibble NibbleA = 10*
| *nat-of-nibble NibbleB = 11*
| *nat-of-nibble NibbleC = 12*
| *nat-of-nibble NibbleD = 13*
| *nat-of-nibble NibbleE = 14*
| *nat-of-nibble NibbleF = 15*

definition
nibble-of-nat :: *nat* ⇒ *nibble* **where**
nibble-of-nat x = (*let y* = *x mod 16 in*
if y = 0 then Nibble0 else
if y = 1 then Nibble1 else
if y = 2 then Nibble2 else
if y = 3 then Nibble3 else
if y = 4 then Nibble4 else
if y = 5 then Nibble5 else
if y = 6 then Nibble6 else
if y = 7 then Nibble7 else
if y = 8 then Nibble8 else
if y = 9 then Nibble9 else
if y = 10 then NibbleA else
if y = 11 then NibbleB else
if y = 12 then NibbleC else
if y = 13 then NibbleD else
if y = 14 then NibbleE else
NibbleF)

lemma *nibble-of-nat-norm*:
nibble-of-nat (*n mod 16*) = *nibble-of-nat n*
unfolding *nibble-of-nat-def Let-def* **by** *auto*

lemmas [*code func*] = *nibble-of-nat-norm* [*symmetric*]

lemma *nibble-of-nat-simps* [*simp*]:
nibble-of-nat 0 = Nibble0
nibble-of-nat 1 = Nibble1
nibble-of-nat 2 = Nibble2
nibble-of-nat 3 = Nibble3
nibble-of-nat 4 = Nibble4
nibble-of-nat 5 = Nibble5
nibble-of-nat 6 = Nibble6
nibble-of-nat 7 = Nibble7
nibble-of-nat 8 = Nibble8
nibble-of-nat 9 = Nibble9
nibble-of-nat 10 = NibbleA
nibble-of-nat 11 = NibbleB
nibble-of-nat 12 = NibbleC
nibble-of-nat 13 = NibbleD
nibble-of-nat 14 = NibbleE
nibble-of-nat 15 = NibbleF
unfolding *nibble-of-nat-def Let-def* **by** *auto*

lemmas *nibble-of-nat-code* [*code func*] = *nibble-of-nat-simps*
[*simplified nat-number Let-def not-neg-number-of-Pls neg-number-of-Bit0 neg-number-of-Bit1 if-False add-0 add-Suc*]

lemma *nibble-of-nat-of-nibble*: *nibble-of-nat* (*nat-of-nibble n*) = *n*
by (*cases n*) (*simp-all only*: *nat-of-nibble.simps nibble-of-nat-simps*)

lemma *nat-of-nibble-of-nat*: *nat-of-nibble* (*nibble-of-nat n*) = *n mod 16*
proof −
have *nibble-nat-enum*:
n mod 16 ∈ {*0*, *1*, *2*, *3*, *4*, *5*, *6*, *7*, *8*, *9*, *10*, *11*, *12*, *13*, *14*, *15*}
proof −
have *set-unfold*: ⋀*n*. {*0*..*Suc n*} = *insert* (*Suc n*) {*0*..*n*} **by** *auto*
have (*n*::*nat*) *mod 16* ∈ {*0*..*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))))))))))))} **by** *simp*
from *this* [*simplified set-unfold atLeastAtMost-singleton*]
show *?thesis* **by** *auto*
qed
then show *?thesis* **unfolding** *nibble-of-nat-def Let-def*
by *auto*
qed

lemma *inj-nat-of-nibble*: *inj nat-of-nibble*
by (*rule inj-on-inverseI*) (*rule nibble-of-nat-of-nibble*)

lemma *nat-of-nibble-eq*: *nat-of-nibble n* = *nat-of-nibble m* ⟷ *n* = *m*
by (*rule inj-eq*) (*rule inj-nat-of-nibble*)

lemma *nat-of-nibble-less-16*: *nat-of-nibble n* < *16*
by (*cases n*) *auto*

lemma *nat-of-nibble-div-16*: *nat-of-nibble n div 16 = 0*
by (*cases n*) *auto*

Conversion between chars and nats.

definition
nibble-pair-of-nat :: *nat* ⇒ *nibble* × *nibble* **where**
nibble-pair-of-nat n = (*nibble-of-nat* (*n div 16*), *nibble-of-nat* (*n mod 16*))

lemma *nibble-of-pair* [*code func*]:
nibble-pair-of-nat n = (*nibble-of-nat* (*n div 16*), *nibble-of-nat n*)
unfolding *nibble-of-nat-norm* [*of n*, *symmetric*] *nibble-pair-of-nat-def* **..**

primrec
nat-of-char :: *char* ⇒ *nat* **where**
nat-of-char (*Char n m*) = *nat-of-nibble n* ∗ *16* + *nat-of-nibble m*

lemmas [*simp del*] = *nat-of-char.simps*

definition
char-of-nat :: *nat* ⇒ *char* **where**
char-of-nat-def: *char-of-nat n* = *split Char* (*nibble-pair-of-nat n*)

lemma *Char-char-of-nat*:
Char n m = *char-of-nat* (*nat-of-nibble n* ∗ *16* + *nat-of-nibble m*)
unfolding *char-of-nat-def Let-def nibble-pair-of-nat-def*
by (*auto simp add*: *div-add1-eq mod-add1-eq nat-of-nibble-div-16 nibble-of-nat-norm nibble-of-nat-of-nibble*)

lemma *char-of-nat-of-char*:
char-of-nat (*nat-of-char c*) = *c*
by (*cases c*) (*simp add*: *nat-of-char.simps*, *simp add*: *Char-char-of-nat*)

lemma *nat-of-char-of-nat*:
nat-of-char (*char-of-nat n*) = *n mod 256*
proof −
from *mod-div-equality* [*of n*, *symmetric*, *of 16*]
have *mod-mult-self3*: ⋀*m k n* :: *nat*. (*k* ∗ *n* + *m*) *mod n* = *m mod n*
proof −
fix *m k n* :: *nat*
show (*k* ∗ *n* + *m*) *mod n* = *m mod n*
by (*simp only*: *mod-mult-self1* [*symmetric*, *of m n k*] *add-commute*)
qed
from *mod-div-decomp* [*of n 256*] **obtain** *k l* **where** *n*: *n* = *k* ∗ *256* + *l*
and *k*: *k* = *n div 256* **and** *l*: *l* = *n mod 256* **by** *blast*
have *16*: (*0*::*nat*) < *16* **by** *auto*
have *256*: (*256* :: *nat*) = *16* ∗ *16* **by** *auto*
have *l-256*: *l mod 256* = *l* **using** *l* **by** *auto*
have *l-div-256*: *l div 16* ∗ *16 mod 256* = *l div 16* ∗ *16*

```
    using l by auto
  have aux2: (k * 256 mod 16 + l mod 16) div 16 = 0
    unfolding 256 mult-assoc [symmetric] mod-mult-self-is-0 by simp
  have aux3: (k * 256 + l) div 16 = k * 16 + l div 16
    unfolding div-add1-eq [of k * 256 l 16] aux2 256
      mult-assoc [symmetric] div-mult-self-is-m [OF 16] by simp
  have aux4: (k * 256 + l) mod 16 = l mod 16
    unfolding 256 mult-assoc [symmetric] mod-mult-self3 ..
  show ?thesis
    by (simp add: nat-of-char.simps char-of-nat-def nibble-of-pair
      nat-of-nibble-of-nat mod-mult-distrib
      n aux3 mod-mult-self3 l-256 aux4 mod-add1-eq [of 256 * k] l-div-256)
qed

lemma nibble-pair-of-nat-char:
  nibble-pair-of-nat (nat-of-char (Char n m)) = (n, m)
proof −
  have nat-of-nibble-256:
    ⋀n m. (nat-of-nibble n * 16 + nat-of-nibble m) mod 256 =
      nat-of-nibble n * 16 + nat-of-nibble m
  proof −
    fix n m
    have nat-of-nibble-less-eq-15: ⋀n. nat-of-nibble n ≤ 15
      using Suc-leI [OF nat-of-nibble-less-16] by (auto simp add: nat-number)
    have less-eq-240: nat-of-nibble n * 16 ≤ 240
      using nat-of-nibble-less-eq-15 by auto
    have nat-of-nibble n * 16 + nat-of-nibble m ≤ 240 + 15
      by (rule add-le-mono [of - 240 - 15]) (auto intro: nat-of-nibble-less-eq-15
less-eq-240)
    then have nat-of-nibble n * 16 + nat-of-nibble m < 256 (is ?rhs < -) by auto
    then show ?rhs mod 256 = ?rhs by auto
  qed
  show ?thesis
   unfolding nibble-pair-of-nat-def Char-char-of-nat nat-of-char-of-nat nat-of-nibble-256
   by (simp add: add-commute nat-of-nibble-div-16 nibble-of-nat-norm nibble-of-nat-of-nibble)
qed
```

Code generator setup

```
code-modulename SML
  Char-nat List

code-modulename OCaml
  Char-nat List

code-modulename Haskell
  Char-nat List

end
```

10 Char-ord: Order on characters

theory *Char-ord*
imports *Product-ord Char-nat*
begin

instantiation *nibble* :: *linorder*
begin

definition
nibble-less-eq-def: $n \leq m \longleftrightarrow$ *nat-of-nibble* $n \leq$ *nat-of-nibble* m

definition
nibble-less-def: $n < m \longleftrightarrow$ *nat-of-nibble* $n <$ *nat-of-nibble* m

instance proof
fix n :: *nibble*
show $n \leq n$ **unfolding** *nibble-less-eq-def nibble-less-def* **by** *auto*
next
fix n m q :: *nibble*
assume $n \leq m$
and $m \leq q$
then show $n \leq q$ **unfolding** *nibble-less-eq-def nibble-less-def* **by** *auto*
next
fix n m :: *nibble*
assume $n \leq m$
and $m \leq n$
then show $n = m$
unfolding *nibble-less-eq-def nibble-less-def*
by (*auto simp add*: *nat-of-nibble-eq*)
next
fix n m :: *nibble*
show $n < m \longleftrightarrow n \leq m \wedge n \neq m$
unfolding *nibble-less-eq-def nibble-less-def less-le*
by (*auto simp add*: *nat-of-nibble-eq*)
next
fix n m :: *nibble*
show $n \leq m \vee m \leq n$
unfolding *nibble-less-eq-def* **by** *auto*
qed

end

instantiation *nibble* :: *distrib-lattice*
begin

definition
(*inf* :: *nibble* $\Rightarrow$ -) = *min*

definition
(*sup* :: *nibble* ⇒ -) = *max*

instance by *default* (*auto simp add*:
inf-nibble-def sup-nibble-def min-max.sup-inf-distrib1)

end

instantiation *char* :: *linorder*
begin

definition
char-less-eq-def [*code func del*]: *c1* ≤ *c2* ⟷ (*case c1 of Char n1 m1* ⇒ *case c2 of Char n2 m2* ⇒
n1 < *n2* ∨ *n1* = *n2* ∧ *m1* ≤ *m2*)

definition
char-less-def [*code func del*]: *c1* < *c2* ⟷ (*case c1 of Char n1 m1* ⇒ *case c2 of Char n2 m2* ⇒
n1 < *n2* ∨ *n1* = *n2* ∧ *m1* < *m2*)

instance
by *default* (*auto simp*: *char-less-eq-def char-less-def split*: *char.splits*)

end

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

definition
(*inf* :: *char* ⇒ -) = *min*

definition
(*sup* :: *char* ⇒ -) = *max*

instance **by** *default* (*auto simp add*:
inf-char-def sup-char-def min-max.sup-inf-distrib1)

end

lemma [*simp*, *code func*]:
shows *char-less-eq-simp*: *Char n1 m1* ≤ *Char n2 m2* ⟷ *n1* < *n2* ∨ *n1* = *n2* ∧ *m1* ≤ *m2*
and *char-less-simp*: *Char n1 m1* < *Char n2 m2* ⟷ *n1* < *n2* ∨ *n1* = *n2* ∧ *m1* < *m2*
unfolding *char-less-eq-def char-less-def* **by** *simp-all*

end

11 Code-Char: Code generation of pretty characters (and strings)

theory *Code-Char*
imports *List*
begin

declare *char.recs* [*code func del*] *char.cases* [*code func del*]

lemma [*code func*]:
  size (*c*::*char*) = *0*
  by (*cases c*) *simp*

lemma [*code func*]:
  char-size (*c*::*char*) = *0*
  by (*cases c*) *simp*

code-type *char*
  (*SML char*)
  (*OCaml char*)
  (*Haskell Char*)

setup ⟪
let
  val charr = @{*const-name Char*}
  val nibbles = [@{*const-name Nibble0*}, @{*const-name Nibble1*},
    @{*const-name Nibble2*}, @{*const-name Nibble3*},
    @{*const-name Nibble4*}, @{*const-name Nibble5*},
    @{*const-name Nibble6*}, @{*const-name Nibble7*},
    @{*const-name Nibble8*}, @{*const-name Nibble9*},
    @{*const-name NibbleA*}, @{*const-name NibbleB*},
    @{*const-name NibbleC*}, @{*const-name NibbleD*},
    @{*const-name NibbleE*}, @{*const-name NibbleF*}];
in
  fold (*fn target => CodeTarget.add-pretty-char target charr nibbles*)
    [*SML*, *OCaml*, *Haskell*]
  #> *CodeTarget.add-pretty-list-string Haskell*
    @{*const-name Nil*} @{*const-name Cons*} *charr nibbles*
end
⟫

code-instance *char* :: *eq*
  (*Haskell* −)

code-reserved *SML*
  char

code-reserved *OCaml*
  char

code-const *op* = :: *char* ⇒ *char* ⇒ *bool*
(*SML* !((- : *char*) = -))
(*OCaml* !((- : *char*) = -))
(*Haskell* **infixl** *4* ==)

end

12 Code-Integer: Pretty integer literals for code generation

theory *Code-Integer*
imports *ATP-Linkup*
begin

HOL numeral expressions are mapped to integer literals in target languages, using predefined target language operations for abstract integer operations.

code-type *int*
(*SML IntInf.int*)
(*OCaml Big'-int.big'-int*)
(*Haskell Integer*)

code-instance *int* :: *eq*
(*Haskell* −)

setup ⟪
fold (*Numeral.add-code* @{*const-name number-int-inst.number-of-int*}
true true) [*SML*, *OCaml*, *Haskell*]
⟫

code-const *Int.Pls* **and** *Int.Min* **and** *Int.Bit0* **and** *Int.Bit1*
(*SML raise/ Fail/ Pls*
and *raise/ Fail/ Min*
and !((-);/ *raise/ Fail/ Bit0*)
and !((-);/ *raise/ Fail/ Bit1*))
(*OCaml failwith/ Pls*
and *failwith/ Min*
and !((-);/ *failwith/ Bit0*)
and !((-);/ *failwith/ Bit1*))
(*Haskell error/ Pls*
and *error/ Min*
and *error/ Bit0*
and *error/ Bit1*)

code-const *Int.pred*
(*SML IntInf.*− ((-), *1*))

```
  (OCaml Big'-int.pred'-big'-int)
  (Haskell !(-/ -/ 1))

code-const Int.succ
  (SML IntInf.+ ((-), 1))
  (OCaml Big'-int.succ'-big'-int)
  (Haskell !(-/ +/ 1))

code-const op + :: int ⇒ int ⇒ int
  (SML IntInf.+ ((-), (-)))
  (OCaml Big'-int.add'-big'-int)
  (Haskell infixl 6 +)

code-const uminus :: int ⇒ int
  (SML IntInf.~)
  (OCaml Big'-int.minus'-big'-int)
  (Haskell negate)

code-const op - :: int ⇒ int ⇒ int
  (SML IntInf.- ((-), (-)))
  (OCaml Big'-int.sub'-big'-int)
  (Haskell infixl 6 -)

code-const op * :: int ⇒ int ⇒ int
  (SML IntInf.* ((-), (-)))
  (OCaml Big'-int.mult'-big'-int)
  (Haskell infixl 7 *)

code-const op = :: int ⇒ int ⇒ bool
  (SML !((- : IntInf.int) = -))
  (OCaml Big'-int.eq'-big'-int)
  (Haskell infixl 4 ==)

code-const op ≤ :: int ⇒ int ⇒ bool
  (SML IntInf.<= ((-), (-)))
  (OCaml Big'-int.le'-big'-int)
  (Haskell infix 4 <=)

code-const op < :: int ⇒ int ⇒ bool
  (SML IntInf.< ((-), (-)))
  (OCaml Big'-int.lt'-big'-int)
  (Haskell infix 4 <)

code-reserved SML IntInf
code-reserved OCaml Big-int

end
```

13 Code-Char-chr: Code generation of pretty characters with character codes

theory *Code-Char-chr*
imports *Char-nat Code-Char Code-Integer*
begin

definition
int-of-char = int o nat-of-char

lemma [*code func*]:
nat-of-char = nat o int-of-char
unfolding *int-of-char-def* **by** (*simp add*: *expand-fun-eq*)

definition
char-of-int = char-of-nat o nat

lemma [*code func*]:
char-of-nat = char-of-int o int
unfolding *char-of-int-def* **by** (*simp add*: *expand-fun-eq*)

lemmas [*code func del*] = *char.recs char.cases char.size*

lemma [*code func*, *code inline*]:
char-rec f c = split f (*nibble-pair-of-nat* (*nat-of-char c*))
by (*cases c*) (*auto simp add*: *nibble-pair-of-nat-char*)

lemma [*code func*, *code inline*]:
char-case f c = split f (*nibble-pair-of-nat* (*nat-of-char c*))
by (*cases c*) (*auto simp add*: *nibble-pair-of-nat-char*)

lemma [*code func*]:
size (*c*::*char*) = *0*
by (*cases c*) *auto*

code-const *int-of-char* **and** *char-of-int*
(*SML* !(*IntInf.fromInt o Char.ord*) **and** !(*Char.chr o IntInf.toInt*))
(*OCaml Big'-int.big'-int'-of'-int* (*Char.code* -) **and** *Char.chr* (*Big'-int.int'-of'-big'-int* -))
(*Haskell toInteger* (*fromEnum* (- :: *Char*)) **and** !(*let chr k* | *k* < *256* = *toEnum k* :: *Char in chr* . *fromInteger*))

end

14 Code-Index: Type of indices

theory *Code-Index*
imports *ATP-Linkup*

begin

Indices are isomorphic to HOL *nat* but mapped to target-language builtin integers

14.1 Datatype of indices

typedef *index* = *UNIV* :: *nat set*
  morphisms *nat-of-index index-of-nat* **by** *rule*

lemma *index-of-nat-nat-of-index* [*simp*]:
  index-of-nat (*nat-of-index k*) = *k*
  by (*rule nat-of-index-inverse*)

lemma *nat-of-index-index-of-nat* [*simp*]:
  nat-of-index (*index-of-nat n*) = *n*
  by (*rule index-of-nat-inverse*)
    (*unfold index-def*, *rule UNIV-I*)

lemma *index*:
  (⋀*n*::*index*. *PROP P n*) ≡ (⋀*n*::*nat*. *PROP P* (*index-of-nat n*))
proof
  fix *n* :: *nat*
  assume ⋀*n*::*index*. *PROP P n*
  then show *PROP P* (*index-of-nat n*) **.**
next
  fix *n* :: *index*
  assume ⋀*n*::*nat*. *PROP P* (*index-of-nat n*)
  then have *PROP P* (*index-of-nat* (*nat-of-index n*)) **.**
  then show *PROP P n* **by** *simp*
qed

lemma *index-case*:
  assumes ⋀*n*. *k* = *index-of-nat n* ⟹ *P*
  shows *P*
  by (*rule assms* [*of nat-of-index k*]) *simp*

lemma *index-induct-raw*:
  assumes ⋀*n*. *P* (*index-of-nat n*)
  shows *P k*
proof −
  from *assms* **have** *P* (*index-of-nat* (*nat-of-index k*)) **.**
  then show *?thesis* **by** *simp*
qed

lemma *nat-of-index-inject* [*simp*]:
  nat-of-index k = *nat-of-index l* ⟷ *k* = *l*
  by (*rule nat-of-index-inject*)

lemma *index-of-nat-inject* [*simp*]:
index-of-nat n = *index-of-nat m* ⟷ *n* = *m*
by (*auto intro*!: *index-of-nat-inject simp add*: *index-def*)

instantiation *index* :: *zero*
begin

definition [*simp*, *code func del*]:
0 = *index-of-nat 0*

instance ..

end

definition [*simp*]:
Suc-index k = *index-of-nat* (*Suc* (*nat-of-index k*))

lemma *index-induct*: *P 0* ⟹ (⋀*k*. *P k* ⟹ *P* (*Suc-index k*)) ⟹ *P k*
proof −
assume *P 0* **then have** *init*: *P* (*index-of-nat 0*) **by** *simp*
assume ⋀*k*. *P k* ⟹ *P* (*Suc-index k*)
then have ⋀*n*. *P* (*index-of-nat n*) ⟹ *P* (*Suc-index* (*index-of-nat* (*n*))) .
then have *step*: ⋀*n*. *P* (*index-of-nat n*) ⟹ *P* (*index-of-nat* (*Suc n*)) **by** *simp*
from *init step* **have** *P* (*index-of-nat* (*nat-of-index k*))
by (*induct nat-of-index k*) *simp-all*
then show *P k* **by** *simp*
qed

lemma *Suc-not-Zero-index*: *Suc-index k* ≠ *0*
by *simp*

lemma *Zero-not-Suc-index*: *0* ≠ *Suc-index k*
by *simp*

lemma *Suc-Suc-index-eq*: *Suc-index k* = *Suc-index l* ⟷ *k* = *l*
by *simp*

rep-datatype *index*
distinct *Suc-not-Zero-index Zero-not-Suc-index*
inject *Suc-Suc-index-eq*
induction *index-induct*

lemmas [*code func del*] = *index.recs index.cases*

declare *index-case* [*case-names nat*, *cases type*: *index*]
declare *index-induct* [*case-names nat*, *induct type*: *index*]

lemma [*code func*]:
index-size = *nat-of-index*

proof (*rule ext*)
fix *k*
have *index-size k = nat-size* (*nat-of-index k*)
by (*induct k rule*: *index.induct*) (*simp-all del*: *zero-index-def Suc-index-def*, *simp-all*)
also have *nat-size* (*nat-of-index k*) = *nat-of-index k* **by** (*induct nat-of-index k*) *simp-all*
finally show *index-size k = nat-of-index k* **.**
qed

lemma [*code func*]:
size = nat-of-index
proof (*rule ext*)
fix *k*
show *size k = nat-of-index k*
by (*induct k*) (*simp-all del*: *zero-index-def Suc-index-def*, *simp-all*)
qed

lemma [*code func*]:
k = l ⟷ *nat-of-index k = nat-of-index l*
by (*cases k*, *cases l*) *simp*

14.2 Indices as datatype of ints

instantiation *index* :: *number*
begin

definition
number-of = index-of-nat o nat

instance ..

end

lemma *nat-of-index-number* [*simp*]:
nat-of-index (*number-of k*) = *number-of k*
by (*simp add*: *number-of-index-def nat-number-of-def number-of-is-id*)

code-datatype *number-of* :: *int* ⇒ *index*

14.3 Basic arithmetic

instantiation *index* :: {*minus*, *ordered-semidom*, *Divides.div*, *linorder*}
begin

lemma *zero-index-code* [*code inline*, *code func*]:
(*0*::*index*) = *Numeral0*
by (*simp add*: *number-of-index-def Pls-def*)
lemma [*code post*]: *Numeral0* = (*0*::*index*)
using *zero-index-code* **..**

definition [*simp*, *code func del*]:
(*1*::*index*) = *index-of-nat 1*

lemma *one-index-code* [*code inline*, *code func*]:
(*1*::*index*) = *Numeral1*
by (*simp add*: *number-of-index-def Pls-def Bit1-def*)
lemma [*code post*]: *Numeral1* = (*1*::*index*)
using *one-index-code* **..**

definition [*simp*, *code func del*]:
n + *m* = *index-of-nat* (*nat-of-index n* + *nat-of-index m*)

lemma *plus-index-code* [*code func*]:
index-of-nat n + *index-of-nat m* = *index-of-nat* (*n* + *m*)
by *simp*

definition [*simp*, *code func del*]:
n − *m* = *index-of-nat* (*nat-of-index n* − *nat-of-index m*)

definition [*simp*, *code func del*]:
n ∗ *m* = *index-of-nat* (*nat-of-index n* ∗ *nat-of-index m*)

lemma *times-index-code* [*code func*]:
index-of-nat n ∗ *index-of-nat m* = *index-of-nat* (*n* ∗ *m*)
by *simp*

definition [*simp*, *code func del*]:
n div m = *index-of-nat* (*nat-of-index n div nat-of-index m*)

definition [*simp*, *code func del*]:
n mod m = *index-of-nat* (*nat-of-index n mod nat-of-index m*)

lemma *div-index-code* [*code func*]:
index-of-nat n div index-of-nat m = *index-of-nat* (*n div m*)
by *simp*

lemma *mod-index-code* [*code func*]:
index-of-nat n mod index-of-nat m = *index-of-nat* (*n mod m*)
by *simp*

definition [*simp*, *code func del*]:
n ≤ *m* ⟷ *nat-of-index n* ≤ *nat-of-index m*

definition [*simp*, *code func del*]:
n < *m* ⟷ *nat-of-index n* < *nat-of-index m*

lemma *less-eq-index-code* [*code func*]:
index-of-nat n ≤ *index-of-nat m* ⟷ *n* ≤ *m*

by *simp*

lemma *less-index-code* [*code func*]:
index-of-nat n < *index-of-nat m* ⟷ *n* < *m*
by *simp*

instance by *default* (*auto simp add*: *left-distrib index*)

end

lemma *Suc-index-minus-one*: *Suc-index n* − *1* = *n* **by** *simp*

lemma *index-of-nat-code* [*code*]:
index-of-nat = *of-nat*
proof
fix *n* :: *nat*
have *of-nat n* = *index-of-nat n*
by (*induct n*) *simp-all*
then show *index-of-nat n* = *of-nat n*
by (*rule sym*)
qed

lemma *index-not-eq-zero*: *i* ≠ *index-of-nat 0* ⟷ *i* ≥ *1*
by (*cases i*) *auto*

definition
nat-of-index-aux :: *index* ⇒ *nat* ⇒ *nat*
where
nat-of-index-aux i n = *nat-of-index i* + *n*

lemma *nat-of-index-aux-code* [*code*]:
nat-of-index-aux i n = (*if i* = *0 then n else nat-of-index-aux* (*i* − *1*) (*Suc n*))
by (*auto simp add*: *nat-of-index-aux-def index-not-eq-zero*)

lemma *nat-of-index-code* [*code*]:
nat-of-index i = *nat-of-index-aux i 0*
by (*simp add*: *nat-of-index-aux-def*)

14.4 ML interface

ML ⟪
structure Index =
struct

fun mk k = *HOLogic.mk-number* @{*typ index*} *k*;

end;
⟫

14.5 Specialized *op −*, *op div* and *op mod* operations

definition
minus-index-aux :: *index* ⇒ *index* ⇒ *index*
where
[*code func del*]: *minus-index-aux* = *op −*

lemma [*code func*]: *op −* = *minus-index-aux*
using *minus-index-aux-def* **..**

definition
div-mod-index :: *index* ⇒ *index* ⇒ *index* × *index*
where
[*code func del*]: *div-mod-index n m* = (*n div m*, *n mod m*)

lemma [*code func*]:
div-mod-index n m = (*if m* = *0 then* (*0*, *n*) *else* (*n div m*, *n mod m*))
unfolding *div-mod-index-def* **by** *auto*

lemma [*code func*]:
n div m = *fst* (*div-mod-index n m*)
unfolding *div-mod-index-def* **by** *simp*

lemma [*code func*]:
n mod m = *snd* (*div-mod-index n m*)
unfolding *div-mod-index-def* **by** *simp*

14.6 Code serialization

Implementation of indices by bounded integers

code-type *index*
(*SML int*)
(*OCaml int*)
(*Haskell Int*)

code-instance *index* :: *eq*
(*Haskell −*)

setup ⟪
fold (*Numeral.add-code* @{*const-name number-index-inst.number-of-index*}
false false) [*SML*, *OCaml*, *Haskell*]
⟫

code-reserved *SML Int int*
code-reserved *OCaml Pervasives int*

code-const *op +* :: *index* ⇒ *index* ⇒ *index*
(*SML Int.+/* ((-),/ (-)))
(*OCaml Pervasives.*(+))

(*Haskell* **infixl** *6* +)

code-const *minus-index-aux* :: *index* ⇒ *index* ⇒ *index*
(*SML Int.max*/ (-/ −/ -,/ *0* : *int*))
(*OCaml Pervasives.max*/ (-/ −/ -)/ (*0* : *int*))
(*Haskell max*/ (-/ −/ -)/ (*0* :: *Int*))

code-const *op* ∗ :: *index* ⇒ *index* ⇒ *index*
(*SML Int.*∗/ ((-),/ (-)))
(*OCaml Pervasives.*(∗))
(*Haskell* **infixl** *7* ∗)

code-const *div-mod-index*
(*SML* (*fn n* => *fn m* =>/ (*n div m*, *n mod m*)))
(*OCaml* (*fun n* -> *fun m* ->/ (*n* '/ *m*, *n mod m*)))
(*Haskell divMod*)

code-const *op* = :: *index* ⇒ *index* ⇒ *bool*
(*SML* !((- : *Int.int*) = -))
(*OCaml* !((- : *int*) = -))
(*Haskell* **infixl** *4* ==)

code-const *op* ≤ :: *index* ⇒ *index* ⇒ *bool*
(*SML Int.*<=/ ((-),/ (-)))
(*OCaml* !((- : *int*) <= -))
(*Haskell* **infix** *4* <=)

code-const *op* < :: *index* ⇒ *index* ⇒ *bool*
(*SML Int.*</ ((-),/ (-)))
(*OCaml* !((- : *int*) < -))
(*Haskell* **infix** *4* <)

end

15 Code-Message: Monolithic strings (message strings) for code generation

theory *Code-Message*
imports *List*
begin

15.1 Datatype of messages

datatype *message-string* = *STR string*

lemmas [*code func del*] = *message-string.recs message-string.cases*

lemma [*code func*]: *size* (*s*::*message-string*) = *0*
by (*cases s*) *simp-all*

lemma [*code func*]: *message-string-size* (*s*::*message-string*) = *0*
by (*cases s*) *simp-all*

15.2 ML interface

ML ⟪
structure Message-String =
struct

fun mk s = @{*term STR*} $ *HOLogic.mk-string s*;

end;
⟫

15.3 Code serialization

code-type *message-string*
(*SML string*)
(*OCaml string*)
(*Haskell String*)

setup ⟪
let
val charr = @{*const-name Char*}
val nibbles = [@{*const-name Nibble0*}, @{*const-name Nibble1*},
@{*const-name Nibble2*}, @{*const-name Nibble3*},
@{*const-name Nibble4*}, @{*const-name Nibble5*},
@{*const-name Nibble6*}, @{*const-name Nibble7*},
@{*const-name Nibble8*}, @{*const-name Nibble9*},
@{*const-name NibbleA*}, @{*const-name NibbleB*},
@{*const-name NibbleC*}, @{*const-name NibbleD*},
@{*const-name NibbleE*}, @{*const-name NibbleF*}];
in
fold (*fn target => CodeTarget.add-pretty-message target*
charr nibbles @{*const-name Nil*} @{*const-name Cons*} @{*const-name STR*})
[*SML*, *OCaml*, *Haskell*]
end
⟫

code-reserved *SML string*
code-reserved *OCaml string*

code-instance *message-string* :: *eq*
(*Haskell* −)

code-const *op* = :: *message-string* ⇒ *message-string* ⇒ *bool*
(*SML* !((- : *string*) = -))

(*OCaml* !((- : *string*) = -))
(*Haskell* **infixl** *4* ==)

end

16 Coinductive-List: Potentially infinite lists as greatest fixed-point

theory *Coinductive-List*
imports *List*
begin

16.1 List constructors over the datatype universe

definition *NIL* = *Datatype.In0* (*Datatype.Numb 0*)
definition *CONS M N* = *Datatype.In1* (*Datatype.Scons M N*)

lemma *CONS-not-NIL* [*iff*]: *CONS M N* ≠ *NIL*
and *NIL-not-CONS* [*iff*]: *NIL* ≠ *CONS M N*
and *CONS-inject* [*iff*]: (*CONS K M*) = (*CONS L N*) = (*K* = *L* ∧ *M* = *N*)
by (*simp-all add*: *NIL-def CONS-def*)

lemma *CONS-mono*: *M* ⊆ *M'* ⟹ *N* ⊆ *N'* ⟹ *CONS M N* ⊆ *CONS M' N'*
by (*simp add*: *CONS-def In1-mono Scons-mono*)

lemma *CONS-UN1*: *CONS M* (⋃*x*. *f x*) = (⋃*x*. *CONS M* (*f x*))
— A continuity result?
by (*simp add*: *CONS-def In1-UN1 Scons-UN1-y*)

definition *List-case c h* = *Datatype.Case* (λ-. *c*) (*Datatype.Split h*)

lemma *List-case-NIL* [*simp*]: *List-case c h NIL* = *c*
and *List-case-CONS* [*simp*]: *List-case c h* (*CONS M N*) = *h M N*
by (*simp-all add*: *List-case-def NIL-def CONS-def*)

16.2 Corecursive lists

coinductive-set *LList* **for** *A*
where *NIL* [*intro*]: *NIL* ∈ *LList A*
| *CONS* [*intro*]: *a* ∈ *A* ⟹ *M* ∈ *LList A* ⟹ *CONS a M* ∈ *LList A*

lemma *LList-mono*:
assumes *subset*: *A* ⊆ *B*
shows *LList A* ⊆ *LList B*
— This justifies using *LList* in other recursive type definitions.
proof
fix *x*
assume *x* ∈ *LList A*

```
  then show x ∈ LList B
  proof coinduct
    case LList
    then show ?case using subset
      by cases blast+
  qed
qed

consts
  LList-corec-aux :: nat ⇒ ('a ⇒ ('b Datatype.item × 'a) option) ⇒
    'a ⇒ 'b Datatype.item
primrec
  LList-corec-aux 0 f x = {}
  LList-corec-aux (Suc k) f x =
    (case f x of
      None ⇒ NIL
    | Some (z, w) ⇒ CONS z (LList-corec-aux k f w))

definition LList-corec a f = (⋃k. LList-corec-aux k f a)
```

Note: the subsequent recursion equation for *LList-corec* may be used with the Simplifier, provided it operates in a non-strict fashion for case expressions (i.e. the usual *case* congruence rule needs to be present).

```
lemma LList-corec:
  LList-corec a f =
    (case f a of None ⇒ NIL | Some (z, w) ⇒ CONS z (LList-corec w f))
  (is ?lhs = ?rhs)
proof
  show ?lhs ⊆ ?rhs
    apply (unfold LList-corec-def)
    apply (rule UN-least)
    apply (case-tac k)
     apply (simp-all (no-asm-simp) split: option.splits)
   apply (rule allI impI subset-refl [THEN CONS-mono] UNIV-I [THEN UN-upper])+
    done
  show ?rhs ⊆ ?lhs
    apply (simp add: LList-corec-def split: option.splits)
    apply (simp add: CONS-UN1)
    apply safe
     apply (rule-tac a = Suc ?k in UN-I, simp, simp)+
    done
qed

lemma LList-corec-type: LList-corec a f ∈ LList UNIV
proof −
  have ∃x. LList-corec a f = LList-corec x f by blast
  then show ?thesis
  proof coinduct
    case (LList L)
```

```
    then obtain x where L: L = LList-corec x f by blast
    show ?case
    proof (cases f x)
      case None
      then have LList-corec x f = NIL
        by (simp add: LList-corec)
      with L have ?NIL by simp
      then show ?thesis ..
    next
      case (Some p)
      then have LList-corec x f = CONS (fst p) (LList-corec (snd p) f)
        by (simp add: LList-corec split: prod.split)
      with L have ?CONS by auto
      then show ?thesis ..
    qed
  qed
qed
```

16.3 Abstract type definition

```
typedef 'a llist = LList (range Datatype.Leaf) :: 'a Datatype.item set
proof
  show NIL ∈ ?llist ..
qed

lemma NIL-type: NIL ∈ llist
  unfolding llist-def by (rule LList.NIL)

lemma CONS-type: a ∈ range Datatype.Leaf ⟹
    M ∈ llist ⟹ CONS a M ∈ llist
  unfolding llist-def by (rule LList.CONS)

lemma llistI: x ∈ LList (range Datatype.Leaf) ⟹ x ∈ llist
  by (simp add: llist-def)

lemma llistD: x ∈ llist ⟹ x ∈ LList (range Datatype.Leaf)
  by (simp add: llist-def)

lemma Rep-llist-UNIV: Rep-llist x ∈ LList UNIV
proof −
  have Rep-llist x ∈ llist by (rule Rep-llist)
  then have Rep-llist x ∈ LList (range Datatype.Leaf)
    by (simp add: llist-def)
  also have ... ⊆ LList UNIV by (rule LList-mono) simp
  finally show ?thesis .
qed

definition LNil = Abs-llist NIL
definition LCons x xs = Abs-llist (CONS (Datatype.Leaf x) (Rep-llist xs))
```

lemma *LCons-not-LNil* [*iff*]: *LCons x xs* ≠ *LNil*
apply (*simp add*: *LNil-def LCons-def*)
apply (*subst Abs-llist-inject*)
apply (*auto intro*: *NIL-type CONS-type Rep-llist*)
done

lemma *LNil-not-LCons* [*iff*]: *LNil* ≠ *LCons x xs*
by (*rule LCons-not-LNil* [*symmetric*])

lemma *LCons-inject* [*iff*]: (*LCons x xs* = *LCons y ys*) = (*x* = *y* ∧ *xs* = *ys*)
apply (*simp add*: *LCons-def*)
apply (*subst Abs-llist-inject*)
apply (*auto simp add*: *Rep-llist-inject intro*: *CONS-type Rep-llist*)
done

lemma *Rep-llist-LNil*: *Rep-llist LNil* = *NIL*
by (*simp add*: *LNil-def add*: *Abs-llist-inverse NIL-type*)

lemma *Rep-llist-LCons*: *Rep-llist* (*LCons x l*) =
CONS (*Datatype.Leaf x*) (*Rep-llist l*)
by (*simp add*: *LCons-def Abs-llist-inverse CONS-type Rep-llist*)

lemma *llist-cases* [*cases type*: *llist*]:
obtains
(*LNil*) *l* = *LNil*
| (*LCons*) *x l'* **where** *l* = *LCons x l'*
proof (*cases l*)
case (*Abs-llist L*)
from ‹*L* ∈ *llist*› **have** *L* ∈ *LList* (*range Datatype.Leaf*) **by** (*rule llistD*)
then show *?thesis*
proof *cases*
case *NIL*
with *Abs-llist* **have** *l* = *LNil* **by** (*simp add*: *LNil-def*)
with *LNil* **show** *?thesis* **.**
next
case (*CONS a K*)
then have *K* ∈ *llist* **by** (*blast intro*: *llistI*)
then obtain *l'* **where** *K* = *Rep-llist l'* **by** *cases*
with *CONS* **and** *Abs-llist* **obtain** *x* **where** *l* = *LCons x l'*
by (*auto simp add*: *LCons-def Abs-llist-inject*)
with *LCons* **show** *?thesis* **.**
qed
qed

definition
llist-case c d l =
List-case c (λ*x y*. *d* (*inv Datatype.Leaf x*) (*Abs-llist y*)) (*Rep-llist l*)

syntax
LNil :: *logic*
LCons :: *logic*
translations
case p of LNil ⇒ a | LCons x l ⇒ b ⇌ CONST llist-case a (λx l. b) p

lemma *llist-case-LNil* [*simp*]: *llist-case c d LNil = c*
by (*simp add*: *llist-case-def LNil-def*
NIL-type Abs-llist-inverse)

lemma *llist-case-LCons* [*simp*]: *llist-case c d (LCons M N) = d M N*
by (*simp add*: *llist-case-def LCons-def*
CONS-type Abs-llist-inverse Rep-llist Rep-llist-inverse inj-Leaf)

definition
llist-corec a f =
Abs-llist (*LList-corec a*
(λ*z*.
case f z of None ⇒ None
| *Some* (*v*, *w*) ⇒ *Some* (*Datatype.Leaf v*, *w*)))

lemma *LList-corec-type2*:
LList-corec a
(λ*z*. *case f z of None ⇒ None*
| *Some* (*v*, *w*) ⇒ *Some* (*Datatype.Leaf v*, *w*)) ∈ *llist*
(**is** *?corec a* ∈ -)
proof (*unfold llist-def*)
let *LList-corec a ?g = ?corec a*
have ∃ *x*. *?corec a = ?corec x* **by** *blast*
then show *?corec a* ∈ *LList* (*range Datatype.Leaf*)
proof *coinduct*
case (*LList L*)
then obtain *x* **where** *L*: *L = ?corec x* **by** *blast*
show *?case*
proof (*cases f x*)
case *None*
then have *?corec x = NIL*
by (*simp add*: *LList-corec*)
with *L* **have** *?NIL* **by** *simp*
then show *?thesis* **..**
next
case (*Some p*)
then have *?corec x =*
CONS (*Datatype.Leaf* (*fst p*)) (*?corec* (*snd p*))
by (*simp add*: *LList-corec split*: *prod.split*)
with *L* **have** *?CONS* **by** *auto*
then show *?thesis* **..**

```
    qed
  qed
qed

lemma llist-corec:
  llist-corec a f =
    (case f a of None ⇒ LNil | Some (z, w) ⇒ LCons z (llist-corec w f))
proof (cases f a)
  case None
  then show ?thesis
    by (simp add: llist-corec-def LList-corec LNil-def)
next
  case (Some p)

  let ?corec a = llist-corec a f
  let ?rep-corec a =
    LList-corec a
      (λz. case f z of None ⇒ None
        | Some (v, w) ⇒ Some (Datatype.Leaf v, w))

  have ?corec a = Abs-llist (?rep-corec a)
    by (simp only: llist-corec-def)
  also from Some have ?rep-corec a =
      CONS (Datatype.Leaf (fst p)) (?rep-corec (snd p))
    by (simp add: LList-corec split: prod.split)
  also have ?rep-corec (snd p) = Rep-llist (?corec (snd p))
    by (simp only: llist-corec-def Abs-llist-inverse LList-corec-type2)
  finally have ?corec a = LCons (fst p) (?corec (snd p))
    by (simp only: LCons-def)
  with Some show ?thesis by (simp split: prod.split)
qed
```

16.4 Equality as greatest fixed-point – the bisimulation principle

```
coinductive-set EqLList for r
where EqNIL: (NIL, NIL) ∈ EqLList r
  | EqCONS: (a, b) ∈ r ⟹ (M, N) ∈ EqLList r ⟹
      (CONS a M, CONS b N) ∈ EqLList r

lemma EqLList-unfold:
    EqLList r = dsum (diag {Datatype.Numb 0}) (dprod r (EqLList r))
  by (fast intro!: EqLList.intros [unfolded NIL-def CONS-def]
          elim: EqLList.cases [unfolded NIL-def CONS-def])

lemma EqLList-implies-ntrunc-equality:
    (M, N) ∈ EqLList (diag A) ⟹ ntrunc k M = ntrunc k N
  apply (induct k arbitrary: M N rule: nat-less-induct)
  apply (erule EqLList.cases)
```

```
  apply (safe del: equalityI)
 apply (case-tac n)
  apply simp
 apply (rename-tac n')
 apply (case-tac n')
  apply (simp-all add: CONS-def less-Suc-eq)
 done

lemma Domain-EqLList: Domain (EqLList (diag A)) ⊆ LList A
 apply (rule subsetI)
 apply (erule LList.coinduct)
 apply (subst (asm) EqLList-unfold)
 apply (auto simp add: NIL-def CONS-def)
 done

lemma EqLList-diag: EqLList (diag A) = diag (LList A)
 (is ?lhs = ?rhs)
proof
 show ?lhs ⊆ ?rhs
  apply (rule subsetI)
  apply (rule-tac p = x in PairE)
  apply clarify
  apply (rule diag-eqI)
   apply (rule EqLList-implies-ntrunc-equality [THEN ntrunc-equality],
     assumption)
  apply (erule DomainI [THEN Domain-EqLList [THEN subsetD]])
  done
 {
  fix M N assume (M, N) ∈ diag (LList A)
  then have (M, N) ∈ EqLList (diag A)
  proof coinduct
   case (EqLList M N)
   then obtain L where L: L ∈ LList A and MN: M = L N = L by blast
   from L show ?case
   proof cases
    case NIL with MN have ?EqNIL by simp
    then show ?thesis ..
   next
    case CONS with MN have ?EqCONS by (simp add: diagI)
    then show ?thesis ..
   qed
  qed
 }
 then show ?rhs ⊆ ?lhs by auto
qed

lemma EqLList-diag-iff [iff]: (p ∈ EqLList (diag A)) = (p ∈ diag (LList A))
 by (simp only: EqLList-diag)
```

To show two LLists are equal, exhibit a bisimulation! (Also admits true

equality.)

lemma *LList-equalityI*
[*consumes 1*, *case-names EqLList*, *case-conclusion EqLList EqNIL EqCONS*]:
assumes *r*: $(M, N) \in r$
and *step*: $\bigwedge M\ N.\ (M, N) \in r \Longrightarrow$
$M = NIL \wedge N = NIL \vee$
$(\exists\, a\ b\ M'\ N'.$
$M = CONS\ a\ M' \wedge N = CONS\ b\ N' \wedge (a, b) \in diag\ A \wedge$
$((M', N') \in r \vee (M', N') \in EqLList\ (diag\ A)))$
shows $M = N$
proof −
from *r* **have** $(M, N) \in EqLList\ (diag\ A)$
proof *coinduct*
case *EqLList*
then show *?case* **by** (*rule step*)
qed
then show *?thesis* **by** *auto*
qed

lemma *LList-fun-equalityI*
[*consumes 1*, *case-names NIL-type NIL CONS*, *case-conclusion CONS EqNIL EqCONS*]:
assumes *M*: $M \in LList\ A$
and *fun-NIL*: $g\ NIL \in LList\ A \quad f\ NIL = g\ NIL$
and *fun-CONS*: $\bigwedge x\ l.\ x \in A \Longrightarrow l \in LList\ A \Longrightarrow$
$(f\ (CONS\ x\ l), g\ (CONS\ x\ l)) = (NIL, NIL) \vee$
$(\exists\, M\ N\ a\ b.$
$(f\ (CONS\ x\ l), g\ (CONS\ x\ l)) = (CONS\ a\ M, CONS\ b\ N) \wedge$
$(a, b) \in diag\ A \wedge$
$(M, N) \in \{(f\ u, g\ u) \mid u.\ u \in LList\ A\} \cup diag\ (LList\ A))$
(**is** $\bigwedge x\ l.$ - $\Longrightarrow$ - $\Longrightarrow$ *?fun-CONS x l*)
shows $f\ M = g\ M$
proof −
let *?bisim* = $\{(f\ L, g\ L) \mid L.\ L \in LList\ A\}$
have $(f\ M, g\ M) \in$ *?bisim* **using** *M* **by** *blast*
then show *?thesis*
proof (*coinduct taking*: *A rule*: *LList-equalityI*)
case (*EqLList M N*)
then obtain *L* **where** *MN*: $M = f\ L\ N = g\ L$ **and** *L*: $L \in LList\ A$ **by** *blast*
from *L* **show** *?case*
proof (*cases L*)
case *NIL*
with *fun-NIL* **and** *MN* **have** $(M, N) \in diag\ (LList\ A)$ **by** *auto*
then have $(M, N) \in EqLList\ (diag\ A)$ **..**
then show *?thesis* **by** *cases simp-all*
next
case (*CONS a K*)
from *fun-CONS* **and** ‹$a \in A$› ‹$K \in LList\ A$›
have *?fun-CONS a K* (**is** *?NIL* ∨ *?CONS*) **.**

```
      then show ?thesis
      proof
        assume ?NIL
        with MN CONS have (M, N) ∈ diag (LList A) by auto
        then have (M, N) ∈ EqLList (diag A) ..
        then show ?thesis by cases simp-all
      next
        assume ?CONS
        with CONS obtain a b M' N' where
            fg: (f L, g L) = (CONS a M', CONS b N')
          and ab: (a, b) ∈ diag A
          and M'N': (M', N') ∈ ?bisim ∪ diag (LList A)
          by blast
        from M'N' show ?thesis
        proof
          assume (M', N') ∈ ?bisim
          with MN fg ab show ?thesis by simp
        next
          assume (M', N') ∈ diag (LList A)
          then have (M', N') ∈ EqLList (diag A) ..
          with MN fg ab show ?thesis by simp
        qed
      qed
    qed
  qed
qed
```

Finality of *llist A*: Uniqueness of functions defined by corecursion.

```
lemma equals-LList-corec:
  assumes h: ⋀x. h x =
    (case f x of None ⇒ NIL | Some (z, w) ⇒ CONS z (h w))
  shows h x = (λx. LList-corec x f) x
proof −
  def h' ≡ λx. LList-corec x f
  then have h': ⋀x. h' x =
      (case f x of None ⇒ NIL | Some (z, w) ⇒ CONS z (h' w))
    unfolding h'-def by (simp add: LList-corec)
  have (h x, h' x) ∈ {(h u, h' u) | u. True} by blast
  then show h x = h' x
  proof (coinduct taking: UNIV rule: LList-equalityI)
    case (EqLList M N)
    then obtain x where MN: M = h x N = h' x by blast
    show ?case
    proof (cases f x)
      case None
      with h h' MN have ?EqNIL by simp
      then show ?thesis ..
    next
      case (Some p)
```

```
    with h h' MN have M = CONS (fst p) (h (snd p))
     and N = CONS (fst p) (h' (snd p))
     by (simp-all split: prod.split)
    then have ?EqCONS by (auto iff: diag-iff)
    then show ?thesis ..
  qed
 qed
qed


lemma llist-equalityI
 [consumes 1, case-names Eqllist, case-conclusion Eqllist EqLNil EqLCons]:
 assumes r: (l1, l2) ∈ r
  and step: ⋀q. q ∈ r ⟹
   q = (LNil, LNil) ∨
    (∃ l1 l2 a b.
      q = (LCons a l1, LCons b l2) ∧ a = b ∧
       ((l1, l2) ∈ r ∨ l1 = l2))
    (is ⋀q. - ⟹ ?EqLNil q ∨ ?EqLCons q)
 shows l1 = l2
proof −
 def M ≡ Rep-llist l1 and N ≡ Rep-llist l2
 with r have (M, N) ∈ {(Rep-llist l1, Rep-llist l2) | l1 l2. (l1, l2) ∈ r}
  by blast
 then have M = N
 proof (coinduct taking: UNIV rule: LList-equalityI)
  case (EqLList M N)
  then obtain l1 l2 where
     MN: M = Rep-llist l1 N = Rep-llist l2 and r: (l1, l2) ∈ r
    by auto
  from step [OF r] show ?case
  proof
    assume ?EqLNil (l1, l2)
    with MN have ?EqNIL by (simp add: Rep-llist-LNil)
    then show ?thesis ..
  next
    assume ?EqLCons (l1, l2)
    with MN have ?EqCONS
     by (force simp add: Rep-llist-LCons EqLList-diag intro: Rep-llist-UNIV)
    then show ?thesis ..
  qed
 qed
 then show ?thesis by (simp add: M-def N-def Rep-llist-inject)
qed

lemma llist-fun-equalityI
 [case-names LNil LCons, case-conclusion LCons EqLNil EqLCons]:
 assumes fun-LNil: f LNil = g LNil
  and fun-LCons: ⋀x l.
```

```
    (f (LCons x l), g (LCons x l)) = (LNil, LNil) ∨
     (∃ l1 l2 a b.
       (f (LCons x l), g (LCons x l)) = (LCons a l1, LCons b l2) ∧
         a = b ∧ ((l1, l2) ∈ {(f u, g u) | u. True} ∨ l1 = l2))
    (is ⋀x l. ?fun-LCons x l)
  shows f l = g l
proof −
  have (f l, g l) ∈ {(f l, g l) | l. True} by blast
  then show ?thesis
  proof (coinduct rule: llist-equalityI)
    case (Eqllist q)
    then obtain l where q: q = (f l, g l) by blast
    show ?case
    proof (cases l)
      case LNil
      with fun-LNil and q have q = (g LNil, g LNil) by simp
      then show ?thesis by (cases g LNil) simp-all
    next
      case (LCons x l')
      with ‹?fun-LCons x l'› q LCons show ?thesis by blast
    qed
  qed
qed
```

16.5 Derived operations – both on the set and abstract type

16.5.1 *Lconst*

```
definition Lconst M ≡ lfp (λN. CONS M N)

lemma Lconst-fun-mono: mono (CONS M)
  by (simp add: monoI CONS-mono)

lemma Lconst: Lconst M = CONS M (Lconst M)
  by (rule Lconst-def [THEN def-lfp-unfold]) (rule Lconst-fun-mono)

lemma Lconst-type:
  assumes M ∈ A
  shows Lconst M ∈ LList A
proof −
  have Lconst M ∈ {Lconst (id M)} by simp
  then show ?thesis
  proof coinduct
    case (LList N)
    then have N = Lconst M by simp
    also have ... = CONS M (Lconst M) by (rule Lconst)
    finally have ?CONS using ‹M ∈ A› by simp
    then show ?case ..
  qed
qed
```

lemma *Lconst-eq-LList-corec*: *Lconst M* = *LList-corec M* (λ*x*. *Some* (*x*, *x*))
apply (*rule equals-LList-corec*)
apply *simp*
apply (*rule Lconst*)
done

lemma *gfp-Lconst-eq-LList-corec*:
gfp (λ*N*. *CONS M N*) = *LList-corec M* (λ*x*. *Some*(*x*, *x*))
apply (*rule equals-LList-corec*)
apply *simp*
apply (*rule Lconst-fun-mono* [*THEN gfp-unfold*])
done

16.5.2 *Lmap* and *lmap*

definition
Lmap f M = *LList-corec M* (*List-case None* (λ*x M'*. *Some* (*f x*, *M'*)))
definition
lmap f l = *llist-corec l*
(λ*z*.
case z of LNil ⇒ *None*
| *LCons y z* ⇒ *Some* (*f y*, *z*))

lemma *Lmap-NIL* [*simp*]: *Lmap f NIL* = *NIL*
and *Lmap-CONS* [*simp*]: *Lmap f* (*CONS M N*) = *CONS* (*f M*) (*Lmap f N*)
by (*simp-all add*: *Lmap-def LList-corec*)

lemma *Lmap-type*:
assumes *M*: *M* ∈ *LList A*
and *f*: ⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ *B*
shows *Lmap f M* ∈ *LList B*
proof −
from *M* **have** *Lmap f M* ∈ {*Lmap f N* | *N*. *N* ∈ *LList A*} **by** *blast*
then show *?thesis*
proof *coinduct*
case (*LList L*)
then obtain *N* **where** *L*: *L* = *Lmap f N* **and** *N*: *N* ∈ *LList A* **by** *blast*
from *N* **show** *?case*
proof *cases*
case *NIL*
with *L* **have** *?NIL* **by** *simp*
then show *?thesis* **..**
next
case (*CONS K a*)
with *f L* **have** *?CONS* **by** *auto*
then show *?thesis* **..**
qed
qed

qed

lemma *Lmap-compose*:
assumes *M*: $M \in LList\ A$
shows $Lmap\ (f\ o\ g)\ M = Lmap\ f\ (Lmap\ g\ M)$ (**is** *?lhs M* = *?rhs M*)
proof −
have (*?lhs M*, *?rhs M*) ∈ {(*?lhs N*, *?rhs N*) | *N*. $N \in LList\ A$}
using *M* **by** *blast*
then show *?thesis*
proof (*coinduct taking*: *range* (λ*N*. *N*) *rule*: *LList-equalityI*)
case (*EqLList L M*)
then obtain *N* **where** *LM*: *L* = *?lhs N M* = *?rhs N* **and** *N*: $N \in LList\ A$
by *blast*
from *N* **show** *?case*
proof *cases*
case *NIL*
with *LM* **have** *?EqNIL* **by** *simp*
then show *?thesis* **..**
next
case *CONS*
with *LM* **have** *?EqCONS* **by** *auto*
then show *?thesis* **..**
qed
qed
qed

lemma *Lmap-ident*:
assumes *M*: $M \in LList\ A$
shows $Lmap\ (\lambda x.\ x)\ M = M$ (**is** *?lmap M* = -)
proof −
have (*?lmap M*, *M*) ∈ {(*?lmap N*, *N*) | *N*. $N \in LList\ A$} **using** *M* **by** *blast*
then show *?thesis*
proof (*coinduct taking*: *range* (λ*N*. *N*) *rule*: *LList-equalityI*)
case (*EqLList L M*)
then obtain *N* **where** *LM*: *L* = *?lmap N M* = *N* **and** *N*: $N \in LList\ A$ **by**
blast
from *N* **show** *?case*
proof *cases*
case *NIL*
with *LM* **have** *?EqNIL* **by** *simp*
then show *?thesis* **..**
next
case *CONS*
with *LM* **have** *?EqCONS* **by** *auto*
then show *?thesis* **..**
qed
qed
qed

lemma *lmap-LNil* [*simp*]: *lmap f LNil = LNil*
and *lmap-LCons* [*simp*]: *lmap f (LCons M N) = LCons (f M) (lmap f N)*
by (*simp-all add*: *lmap-def llist-corec*)

lemma *lmap-compose* [*simp*]: *lmap (f o g) l = lmap f (lmap g l)*
by (*coinduct l rule*: *llist-fun-equalityI*) *auto*

lemma *lmap-ident* [*simp*]: *lmap (λx. x) l = l*
by (*coinduct l rule*: *llist-fun-equalityI*) *auto*

16.5.3 *Lappend*

definition
Lappend M N = LList-corec (M, N)
(split (List-case
(List-case None (λN1 N2. Some (N1, (NIL, N2))))
(λM1 M2 N. Some (M1, (M2, N)))))
definition
lappend l n = llist-corec (l, n)
(split (llist-case
(llist-case None (λn1 n2. Some (n1, (LNil, n2))))
(λl1 l2 n. Some (l1, (l2, n)))))

lemma *Lappend-NIL-NIL* [*simp*]:
Lappend NIL NIL = NIL
and *Lappend-NIL-CONS* [*simp*]:
Lappend NIL (CONS N N′) = CONS N (Lappend NIL N′)
and *Lappend-CONS* [*simp*]:
Lappend (CONS M M′) N = CONS M (Lappend M′ N)
by (*simp-all add*: *Lappend-def LList-corec*)

lemma *Lappend-NIL* [*simp*]: *M ∈ LList A ⟹ Lappend NIL M = M*
by (*erule LList-fun-equalityI*) *auto*

lemma *Lappend-NIL2*: *M ∈ LList A ⟹ Lappend M NIL = M*
by (*erule LList-fun-equalityI*) *auto*

lemma *Lappend-type*:
assumes *M*: *M ∈ LList A* **and** *N*: *N ∈ LList A*
shows *Lappend M N ∈ LList A*
proof −
have *Lappend M N ∈ {Lappend u v | u v. u ∈ LList A ∧ v ∈ LList A}*
using *M N* **by** *blast*
then show *?thesis*
proof *coinduct*
case (*LList L*)
then obtain *M N* **where** *L*: *L = Lappend M N*
and *M*: *M ∈ LList A* **and** *N*: *N ∈ LList A*
by *blast*

```
    from M show ?case
    proof cases
      case NIL
      from N show ?thesis
      proof cases
        case NIL
        with L and ‹M = NIL› have ?NIL by simp
        then show ?thesis ..
      next
        case CONS
        with L and ‹M = NIL› have ?CONS by simp
        then show ?thesis ..
      qed
    next
      case CONS
      with L N have ?CONS by auto
      then show ?thesis ..
    qed
  qed
qed

lemma lappend-LNil-LNil [simp]: lappend LNil LNil = LNil
  and lappend-LNil-LCons [simp]: lappend LNil (LCons l l') = LCons l (lappend
LNil l')
  and lappend-LCons [simp]: lappend (LCons l l') m = LCons l (lappend l' m)
  by (simp-all add: lappend-def llist-corec)

lemma lappend-LNil1 [simp]: lappend LNil l = l
  by (coinduct l rule: llist-fun-equalityI) auto

lemma lappend-LNil2 [simp]: lappend l LNil = l
  by (coinduct l rule: llist-fun-equalityI) auto

lemma lappend-assoc: lappend (lappend l1 l2) l3 = lappend l1 (lappend l2 l3)
  by (coinduct l1 rule: llist-fun-equalityI) auto

lemma lmap-lappend-distrib: lmap f (lappend l n) = lappend (lmap f l) (lmap f n)
  by (coinduct l rule: llist-fun-equalityI) auto
```

16.6 iterates

llist-fun-equalityI cannot be used here!

```
definition
  iterates :: ('a ⇒ 'a) ⇒ 'a ⇒ 'a llist where
  iterates f a = llist-corec a (λx. Some (x, f x))

lemma iterates: iterates f x = LCons x (iterates f (f x))
  apply (unfold iterates-def)
  apply (subst llist-corec)
```

apply *simp*
done

lemma *lmap-iterates*: *lmap f* (*iterates f x*) = *iterates f* (*f x*)
proof −
have (*lmap f* (*iterates f x*), *iterates f* (*f x*)) ∈
{(*lmap f* (*iterates f u*), *iterates f* (*f u*)) | *u*. *True*} **by** *blast*
then show *?thesis*
proof (*coinduct rule*: *llist-equalityI*)
case (*Eqllist q*)
then obtain *x* **where** *q*: *q* = (*lmap f* (*iterates f x*), *iterates f* (*f x*))
by *blast*
also have *iterates f* (*f x*) = *LCons* (*f x*) (*iterates f* (*f* (*f x*)))
by (*subst iterates*) *rule*
also have *iterates f x* = *LCons x* (*iterates f* (*f x*))
by (*subst iterates*) *rule*
finally have *?EqLCons* **by** *auto*
then show *?case* **..**
qed
qed

lemma *iterates-lmap*: *iterates f x* = *LCons x* (*lmap f* (*iterates f x*))
by (*subst lmap-iterates*) (*rule iterates*)

16.7 A rather complex proof about iterates – cf. Andy Pitts

lemma *funpow-lmap*:
fixes *f* :: *'a* ⇒ *'a*
shows (*lmap f* ^ *n*) (*LCons b l*) = *LCons* ((*f* ^ *n*) *b*) ((*lmap f* ^ *n*) *l*)
by (*induct n*) *simp-all*

lemma *iterates-equality*:
assumes *h*: ⋀*x*. *h x* = *LCons x* (*lmap f* (*h x*))
shows *h* = *iterates f*
proof
fix *x*
have (*h x*, *iterates f x*) ∈
{((*lmap f* ^ *n*) (*h u*), (*lmap f* ^ *n*) (*iterates f u*)) | *u n*. *True*}
proof −
have (*h x*, *iterates f x*) = ((*lmap f* ^ *0*) (*h x*), (*lmap f* ^ *0*) (*iterates f x*))
by *simp*
then show *?thesis* **by** *blast*
qed
then show *h x* = *iterates f x*
proof (*coinduct rule*: *llist-equalityI*)
case (*Eqllist q*)
then obtain *u n* **where** *q* = ((*lmap f* ^ *n*) (*h u*), (*lmap f* ^ *n*) (*iterates f u*))
(**is** - = (*?q1*, *?q2*))

```
    by auto
  also have ?q1 = LCons ((f ^ n) u) ((lmap f ^ Suc n) (h u))
  proof −
    have ?q1 = (lmap f ^ n) (LCons u (lmap f (h u)))
      by (subst h) rule
    also have ... = LCons ((f ^ n) u) ((lmap f ^ n) (lmap f (h u)))
      by (rule funpow-lmap)
    also have (lmap f ^ n) (lmap f (h u)) = (lmap f ^ Suc n) (h u)
      by (simp add: funpow-swap1)
    finally show ?thesis .
  qed
  also have ?q2 = LCons ((f ^ n) u) ((lmap f ^ Suc n) (iterates f u))
  proof −
    have ?q2 = (lmap f ^ n) (LCons u (iterates f (f u)))
      by (subst iterates) rule
    also have ... = LCons ((f ^ n) u) ((lmap f ^ n) (iterates f (f u)))
      by (rule funpow-lmap)
    also have (lmap f ^ n) (iterates f (f u)) = (lmap f ^ Suc n) (iterates f u)
      by (simp add: lmap-iterates funpow-swap1)
    finally show ?thesis .
  qed
  finally have ?EqLCons by (auto simp del: funpow.simps)
  then show ?case ..
 qed
qed

lemma lappend-iterates: lappend (iterates f x) l = iterates f x
proof −
 have (lappend (iterates f x) l, iterates f x) ∈
   {(lappend (iterates f u) l, iterates f u) | u. True} by blast
 then show ?thesis
 proof (coinduct rule: llist-equalityI)
   case (Eqllist q)
   then obtain x where q = (lappend (iterates f x) l, iterates f x) by blast
   also have iterates f x = LCons x (iterates f (f x)) by (rule iterates)
   finally have ?EqLCons by auto
   then show ?case ..
 qed
qed

end
```

17 Parity: Even and Odd for int and nat

```
theory Parity
imports ATP-Linkup
begin
```

class *even-odd* = *type* +
  fixes *even* :: *'a* ⇒ *bool*

abbreviation
  odd :: *'a*::*even-odd* ⇒ *bool* **where**
  odd x ≡ ¬ *even x*

instantiation *nat* **and** *int* :: *even-odd*
begin

definition
  even-def [*presburger*]: *even x* ⟷ (*x*::*int*) *mod 2* = *0*

definition
  even-nat-def [*presburger*]: *even x* ⟷ *even* (*int x*)

instance ..

end

17.1 Even and odd are mutually exclusive

lemma *int-pos-lt-two-imp-zero-or-one*:
  0 <= *x* ==> (*x*::*int*) < *2* ==> *x* = *0* | *x* = *1*
  by *presburger*

lemma *neq-one-mod-two* [*simp*, *presburger*]:
  ((*x*::*int*) *mod 2* ~= *0*) = (*x mod 2* = *1*) **by** *presburger*

17.2 Behavior under integer arithmetic operations

lemma *even-times-anything*: *even* (*x*::*int*) ==> *even* (*x* * *y*)
  by (*simp add*: *even-def zmod-zmult1-eq'*)

lemma *anything-times-even*: *even* (*y*::*int*) ==> *even* (*x* * *y*)
  by (*simp add*: *even-def zmod-zmult1-eq*)

lemma *odd-times-odd*: *odd* (*x*::*int*) ==> *odd y* ==> *odd* (*x* * *y*)
  by (*simp add*: *even-def zmod-zmult1-eq*)

lemma *even-product*[*presburger*]: *even*((*x*::*int*) * *y*) = (*even x* | *even y*)
  apply (*auto simp add*: *even-times-anything anything-times-even*)
  apply (*rule ccontr*)
  apply (*auto simp add*: *odd-times-odd*)
  done

lemma *even-plus-even*: *even* (*x*::*int*) ==> *even y* ==> *even* (*x* + *y*)
  by *presburger*

lemma *even-plus-odd*: *even* (*x*::*int*) ==> *odd y* ==> *odd* (*x* + *y*)

by *presburger*

lemma *odd-plus-even*: *odd* (*x*::*int*) ==> *even y* ==> *odd* (*x* + *y*)
by *presburger*

lemma *odd-plus-odd*: *odd* (*x*::*int*) ==> *odd y* ==> *even* (*x* + *y*) **by** *presburger*

lemma *even-sum*[*presburger*]: *even* ((*x*::*int*) + *y*) = ((*even x* & *even y*) | (*odd x* & *odd y*))
by *presburger*

lemma *even-neg*[*presburger*]: *even* (−(*x*::*int*)) = *even x* **by** *presburger*

lemma *even-difference*:
even ((*x*::*int*) − *y*) = ((*even x* & *even y*) | (*odd x* & *odd y*)) **by** *presburger*

lemma *even-pow-gt-zero*:
even (*x*::*int*) ==> *0* < *n* ==> *even* (*x* ^ *n*)
by (*induct n*) (*auto simp add*: *even-product*)

lemma *odd-pow-iff*[*presburger*]: *odd* ((*x*::*int*) ^ *n*) ⟷ (*n* = *0* ∨ *odd x*)
apply (*induct n*, *simp-all*)
apply *presburger*
apply (*case-tac n*, *auto*)
apply (*simp-all add*: *even-product*)
done

lemma *odd-pow*: *odd x* ==> *odd*((*x*::*int*) ^ *n*) **by** (*simp add*: *odd-pow-iff*)

lemma *even-power*[*presburger*]: *even* ((*x*::*int*) ^ *n*) = (*even x* & *0* < *n*)
apply (*auto simp add*: *even-pow-gt-zero*)
apply (*erule contrapos-pp*, *erule odd-pow*)
apply (*erule contrapos-pp*, *simp add*: *even-def*)
done

lemma *even-zero*[*presburger*]: *even* (*0*::*int*) **by** *presburger*

lemma *odd-one*[*presburger*]: *odd* (*1*::*int*) **by** *presburger*

lemmas *even-odd-simps* [*simp*] = *even-def*[*of number-of v*,*standard*] *even-zero*
odd-one even-product even-sum even-neg even-difference even-power

17.3 Equivalent definitions

lemma *two-times-even-div-two*: *even* (*x*::*int*) ==> *2* * (*x div 2*) = *x*
by *presburger*

lemma *two-times-odd-div-two-plus-one*: *odd* (*x*::*int*) ==>
2 * (*x div 2*) + *1* = *x* **by** *presburger*

lemma *even-equiv-def*: *even* (*x*::*int*) = (*EX y*. *x* = *2* ∗ *y*) **by** *presburger*

lemma *odd-equiv-def*: *odd* (*x*::*int*) = (*EX y*. *x* = *2* ∗ *y* + *1*) **by** *presburger*

17.4 even and odd for nats

lemma *pos-int-even-equiv-nat-even*: *0* ≤ *x* ==> *even x* = *even* (*nat x*)
by (*simp add*: *even-nat-def*)

lemma *even-nat-product*[*presburger*]: *even*((*x*::*nat*) ∗ *y*) = (*even x* | *even y*)
by (*simp add*: *even-nat-def int-mult*)

lemma *even-nat-sum*[*presburger*]: *even* ((*x*::*nat*) + *y*) =
((*even x* & *even y*) | (*odd x* & *odd y*)) **by** *presburger*

lemma *even-nat-difference*[*presburger*]:
even ((*x*::*nat*) − *y*) = (*x* < *y* | (*even x* & *even y*) | (*odd x* & *odd y*))
by *presburger*

lemma *even-nat-Suc*[*presburger*]: *even* (*Suc x*) = *odd x* **by** *presburger*

lemma *even-nat-power*[*presburger*]: *even* ((*x*::*nat*) ^ *y*) = (*even x* & *0* < *y*)
by (*simp add*: *even-nat-def int-power*)

lemma *even-nat-zero*[*presburger*]: *even* (*0*::*nat*) **by** *presburger*

lemmas *even-odd-nat-simps* [*simp*] = *even-nat-def*[*of number-of v*,*standard*]
even-nat-zero even-nat-Suc even-nat-product even-nat-sum even-nat-power

17.5 Equivalent definitions

lemma *nat-lt-two-imp-zero-or-one*: (*x*::*nat*) < *Suc* (*Suc 0*) ==>
x = *0* | *x* = *Suc 0* **by** *presburger*

lemma *even-nat-mod-two-eq-zero*: *even* (*x*::*nat*) ==> *x mod* (*Suc* (*Suc 0*)) = *0*
by *presburger*

lemma *odd-nat-mod-two-eq-one*: *odd* (*x*::*nat*) ==> *x mod* (*Suc* (*Suc 0*)) = *Suc 0*
by *presburger*

lemma *even-nat-equiv-def*: *even* (*x*::*nat*) = (*x mod Suc* (*Suc 0*) = *0*)
by *presburger*

lemma *odd-nat-equiv-def*: *odd* (*x*::*nat*) = (*x mod Suc* (*Suc 0*) = *Suc 0*)
by *presburger*

lemma *even-nat-div-two-times-two*: *even* (*x*::*nat*) ==>
Suc (*Suc 0*) ∗ (*x div Suc* (*Suc 0*)) = *x* **by** *presburger*

lemma *odd-nat-div-two-times-two-plus-one*: *odd* (*x*::*nat*) ==>
Suc(*Suc* (*Suc 0*) * (*x div Suc* (*Suc 0*))) = *x* **by** *presburger*

lemma *even-nat-equiv-def2*: *even* (*x*::*nat*) = (*EX y*. *x* = *Suc* (*Suc 0*) * *y*)
by *presburger*

lemma *odd-nat-equiv-def2*: *odd* (*x*::*nat*) = (*EX y*. *x* = *Suc*(*Suc* (*Suc 0*) * *y*))
by *presburger*

17.6 Parity and powers

lemma *minus-one-even-odd-power*:
(*even x* --> (− *1*::'*a*::{*comm-ring-1*,*recpower*}) ^*x* = *1*) &
(*odd x* --> (− *1*::'*a*) ^*x* = − *1*)
apply (*induct x*)
apply (*rule conjI*)
apply *simp*
apply (*insert even-nat-zero*, *blast*)
apply (*simp add*: *power-Suc*)
done

lemma *minus-one-even-power* [*simp*]:
even x ==> (− *1*::'*a*::{*comm-ring-1*,*recpower*}) ^*x* = *1*
using *minus-one-even-odd-power* **by** *blast*

lemma *minus-one-odd-power* [*simp*]:
odd x ==> (− *1*::'*a*::{*comm-ring-1*,*recpower*}) ^*x* = − *1*
using *minus-one-even-odd-power* **by** *blast*

lemma *neg-one-even-odd-power*:
(*even x* --> (−*1*::'*a*::{*number-ring*,*recpower*}) ^*x* = *1*) &
(*odd x* --> (−*1*::'*a*) ^*x* = −*1*)
apply (*induct x*)
apply (*simp*, *simp add*: *power-Suc*)
done

lemma *neg-one-even-power* [*simp*]:
even x ==> (−*1*::'*a*::{*number-ring*,*recpower*}) ^*x* = *1*
using *neg-one-even-odd-power* **by** *blast*

lemma *neg-one-odd-power* [*simp*]:
odd x ==> (−*1*::'*a*::{*number-ring*,*recpower*}) ^*x* = −*1*
using *neg-one-even-odd-power* **by** *blast*

lemma *neg-power-if*:
(−*x*::'*a*::{*comm-ring-1*,*recpower*}) ^ *n* =
(*if even n then* (*x* ^ *n*) *else* −(*x* ^ *n*))
apply (*induct n*)
apply (*simp-all split*: *split-if-asm add*: *power-Suc*)

done

lemma *zero-le-even-power*: *even n ==>*
0 <= (x::'a::{recpower,ordered-ring-strict}) ^ n
apply (*simp add*: *even-nat-equiv-def2*)
apply (*erule exE*)
apply (*erule ssubst*)
apply (*subst power-add*)
apply (*rule zero-le-square*)
done

lemma *zero-le-odd-power*: *odd n ==>*
(0 <= (x::'a::{recpower,ordered-idom}) ^ n) = (0 <= x)
apply (*simp add*: *odd-nat-equiv-def2*)
apply (*erule exE*)
apply (*erule ssubst*)
apply (*subst power-Suc*)
apply (*subst power-add*)
apply (*subst zero-le-mult-iff*)
apply *auto*
apply (*subgoal-tac x = 0 & y > 0*)
apply (*erule conjE*, *assumption*)
apply (*subst power-eq-0-iff* [*symmetric*])
apply (*subgoal-tac 0 <= x^y * x^y*)
apply *simp*
apply (*rule zero-le-square*)+
done

lemma *zero-le-power-eq*[*presburger*]: *(0 <= (x::'a::{recpower,ordered-idom}) ^ n)*
=
(even n | (odd n & 0 <= x))
apply *auto*
apply (*subst zero-le-odd-power* [*symmetric*])
apply *assumption*+
apply (*erule zero-le-even-power*)
done

lemma *zero-less-power-eq*[*presburger*]: *(0 < (x::'a::{recpower,ordered-idom}) ^ n)*
=
(n = 0 | (even n & x ~= 0) | (odd n & 0 < x))
apply (*rule iffI*)
apply *clarsimp*
apply (*rule conjI*)
apply *clarsimp*
apply (*rule ccontr*)
apply (*subgoal-tac ~ (0 <= x^n)*)
apply *simp*
apply (*subst zero-le-odd-power*)
apply *assumption*

apply *simp*
apply (*rule notI*)
apply (*simp add*: *power-0-left*)
apply (*rule notI*)
apply (*simp add*: *power-0-left*)
apply *auto*
apply (*subgoal-tac 0 <= x^n*)
apply (*frule order-le-imp-less-or-eq*)
apply *simp*
apply (*erule zero-le-even-power*)
done

lemma *power-less-zero-eq*[*presburger*]: ((*x*::'*a*::{*recpower*,*ordered-idom*}) ^ *n* < *0*)
=
(*odd n* & *x* < *0*)
apply (*subst linorder-not-le* [*symmetric*])+
apply (*subst zero-le-power-eq*)
apply *auto*
done

lemma *power-le-zero-eq*[*presburger*]: ((*x*::'*a*::{*recpower*,*ordered-idom*}) ^ *n* <= *0*)
=
(*n* ~= *0* & ((*odd n* & *x* <= *0*) | (*even n* & *x* = *0*)))
apply (*subst linorder-not-less* [*symmetric*])+
apply (*subst zero-less-power-eq*)
apply *auto*
done

lemma *power-even-abs*: *even n* ==>
(*abs* (*x*::'*a*::{*recpower*,*ordered-idom*})) ^*n* = *x*^*n*
apply (*subst power-abs* [*symmetric*])
apply (*simp add*: *zero-le-even-power*)
done

lemma *zero-less-power-nat-eq*[*presburger*]: (*0* < (*x*::*nat*) ^ *n*) = (*n* = *0* | *0* < *x*)
by (*induct n*) *auto*

lemma *power-minus-even* [*simp*]: *even n* ==>
(− *x*) ^*n* = (*x*^*n*::'*a*::{*recpower*,*comm-ring-1*})
apply (*subst power-minus*)
apply *simp*
done

lemma *power-minus-odd* [*simp*]: *odd n* ==>
(− *x*) ^*n* = − (*x*^*n*::'*a*::{*recpower*,*comm-ring-1*})
apply (*subst power-minus*)
apply *simp*
done

17.7 General Lemmas About Division

lemma *Suc-times-mod-eq*: *1<k ==> Suc (k * m) mod k = 1*
apply (*induct m*)
apply (*simp-all add*: *mod-Suc*)
done

declare *Suc-times-mod-eq* [*of number-of w*, *standard*, *simp*]

lemma [*simp*]: *n div k ≤ (Suc n) div k*
by (*simp add*: *div-le-mono*)

lemma *Suc-n-div-2-gt-zero* [*simp*]: *(0::nat) < n ==> 0 < (n + 1) div 2*
by *arith*

lemma *div-2-gt-zero* [*simp*]: *(1::nat) < n ==> 0 < n div 2*
by *arith*

lemma *mod-mult-self3* [*simp*]: *(k*n + m) mod n = m mod (n::nat)*
by (*simp add*: *mult-ac add-ac*)

lemma *mod-mult-self4* [*simp*]: *Suc (k*n + m) mod n = Suc m mod n*
proof −
  have *Suc (k * n + m) mod n = (k * n + Suc m) mod n* **by** *simp*
  also have *... = Suc m mod n* **by** (*rule mod-mult-self3*)
  finally show *?thesis* **.**
qed

lemma *mod-Suc-eq-Suc-mod*: *Suc m mod n = Suc (m mod n) mod n*
apply (*subst mod-Suc* [*of m*])
apply (*subst mod-Suc* [*of m mod n*], *simp*)
done

17.8 More Even/Odd Results

lemma *even-mult-two-ex*: *even(n) = (∃m::nat. n = 2*m)*
by (*simp add*: *even-nat-equiv-def2 numeral-2-eq-2*)

lemma *odd-Suc-mult-two-ex*: *odd(n) = (∃m. n = Suc (2*m))*
by (*simp add*: *odd-nat-equiv-def2 numeral-2-eq-2*)

lemma *even-add* [*simp*]: *even(m + n::nat) = (even m = even n)*
by *auto*

lemma *odd-add* [*simp*]: *odd(m + n::nat) = (odd m ≠ odd n)*
by *auto*

lemma *div-Suc*: *Suc a div c = a div c + Suc 0 div c +*
  (a mod c + Suc 0 mod c) div c
 apply (*subgoal-tac Suc a = a + Suc 0*)

apply (*erule ssubst*)
apply (*rule div-add1-eq*, *simp*)
done

lemma *lemma-even-div2* [*simp*]: *even* (*n*::*nat*) ==> (*n* + *1*) *div* *2* = *n div* *2*
apply (*simp add*: *numeral-2-eq-2*)
apply (*subst div-Suc*)
apply (*simp add*: *even-nat-mod-two-eq-zero*)
done

lemma *lemma-not-even-div2* [*simp*]: ~*even* *n* ==> (*n* + *1*) *div* *2* = *Suc* (*n div* *2*)
apply (*simp add*: *numeral-2-eq-2*)
apply (*subst div-Suc*)
apply (*simp add*: *odd-nat-mod-two-eq-one*)
done

lemma *even-num-iff*: *0* < *n* ==> *even* *n* = (~ *even*(*n* − *1* :: *nat*))
by (*case-tac n*, *auto*)

lemma *even-even-mod-4-iff*: *even* (*n*::*nat*) = *even* (*n mod* *4*)
apply (*induct n*, *simp*)
apply (*subst mod-Suc*, *simp*)
done

lemma *lemma-odd-mod-4-div-2*: *n mod* *4* = (*3*::*nat*) ==> *odd*((*n* − *1*) *div* *2*)
apply (*rule-tac t* = *n* **and** *n1* = *4* **in** *mod-div-equality* [*THEN subst*])
apply (*simp add*: *even-num-iff*)
done

lemma *lemma-even-mod-4-div-2*: *n mod* *4* = (*1*::*nat*) ==> *even* ((*n* − *1*) *div* *2*)
by (*rule-tac t* = *n* **and** *n1* = *4* **in** *mod-div-equality* [*THEN subst*], *simp*)

Simplify, when the exponent is a numeral

lemmas *power-0-left-number-of* = *power-0-left* [*of number-of w*, *standard*]
declare *power-0-left-number-of* [*simp*]

lemmas *zero-le-power-eq-number-of* [*simp*] =
zero-le-power-eq [*of* - *number-of w*, *standard*]

lemmas *zero-less-power-eq-number-of* [*simp*] =
zero-less-power-eq [*of* - *number-of w*, *standard*]

lemmas *power-le-zero-eq-number-of* [*simp*] =
power-le-zero-eq [*of* - *number-of w*, *standard*]

lemmas *power-less-zero-eq-number-of* [*simp*] =
power-less-zero-eq [*of* - *number-of w*, *standard*]

lemmas *zero-less-power-nat-eq-number-of* [*simp*] =
zero-less-power-nat-eq [*of - number-of w*, *standard*]

lemmas *power-eq-0-iff-number-of* [*simp*] = *power-eq-0-iff* [*of - number-of w*, *standard*]

lemmas *power-even-abs-number-of* [*simp*] = *power-even-abs* [*of number-of w -*, *standard*]

17.9 An Equivalence for $0 \leq a\hat{}n$

lemma *even-power-le-0-imp-0*:
$a \hat{} (2*k) \leq (0::'a::\{ordered\text{-}idom,recpower\}) ==> a=0$
by (*induct k*) (*auto simp add*: *zero-le-mult-iff mult-le-0-iff power-Suc*)

lemma *zero-le-power-iff*[*presburger*]:
$(0 \leq a\hat{}n) = (0 \leq (a::'a::\{ordered\text{-}idom,recpower\}) \mid even\ n)$
proof *cases*
assume *even*: *even n*
then obtain *k* **where** $n = 2*k$
by (*auto simp add*: *even-nat-equiv-def2 numeral-2-eq-2*)
thus *?thesis* **by** (*simp add*: *zero-le-even-power even*)
next
assume *odd*: *odd n*
then obtain *k* **where** $n = Suc(2*k)$
by (*auto simp add*: *odd-nat-equiv-def2 numeral-2-eq-2*)
thus *?thesis*
by (*auto simp add*: *power-Suc zero-le-mult-iff zero-le-even-power*
dest!: *even-power-le-0-imp-0*)
qed

17.10 Miscellaneous

lemma *odd-pos*: *odd* $(n::nat) \Longrightarrow 0 < n$
by (*cases n*, *simp-all*)

lemma [*presburger*]:$(x + 1)\ div\ 2 = x\ div\ 2 \longleftrightarrow even\ (x::int)$ **by** *presburger*
lemma [*presburger*]: $(x + 1)\ div\ 2 = x\ div\ 2 + 1 \longleftrightarrow odd\ (x::int)$ **by** *presburger*
lemma *even-plus-one-div-two*: *even* $(x::int) ==> (x + 1)\ div\ 2 = x\ div\ 2$ **by** *presburger*
lemma *odd-plus-one-div-two*: *odd* $(x::int) ==> (x + 1)\ div\ 2 = x\ div\ 2 + 1$ **by** *presburger*

lemma [*presburger*]: $(Suc\ x)\ div\ Suc\ (Suc\ 0) = x\ div\ Suc\ (Suc\ 0) \longleftrightarrow even\ x$ **by** *presburger*
lemma [*presburger*]: $(Suc\ x)\ div\ Suc\ (Suc\ 0) = x\ div\ Suc\ (Suc\ 0) \longleftrightarrow even\ x$ **by** *presburger*
lemma *even-nat-plus-one-div-two*: *even* $(x::nat) ==>$
$(Suc\ x)\ div\ Suc\ (Suc\ 0) = x\ div\ Suc\ (Suc\ 0)$ **by** *presburger*

lemma *odd-nat-plus-one-div-two*: *odd* (*x*::*nat*) ==>
(*Suc x*) *div Suc* (*Suc 0*) = *Suc* (*x div Suc* (*Suc 0*)) **by** *presburger*

end

18 Commutative-Ring: Proving equalities in commutative rings

theory *Commutative-Ring*
imports *List Parity*
uses (*comm-ring.ML*)
begin

Syntax of multivariate polynomials (pol) and polynomial expressions.

datatype *'a pol* =
Pc 'a
| *Pinj nat 'a pol*
| *PX 'a pol nat 'a pol*

datatype *'a polex* =
Pol 'a pol
| *Add 'a polex 'a polex*
| *Sub 'a polex 'a polex*
| *Mul 'a polex 'a polex*
| *Pow 'a polex nat*
| *Neg 'a polex*

Interpretation functions for the shadow syntax.

fun
Ipol :: *'a*::{*comm-ring*,*recpower*} *list* ⇒ *'a pol* ⇒ *'a*
where
Ipol l (*Pc c*) = *c*
| *Ipol l* (*Pinj i P*) = *Ipol* (*drop i l*) *P*
| *Ipol l* (*PX P x Q*) = *Ipol l P* ∗ (*hd l*) ^*x* + *Ipol* (*drop 1 l*) *Q*

fun
Ipolex :: *'a*::{*comm-ring*,*recpower*} *list* ⇒ *'a polex* ⇒ *'a*
where
Ipolex l (*Pol P*) = *Ipol l P*
| *Ipolex l* (*Add P Q*) = *Ipolex l P* + *Ipolex l Q*
| *Ipolex l* (*Sub P Q*) = *Ipolex l P* − *Ipolex l Q*
| *Ipolex l* (*Mul P Q*) = *Ipolex l P* ∗ *Ipolex l Q*
| *Ipolex l* (*Pow p n*) = *Ipolex l p* ^ *n*
| *Ipolex l* (*Neg P*) = − *Ipolex l P*

Create polynomial normalized polynomials given normalized inputs.

definition

```
  mkPinj :: nat ⇒ 'a pol ⇒ 'a pol where
  mkPinj x P = (case P of
    Pc c ⇒ Pc c |
    Pinj y P ⇒ Pinj (x + y) P |
    PX p1 y p2 ⇒ Pinj x P)

definition
  mkPX :: 'a::{comm-ring,recpower} pol ⇒ nat ⇒ 'a pol ⇒ 'a pol where
  mkPX P i Q = (case P of
    Pc c ⇒ (if (c = 0) then (mkPinj 1 Q) else (PX P i Q)) |
    Pinj j R ⇒ PX P i Q |
    PX P2 i2 Q2 ⇒ (if (Q2 = (Pc 0)) then (PX P2 (i+i2) Q) else (PX P i Q))
)
```

Defining the basic ring operations on normalized polynomials

```
function
  add :: 'a::{comm-ring,recpower} pol ⇒ 'a pol ⇒ 'a pol (infixl ⊕ 65)
where
    Pc a ⊕ Pc b = Pc (a + b)
  | Pc c ⊕ Pinj i P = Pinj i (P ⊕ Pc c)
  | Pinj i P ⊕ Pc c = Pinj i (P ⊕ Pc c)
  | Pc c ⊕ PX P i Q = PX P i (Q ⊕ Pc c)
  | PX P i Q ⊕ Pc c = PX P i (Q ⊕ Pc c)
  | Pinj x P ⊕ Pinj y Q =
      (if x = y then mkPinj x (P ⊕ Q)
       else (if x > y then mkPinj y (Pinj (x − y) P ⊕ Q)
         else mkPinj x (Pinj (y − x) Q ⊕ P)))
  | Pinj x P ⊕ PX Q y R =
      (if x = 0 then P ⊕ PX Q y R
       else (if x = 1 then PX Q y (R ⊕ P)
         else PX Q y (R ⊕ Pinj (x − 1) P)))
  | PX P x R ⊕ Pinj y Q =
      (if y = 0 then PX P x R ⊕ Q
       else (if y = 1 then PX P x (R ⊕ Q)
         else PX P x (R ⊕ Pinj (y − 1) Q)))
  | PX P1 x P2 ⊕ PX Q1 y Q2 =
      (if x = y then mkPX (P1 ⊕ Q1) x (P2 ⊕ Q2)
       else (if x > y then mkPX (PX P1 (x − y) (Pc 0) ⊕ Q1) y (P2 ⊕ Q2)
         else mkPX (PX Q1 (y−x) (Pc 0) ⊕ P1) x (P2 ⊕ Q2)))
by pat-completeness auto
termination by (relation measure (λ(x, y). size x + size y)) auto

function
  mul :: 'a::{comm-ring,recpower} pol ⇒ 'a pol ⇒ 'a pol (infixl ⊗ 70)
where
    Pc a ⊗ Pc b = Pc (a * b)
  | Pc c ⊗ Pinj i P =
      (if c = 0 then Pc 0 else mkPinj i (P ⊗ Pc c))
  | Pinj i P ⊗ Pc c =
```

```
    (if c = 0 then Pc 0 else mkPinj i (P ⊗ Pc c))
| Pc c ⊗ PX P i Q =
    (if c = 0 then Pc 0 else mkPX (P ⊗ Pc c) i (Q ⊗ Pc c))
| PX P i Q ⊗ Pc c =
    (if c = 0 then Pc 0 else mkPX (P ⊗ Pc c) i (Q ⊗ Pc c))
| Pinj x P ⊗ Pinj y Q =
    (if x = y then mkPinj x (P ⊗ Q) else
      (if x > y then mkPinj y (Pinj (x−y) P ⊗ Q)
        else mkPinj x (Pinj (y − x) Q ⊗ P)))
| Pinj x P ⊗ PX Q y R =
    (if x = 0 then P ⊗ PX Q y R else
      (if x = 1 then mkPX (Pinj x P ⊗ Q) y (R ⊗ P)
        else mkPX (Pinj x P ⊗ Q) y (R ⊗ Pinj (x − 1) P)))
| PX P x R ⊗ Pinj y Q =
    (if y = 0 then PX P x R ⊗ Q else
      (if y = 1 then mkPX (Pinj y Q ⊗ P) x (R ⊗ Q)
        else mkPX (Pinj y Q ⊗ P) x (R ⊗ Pinj (y − 1) Q)))
| PX P1 x P2 ⊗ PX Q1 y Q2 =
    mkPX (P1 ⊗ Q1) (x + y) (P2 ⊗ Q2) ⊕
      (mkPX (P1 ⊗ mkPinj 1 Q2) x (Pc 0) ⊕
        (mkPX (Q1 ⊗ mkPinj 1 P2) y (Pc 0)))
by pat-completeness auto
termination by (relation measure (λ(x, y). size x + size y))
  (auto simp add: mkPinj-def split: pol.split)
```

Negation

```
fun
  neg :: 'a::{comm-ring,recpower} pol ⇒ 'a pol
where
    neg (Pc c) = Pc (−c)
  | neg (Pinj i P) = Pinj i (neg P)
  | neg (PX P x Q) = PX (neg P) x (neg Q)
```

Substraction

```
definition
  sub :: 'a::{comm-ring,recpower} pol ⇒ 'a pol ⇒ 'a pol (infixl ⊖ 65)
where
  sub P Q = P ⊕ neg Q
```

Square for Fast Exponentation

```
fun
  sqr :: 'a::{comm-ring,recpower} pol ⇒ 'a pol
where
    sqr (Pc c) = Pc (c ∗ c)
  | sqr (Pinj i P) = mkPinj i (sqr P)
  | sqr (PX A x B) = mkPX (sqr A) (x + x) (sqr B) ⊕
      mkPX (Pc (1 + 1) ⊗ A ⊗ mkPinj 1 B) x (Pc 0)
```

Fast Exponentation

```
fun
```

```
  pow :: nat ⇒ 'a::{comm-ring,recpower} pol ⇒ 'a pol
where
    pow 0 P = Pc 1
  | pow n P = (if even n then pow (n div 2) (sqr P)
       else P ⊗ pow (n div 2) (sqr P))

lemma pow-if:
  pow n P =
   (if n = 0 then Pc 1 else if even n then pow (n div 2) (sqr P)
    else P ⊗ pow (n div 2) (sqr P))
  by (cases n) simp-all
```

Normalization of polynomial expressions

```
fun
  norm :: 'a::{comm-ring,recpower} polex ⇒ 'a pol
where
    norm (Pol P) = P
  | norm (Add P Q) = norm P ⊕ norm Q
  | norm (Sub P Q) = norm P ⊖ norm Q
  | norm (Mul P Q) = norm P ⊗ norm Q
  | norm (Pow P n) = pow n (norm P)
  | norm (Neg P) = neg (norm P)
```

mkPinj preserve semantics

```
lemma mkPinj-ci: Ipol l (mkPinj a B) = Ipol l (Pinj a B)
  by (induct B) (auto simp add: mkPinj-def ring-simps)
```

mkPX preserves semantics

```
lemma mkPX-ci: Ipol l (mkPX A b C) = Ipol l (PX A b C)
  by (cases A) (auto simp add: mkPX-def mkPinj-ci power-add ring-simps)
```

Correctness theorems for the implemented operations

Negation

```
lemma neg-ci: Ipol l (neg P) = -(Ipol l P)
  by (induct P arbitrary: l) auto
```

Addition

```
lemma add-ci: Ipol l (P ⊕ Q) = Ipol l P + Ipol l Q
proof (induct P Q arbitrary: l rule: add.induct)
  case (6 x P y Q)
  show ?case
  proof (rule linorder-cases)
    assume x < y
    with 6 show ?case by (simp add: mkPinj-ci ring-simps)
  next
    assume x = y
    with 6 show ?case by (simp add: mkPinj-ci)
  next
```

```
    assume x > y
    with 6 show ?case by (simp add: mkPinj-ci ring-simps)
  qed
next
  case (7 x P Q y R)
  have x = 0 ∨ x = 1 ∨ x > 1 by arith
  moreover
  { assume x = 0 with 7 have ?case by simp }
  moreover
  { assume x = 1 with 7 have ?case by (simp add: ring-simps) }
  moreover
  { assume x > 1 from 7 have ?case by (cases x) simp-all }
  ultimately show ?case by blast
next
  case (8 P x R y Q)
  have y = 0 ∨ y = 1 ∨ y > 1 by arith
  moreover
  { assume y = 0 with 8 have ?case by simp }
  moreover
  { assume y = 1 with 8 have ?case by simp }
  moreover
  { assume y > 1 with 8 have ?case by simp }
  ultimately show ?case by blast
next
  case (9 P1 x P2 Q1 y Q2)
  show ?case
  proof (rule linorder-cases)
    assume a: x < y hence EX d. d + x = y by arith
    with 9 a show ?case by (auto simp add: mkPX-ci power-add ring-simps)
  next
    assume a: y < x hence EX d. d + y = x by arith
    with 9 a show ?case by (auto simp add: power-add mkPX-ci ring-simps)
  next
    assume x = y
    with 9 show ?case by (simp add: mkPX-ci ring-simps)
  qed
qed (auto simp add: ring-simps)
```

Multiplication

```
lemma mul-ci: Ipol l (P ⊗ Q) = Ipol l P * Ipol l Q
  by (induct P Q arbitrary: l rule: mul.induct)
    (simp-all add: mkPX-ci mkPinj-ci ring-simps add-ci power-add)
```

Substraction

```
lemma sub-ci: Ipol l (P ⊖ Q) = Ipol l P - Ipol l Q
  by (simp add: add-ci neg-ci sub-def)
```

Square

```
lemma sqr-ci: Ipol ls (sqr P) = Ipol ls P * Ipol ls P
```

```
  by (induct P arbitrary: ls)
    (simp-all add: add-ci mkPinj-ci mkPX-ci mul-ci ring-simps power-add)
```

Power

```
lemma even-pow:even n ⟹ pow n P = pow (n div 2) (sqr P)
  by (induct n) simp-all

lemma pow-ci: Ipol ls (pow n P) = Ipol ls P ^ n
proof (induct n arbitrary: P rule: nat-less-induct)
  case (1 k)
  show ?case
  proof (cases k)
    case 0
    then show ?thesis by simp
  next
    case (Suc l)
    show ?thesis
    proof cases
      assume even l
      then have Suc l div 2 = l div 2
        by (simp add: nat-number even-nat-plus-one-div-two)
      moreover
      from Suc have l < k by simp
      with 1 have ⋀P. Ipol ls (pow l P) = Ipol ls P ^ l by simp
      moreover
      note Suc ‹even l› even-nat-plus-one-div-two
      ultimately show ?thesis by (auto simp add: mul-ci power-Suc even-pow)
    next
      assume odd l
      {
        fix p
        have Ipol ls (sqr P) ^ (Suc l div 2) = Ipol ls P ^ Suc l
        proof (cases l)
          case 0
          with ‹odd l› show ?thesis by simp
        next
          case (Suc w)
          with ‹odd l› have even w by simp
          have two-times: 2 * (w div 2) = w
            by (simp only: numerals even-nat-div-two-times-two [OF ‹even w›])
          have Ipol ls P * Ipol ls P = Ipol ls P ^ Suc (Suc 0)
            by (simp add: power-Suc)
          then have Ipol ls P * Ipol ls P = Ipol ls P ^ 2
            by (simp add: numerals)
          with Suc show ?thesis
            by (auto simp add: power-mult [symmetric, of - 2 -] two-times mul-ci
sqr-ci)
        qed
      } with 1 Suc ‹odd l› show ?thesis by simp
```

qed
qed
qed

Normalization preserves semantics

lemma *norm-ci*: *Ipolex l Pe* = *Ipol l* (*norm Pe*)
by (*induct Pe*) (*simp-all add*: *add-ci sub-ci mul-ci neg-ci pow-ci*)

Reflection lemma: Key to the (incomplete) decision procedure

lemma *norm-eq*:
assumes *norm P1* = *norm P2*
shows *Ipolex l P1* = *Ipolex l P2*
proof −
from *prems* **have** *Ipol l* (*norm P1*) = *Ipol l* (*norm P2*) **by** *simp*
then show *?thesis* **by** (*simp only*: *norm-ci*)
qed

use *comm-ring.ML*
setup *CommRing.setup*

end

19 Continuity: Continuity and iterations (of set transformers)

theory *Continuity*
imports *ATP-Linkup*
begin

19.1 Continuity for complete lattices

definition
chain :: (*nat* ⇒ *'a*::*complete-lattice*) ⇒ *bool* **where**
chain M ⟷ (∀ *i*. *M i* ≤ *M* (*Suc i*))

definition
continuous :: (*'a*::*complete-lattice* ⇒ *'a*::*complete-lattice*) ⇒ *bool* **where**
continuous F ⟷ (∀ *M*. *chain M* ⟶ *F* (*SUP i*. *M i*) = (*SUP i*. *F* (*M i*)))

lemma *SUP-nat-conv*:
(*SUP n*. *M n*) = *sup* (*M 0*) (*SUP n*. *M*(*Suc n*))
apply(*rule order-antisym*)
apply(*rule SUP-leI*)
apply(*case-tac n*)
apply *simp*
apply (*fast intro*:*le-SUPI le-supI2*)

apply(*simp*)
apply (*blast intro*:*SUP-leI le-SUPI*)
done

lemma *continuous-mono*: **fixes** *F* :: *'a*::*complete-lattice* ⇒ *'a*::*complete-lattice*
assumes *continuous F* **shows** *mono F*
proof
fix *A B* :: *'a* **assume** *A* <= *B*
let *?C* = %*i*::*nat*. *if i=0 then A else B*
have *chain ?C* **using** ‹*A* <= *B*› **by**(*simp add*:*chain-def*)
have *F B* = *sup* (*F A*) (*F B*)
proof −
have *sup A B* = *B* **using** ‹*A* <= *B*› **by** (*simp add*:*sup-absorb2*)
hence *F B* = *F*(*SUP i*. *?C i*) **by** (*subst SUP-nat-conv*) *simp*
also have ... = (*SUP i*. *F*(*?C i*))
using ‹*chain ?C*› ‹*continuous F*› **by**(*simp add*:*continuous-def*)
also have ... = *sup* (*F A*) (*F B*) **by** (*subst SUP-nat-conv*) *simp*
finally show *?thesis* **.**
qed
thus *F A* ≤ *F B* **by**(*subst le-iff-sup*, *simp*)
qed

lemma *continuous-lfp*:
assumes *continuous F* **shows** *lfp F* = (*SUP i*. (*F*^*i*) *bot*)
proof −
note *mono* = *continuous-mono*[*OF* ‹*continuous F*›]
{ fix *i* **have** (*F*^*i*) *bot* ≤ *lfp F*
proof (*induct i*)
show (*F*^*0*) *bot* ≤ *lfp F* **by** *simp*
next
case (*Suc i*)
have (*F*^(*Suc i*)) *bot* = *F*((*F*^*i*) *bot*) **by** *simp*
also have ... ≤ *F*(*lfp F*) **by**(*rule monoD*[*OF mono Suc*])
also have ... = *lfp F* **by**(*simp add*:*lfp-unfold*[*OF mono*, *symmetric*])
finally show *?case* **.**
qed **}**
hence (*SUP i*. (*F*^*i*) *bot*) ≤ *lfp F* **by** (*blast intro*!:*SUP-leI*)
moreover have *lfp F* ≤ (*SUP i*. (*F*^*i*) *bot*) (**is** - ≤ *?U*)
proof (*rule lfp-lowerbound*)
have *chain*(%*i*. (*F*^*i*) *bot*)
proof −
{ fix *i* **have** (*F*^*i*) *bot* ≤ (*F*^(*Suc i*)) *bot*
proof (*induct i*)
case *0* **show** *?case* **by** *simp*
next
case *Suc* **thus** *?case* **using** *monoD*[*OF mono Suc*] **by** *auto*
qed **}**
thus *?thesis* **by**(*auto simp add*:*chain-def*)
qed

hence *F ?U = (SUP i. (F^(i+1)) bot)* **using** ‹*continuous F*› **by** (*simp add*:*continuous-def*)
also have ... ≤ *?U* **by**(*fast intro*:*SUP-leI le-SUPI*)
finally show *F ?U* ≤ *?U* **.**
qed
ultimately show *?thesis* **by** (*blast intro*:*order-antisym*)
qed

The following development is just for sets but presents an up and a down version of chains and continuity and covers *gfp*.

19.2 Chains

definition
up-chain :: (*nat* => *'a set*) => *bool* **where**
up-chain F = (∀ *i*. *F i* ⊆ *F* (*Suc i*))

lemma *up-chainI*: (!!*i*. *F i* ⊆ *F* (*Suc i*)) ==> *up-chain F*
by (*simp add*: *up-chain-def*)

lemma *up-chainD*: *up-chain F* ==> *F i* ⊆ *F* (*Suc i*)
by (*simp add*: *up-chain-def*)

lemma *up-chain-less-mono*:
up-chain F ==> *x* < *y* ==> *F x* ⊆ *F y*
apply (*induct y*)
apply (*blast dest*: *up-chainD elim*: *less-SucE*)+
done

lemma *up-chain-mono*: *up-chain F* ==> *x* ≤ *y* ==> *F x* ⊆ *F y*
apply (*drule le-imp-less-or-eq*)
apply (*blast dest*: *up-chain-less-mono*)
done

definition
down-chain :: (*nat* => *'a set*) => *bool* **where**
down-chain F = (∀ *i*. *F* (*Suc i*) ⊆ *F i*)

lemma *down-chainI*: (!!*i*. *F* (*Suc i*) ⊆ *F i*) ==> *down-chain F*
by (*simp add*: *down-chain-def*)

lemma *down-chainD*: *down-chain F* ==> *F* (*Suc i*) ⊆ *F i*
by (*simp add*: *down-chain-def*)

lemma *down-chain-less-mono*:
down-chain F ==> *x* < *y* ==> *F y* ⊆ *F x*
apply (*induct y*)
apply (*blast dest*: *down-chainD elim*: *less-SucE*)+

done

lemma *down-chain-mono*: *down-chain F ==> x ≤ y ==> F y ⊆ F x*
apply (*drule le-imp-less-or-eq*)
apply (*blast dest*: *down-chain-less-mono*)
done

19.3 Continuity

definition
up-cont :: (*'a set => 'a set*) => *bool* **where**
up-cont f = (∀ *F*. *up-chain F --> f* (⋃(*range F*)) = ⋃(*f ' range F*))

lemma *up-contI*:
(!!*F*. *up-chain F ==> f* (⋃(*range F*)) = ⋃(*f ' range F*)) ==> *up-cont f*
apply (*unfold up-cont-def*)
apply *blast*
done

lemma *up-contD*:
up-cont f ==> up-chain F ==> f (⋃(*range F*)) = ⋃(*f ' range F*)
apply (*unfold up-cont-def*)
apply *auto*
done

lemma *up-cont-mono*: *up-cont f ==> mono f*
apply (*rule monoI*)
apply (*drule-tac F* = λ*i*. *if i* = *0 then x else y* **in** *up-contD*)
apply (*rule up-chainI*)
apply *simp*
apply (*drule Un-absorb1*)
apply (*auto simp add*: *nat-not-singleton*)
done

definition
down-cont :: (*'a set => 'a set*) => *bool* **where**
down-cont f =
(∀ *F*. *down-chain F --> f* (*Inter* (*range F*)) = *Inter* (*f ' range F*))

lemma *down-contI*:
(!!*F*. *down-chain F ==> f* (*Inter* (*range F*)) = *Inter* (*f ' range F*)) ==>
down-cont f
apply (*unfold down-cont-def*)
apply *blast*
done

lemma *down-contD*: *down-cont f ==> down-chain F ==>*

```
    f (Inter (range F)) = Inter (f ' range F)
  apply (unfold down-cont-def)
  apply auto
  done

lemma down-cont-mono: down-cont f ==> mono f
apply (rule monoI)
apply (drule-tac F = λi. if i = 0 then y else x in down-contD)
 apply (rule down-chainI)
 apply simp
apply (drule Int-absorb1)
apply auto
apply (auto simp add: nat-not-singleton)
done
```

19.4 Iteration

```
definition
  up-iterate :: ('a set => 'a set) => nat => 'a set where
  up-iterate f n = (f^n) {}

lemma up-iterate-0 [simp]: up-iterate f 0 = {}
  by (simp add: up-iterate-def)

lemma up-iterate-Suc [simp]: up-iterate f (Suc i) = f (up-iterate f i)
  by (simp add: up-iterate-def)

lemma up-iterate-chain: mono F ==> up-chain (up-iterate F)
  apply (rule up-chainI)
  apply (induct-tac i)
   apply simp+
  apply (erule (1) monoD)
  done

lemma UNION-up-iterate-is-fp:
  up-cont F ==>
    F (UNION UNIV (up-iterate F)) = UNION UNIV (up-iterate F)
  apply (frule up-cont-mono [THEN up-iterate-chain])
  apply (drule (1) up-contD)
  apply simp
  apply (auto simp del: up-iterate-Suc simp add: up-iterate-Suc [symmetric])
  apply (case-tac xa)
   apply auto
  done

lemma UNION-up-iterate-lowerbound:
    mono F ==> F P = P ==> UNION UNIV (up-iterate F) ⊆ P
  apply (subgoal-tac (!!i. up-iterate F i ⊆ P))
   apply fast
```

apply (*induct-tac i*)
prefer *2* **apply** (*drule* (*1*) *monoD*)
apply *auto*
done

lemma *UNION-up-iterate-is-lfp*:
up-cont F ==> lfp F = UNION UNIV (*up-iterate F*)
apply (*rule set-eq-subset* [*THEN iffD2*])
apply (*rule conjI*)
prefer *2*
apply (*drule up-cont-mono*)
apply (*rule UNION-up-iterate-lowerbound*)
apply *assumption*
apply (*erule lfp-unfold* [*symmetric*])
apply (*rule lfp-lowerbound*)
apply (*rule set-eq-subset* [*THEN iffD1*, *THEN conjunct2*])
apply (*erule UNION-up-iterate-is-fp* [*symmetric*])
done

definition
down-iterate :: (*'a set => 'a set*) *=> nat => 'a set* **where**
down-iterate f n = (*f^n*) *UNIV*

lemma *down-iterate-0* [*simp*]: *down-iterate f 0 = UNIV*
by (*simp add*: *down-iterate-def*)

lemma *down-iterate-Suc* [*simp*]:
down-iterate f (*Suc i*) = *f* (*down-iterate f i*)
by (*simp add*: *down-iterate-def*)

lemma *down-iterate-chain*: *mono F ==> down-chain* (*down-iterate F*)
apply (*rule down-chainI*)
apply (*induct-tac i*)
apply *simp+*
apply (*erule* (*1*) *monoD*)
done

lemma *INTER-down-iterate-is-fp*:
down-cont F ==>
F (*INTER UNIV* (*down-iterate F*)) = *INTER UNIV* (*down-iterate F*)
apply (*frule down-cont-mono* [*THEN down-iterate-chain*])
apply (*drule* (*1*) *down-contD*)
apply *simp*
apply (*auto simp del*: *down-iterate-Suc simp add*: *down-iterate-Suc* [*symmetric*])
apply (*case-tac xa*)
apply *auto*
done

lemma *INTER-down-iterate-upperbound*:
  mono F ==> F P = P ==> P ⊆ INTER UNIV (down-iterate F)
 apply (*subgoal-tac* (!!*i*. *P ⊆ down-iterate F i*))
  apply *fast*
 apply (*induct-tac i*)
 prefer *2* **apply** (*drule* (*1*) *monoD*)
  apply *auto*
 done

lemma *INTER-down-iterate-is-gfp*:
  down-cont F ==> gfp F = INTER UNIV (down-iterate F)
 apply (*rule set-eq-subset* [*THEN iffD2*])
 apply (*rule conjI*)
  apply (*drule down-cont-mono*)
  apply (*rule INTER-down-iterate-upperbound*)
   apply *assumption*
  apply (*erule gfp-unfold* [*symmetric*])
 apply (*rule gfp-upperbound*)
 apply (*rule set-eq-subset* [*THEN iffD1*, *THEN conjunct2*])
 apply (*erule INTER-down-iterate-is-fp*)
 done

end

20 Countable: Encoding (almost) everything into natural numbers

theory *Countable*
imports *Finite-Set List Hilbert-Choice*
begin

20.1 The class of countable types

class *countable* = *itself* +
 assumes *ex-inj*: ∃ *to-nat* :: *'a* ⇒ *nat*. *inj to-nat*

lemma *countable-classI*:
 fixes *f* :: *'a* ⇒ *nat*
 assumes ⋀*x y*. *f x* = *f y* ⟹ *x* = *y*
 shows *OFCLASS*(*'a*, *countable-class*)
proof (*intro-classes*, *rule exI*)
 show *inj f*
  by (*rule injI* [*OF assms*]) *assumption*
qed

20.2 Conversion functions

definition *to-nat* :: *'a*::*countable* ⇒ *nat* **where**

to-nat = (*SOME f*. *inj f*)

definition *from-nat* :: *nat* ⇒ *'a*::*countable* **where**
from-nat = *inv* (*to-nat* :: *'a* ⇒ *nat*)

lemma *inj-to-nat* [*simp*]: *inj to-nat*
by (*rule exE-some* [*OF ex-inj*]) (*simp add*: *to-nat-def*)

lemma *to-nat-split* [*simp*]: *to-nat x* = *to-nat y* ⟷ *x* = *y*
using *injD* [*OF inj-to-nat*] **by** *auto*

lemma *from-nat-to-nat* [*simp*]:
from-nat (*to-nat x*) = *x*
by (*simp add*: *from-nat-def*)

20.3 Countable types

instance *nat* :: *countable*
by (*rule countable-classI* [*of id*]) *simp*

subclass (**in** *finite*) *countable*
proof *unfold-locales*
have *finite* (*UNIV*::*'a set*) **by** (*rule finite-UNIV*)
with *finite-conv-nat-seg-image* [*of UNIV*]
obtain *n* **and** *f* :: *nat* ⇒ *'a*
where *UNIV* = *f* ' {*i*. *i* < *n*} **by** *auto*
then have *surj f* **unfolding** *surj-def* **by** *auto*
then have *inj* (*inv f*) **by** (*rule surj-imp-inj-inv*)
then show ∃ *to-nat* :: *'a* ⇒ *nat*. *inj to-nat* **by** (*rule exI*[*of inj*])
qed

Pairs

primrec *sum* :: *nat* ⇒ *nat*
where
sum 0 = 0
| *sum* (*Suc n*) = *Suc n* + *sum n*

lemma *sum-arith*: *sum n* = *n* ∗ *Suc n div* 2
by (*induct n*) *auto*

lemma *sum-mono*: *n* ≥ *m* ⟹ *sum n* ≥ *sum m*
by (*induct n m rule*: *diff-induct*) *auto*

definition
pair-encode = (λ(*m*, *n*). *sum* (*m* + *n*) + *m*)

lemma *inj-pair-cencode*: *inj pair-encode*
unfolding *pair-encode-def*
proof (*rule injI*, *simp only*: *split-paired-all split-conv*)
fix *a b c d*

```
  assume eq: sum (a + b) + a = sum (c + d) + c
  have a + b = c + d ∨ a + b ≥ Suc (c + d) ∨ c + d ≥ Suc (a + b) by arith
  then
  show (a, b) = (c, d)
  proof (elim disjE)
    assume sumeq: a + b = c + d
    then have a = c using eq by auto
    moreover from sumeq this have b = d by auto
    ultimately show ?thesis by simp
  next
    assume a + b ≥ Suc (c + d)
    from sum-mono[OF this] eq
    show ?thesis by auto
  next
    assume c + d ≥ Suc (a + b)
    from sum-mono[OF this] eq
    show ?thesis by auto
  qed
qed

instance ∗ :: (countable, countable) countable
by (rule countable-classI [of λ(x, y). pair-encode (to-nat x, to-nat y)])
  (auto dest: injD [OF inj-pair-cencode] injD [OF inj-to-nat])
```

Sums

```
instance +:: (countable, countable) countable
  by (rule countable-classI [of (λx. case x of Inl a ⇒ to-nat (False, to-nat a)
                                  | Inr b ⇒ to-nat (True, to-nat b))])
    (auto split:sum.splits)
```

Integers

```
lemma int-cases: (i::int) = 0 ∨ i < 0 ∨ i > 0
by presburger

lemma int-pos-neg-zero:
  obtains (zero) (z::int) = 0 sgn z = 0 abs z = 0
  | (pos) n where z = of-nat n sgn z = 1 abs z = of-nat n
  | (neg) n where z = − (of-nat n) sgn z = −1 abs z = of-nat n
apply atomize-elim
apply (insert int-cases[of z])
apply (auto simp:zsgn-def)
apply (rule-tac x=nat (−z) in exI, simp)
apply (rule-tac x=nat z in exI, simp)
done

instance int :: countable
proof (rule countable-classI [of (λi. to-nat (nat (sgn i + 1), nat (abs i)))],
    auto dest: injD [OF inj-to-nat])
  fix x y
```

```
  assume a: nat (sgn x + 1) = nat (sgn y + 1) nat (abs x) = nat (abs y)
  show x = y
  proof (cases rule: int-pos-neg-zero[of x])
    case zero
    with a show x = y by (cases rule: int-pos-neg-zero[of y]) auto
  next
    case (pos n)
    with a show x = y by (cases rule: int-pos-neg-zero[of y]) auto
  next
    case (neg n)
    with a show x = y by (cases rule: int-pos-neg-zero[of y]) auto
  qed
qed
```

Options

```
instance option :: (countable) countable
by (rule countable-classI[of λx. case x of None ⇒ 0
                                 | Some y ⇒ Suc (to-nat y)])
 (auto split:option.splits)
```

Lists

```
lemma from-nat-to-nat-map [simp]: map from-nat (map to-nat xs) = xs
  by (simp add: comp-def map-compose [symmetric])

primrec
  list-encode :: 'a::countable list ⇒ nat
where
  list-encode [] = 0
| list-encode (x#xs) = Suc (to-nat (x, list-encode xs))

instance list :: (countable) countable
proof (rule countable-classI [of list-encode])
  fix xs ys :: 'a list
  assume cenc: list-encode xs = list-encode ys
  then show xs = ys
  proof (induct xs arbitrary: ys)
    case (Nil ys)
    with cenc show ?case by (cases ys, auto)
  next
    case (Cons x xs' ys)
    thus ?case by (cases ys) auto
  qed
qed
```

Functions

```
instance fun :: (finite, countable) countable
proof
  obtain xs :: 'a list where xs: set xs = UNIV
    using finite-list [OF finite-UNIV] ..
```

show $\exists$ *to-nat*::('*a* $\Rightarrow$ '*b*) $\Rightarrow$ *nat*. *inj to-nat*
proof
show *inj* (λf. *to-nat* (*map f xs*))
by (*rule injI*, *simp add*: *xs expand-fun-eq*)
qed
qed

end

21 Dense-Linear-Order: Dense linear order without endpoints and a quantifier elimination procedure in Ferrante and Rackoff style

theory *Dense-Linear-Order*
imports *Arith-Tools*
uses
~~/*src*/*HOL*/*Tools*/*Qelim*/*qelim.ML*
~~/*src*/*HOL*/*Tools*/*Qelim*/*langford-data.ML*
~~/*src*/*HOL*/*Tools*/*Qelim*/*ferrante-rackoff-data.ML*
(~~/*src*/*HOL*/*Tools*/*Qelim*/*langford.ML*)
(~~/*src*/*HOL*/*Tools*/*Qelim*/*ferrante-rackoff.ML*)
begin

setup *Langford-Data.setup*
setup *Ferrante-Rackoff-Data.setup*

context *linorder*
begin

lemma *less-not-permute*: $\neg$ ($x < y \wedge y < x$) **by** (*simp add*: *not-less linear*)

lemma *gather-simps*:
shows
($\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge x < u \wedge P\ x) \longleftrightarrow (\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in (\mathit{insert}\ u\ U).\ x < y) \wedge P\ x)$
and $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge l < x \wedge P\ x) \longleftrightarrow (\exists x.\ (\forall y \in (\mathit{insert}\ l\ L).\ y < x) \wedge (\forall y \in U.\ x < y) \wedge P\ x)$
($\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge x < u) \longleftrightarrow (\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in (\mathit{insert}\ u\ U).\ x < y))$
and $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge l < x) \longleftrightarrow (\exists x.\ (\forall y \in (\mathit{insert}\ l\ L).\ y < x) \wedge (\forall y \in U.\ x < y))$ **by** *auto*

lemma
gather-start: $(\exists x.\ P\ x) \equiv (\exists x.\ (\forall y \in \{\}.\ y < x) \wedge (\forall y \in \{\}.\ x < y) \wedge P\ x)$
by *simp*

Theorems for $\exists z.\ \forall x.\ x < z \longrightarrow (P\ x \longleftrightarrow P_{-\infty})$

lemma *minf-lt*: $\exists\, z\,.\ \forall\, x.\ x < z \longrightarrow (x < t \longleftrightarrow \mathit{True})$ **by** *auto*
lemma *minf-gt*: $\exists\, z\,.\ \forall\, x.\ x < z \longrightarrow (t < x \longleftrightarrow \mathit{False})$
by (*simp add*: *not-less*) (*rule exI*[**where** x=t], *auto simp add*: *less-le*)

lemma *minf-le*: $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (x \leq t \longleftrightarrow \mathit{True})$ **by** (*auto simp add*: *less-le*)
lemma *minf-ge*: $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (t \leq x \longleftrightarrow \mathit{False})$
by (*auto simp add*: *less-le not-less not-le*)
lemma *minf-eq*: $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (x = t \longleftrightarrow \mathit{False})$ **by** *auto*
lemma *minf-neq*: $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (x \neq t \longleftrightarrow \mathit{True})$ **by** *auto*
lemma *minf-P*: $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (P \longleftrightarrow P)$ **by** *blast*

Theorems for $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (P\ x \longleftrightarrow P_{+\infty})$

lemma *pinf-gt*: $\exists\, z\,.\ \forall\, x.\ z < x \longrightarrow (t < x \longleftrightarrow \mathit{True})$ **by** *auto*
lemma *pinf-lt*: $\exists\, z\,.\ \forall\, x.\ z < x \longrightarrow (x < t \longleftrightarrow \mathit{False})$
by (*simp add*: *not-less*) (*rule exI*[**where** x=t], *auto simp add*: *less-le*)

lemma *pinf-ge*: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (t \leq x \longleftrightarrow \mathit{True})$ **by** (*auto simp add*: *less-le*)
lemma *pinf-le*: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (x \leq t \longleftrightarrow \mathit{False})$
by (*auto simp add*: *less-le not-less not-le*)
lemma *pinf-eq*: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (x = t \longleftrightarrow \mathit{False})$ **by** *auto*
lemma *pinf-neq*: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (x \neq t \longleftrightarrow \mathit{True})$ **by** *auto*
lemma *pinf-P*: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (P \longleftrightarrow P)$ **by** *blast*

lemma *nmi-lt*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{True} \wedge x < t \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)$ **by** *auto*
lemma *nmi-gt*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{False} \wedge t < x \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)$
by (*auto simp add*: *le-less*)
lemma *nmi-le*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{True} \wedge x{\leq}\ t \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)$ **by** *auto*
lemma *nmi-ge*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{False} \wedge t{\leq}\ x \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)$ **by** *auto*
lemma *nmi-eq*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{False} \wedge\ x = t \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)$ **by** *auto*
lemma *nmi-neq*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{True} \wedge x \neq t \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)$ **by** *auto*
lemma *nmi-P*: $\forall\ x.\ {}^{\sim}P \wedge P \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)$ **by** *auto*
lemma *nmi-conj*: $[\![\forall\, x.\ \neg P1' \wedge P1\ x \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)\ ;$
$\forall\, x.\ \neg P2' \wedge P2\ x \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)]\!] \Longrightarrow$
$\forall\, x.\ \neg(P1' \wedge P2') \wedge (P1\ x \wedge P2\ x) \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)$ **by** *auto*
lemma *nmi-disj*: $[\![\forall\, x.\ \neg P1' \wedge P1\ x \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)\ ;$
$\forall\, x.\ \neg P2' \wedge P2\ x \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)]\!] \Longrightarrow$
$\forall\, x.\ \neg(P1' \vee P2') \wedge (P1\ x \vee P2\ x) \longrightarrow (\exists\ u{\in}\ U.\ u \leq x)$ **by** *auto*

lemma *npi-lt*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{False} \wedge\ x < t \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)$ **by** (*auto simp add*: *le-less*)
lemma *npi-gt*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{True} \wedge t < x \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)$ **by** *auto*
lemma *npi-le*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{False} \wedge\ x \leq t \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)$ **by** *auto*
lemma *npi-ge*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{True} \wedge t \leq x \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)$ **by** *auto*
lemma *npi-eq*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{False} \wedge\ x = t \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)$ **by** *auto*
lemma *npi-neq*: $t \in U \Longrightarrow \forall\, x.\ \neg \mathit{True} \wedge x \neq t \longrightarrow (\exists\ u{\in}\ U.\ x \leq u\ )$ **by** *auto*
lemma *npi-P*: $\forall\ x.\ {}^{\sim}P \wedge P \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)$ **by** *auto*
lemma *npi-conj*: $[\![\forall\, x.\ \neg P1' \wedge P1\ x \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)\ ;\ \forall\, x.\ \neg P2' \wedge P2\ x$
$\longrightarrow (\exists\ u{\in}\ U.\ x \leq u)]\!]$
$\Longrightarrow\ \forall\, x.\ \neg(P1' \wedge P2') \wedge (P1\ x \wedge P2\ x) \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)$ **by** *auto*

lemma *npi-disj*: $[\![\forall x.\ \neg P1' \wedge P1\ x \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)\ ;\ \forall x.\ \neg P2' \wedge P2\ x \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)]\!]$
$\Longrightarrow \forall x.\ \neg(P1' \vee P2') \wedge (P1\ x \vee P2\ x) \longrightarrow (\exists\ u{\in}\ U.\ x \leq u)$ **by** *auto*

lemma *lin-dense-lt*: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l{<}\ x \wedge x < u \wedge x < t \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow y < t)$
proof(*clarsimp*)
fix *x l u y* **assume** *tU*: $t \in U$ **and** *noU*: $\forall t.\ l < t \wedge t < u \longrightarrow t \notin U$ **and** *lx*: $l < x$
and *xu*: $x{<}u$ **and** *px*: $x < t$ **and** *ly*: $l{<}y$ **and** *yu*:$y < u$
from *tU noU ly yu* **have** *tny*: $t{\neq}y$ **by** *auto*
{assume *H*: $t < y$
from *less-trans*[*OF lx px*] *less-trans*[*OF H yu*]
have $l < t \wedge t < u$ **by** *simp*
with *tU noU* **have** *False* **by** *auto***}**
hence $\neg\ t < y$ **by** *auto* **hence** $y \leq t$ **by** (*simp add*: *not-less*)
thus $y < t$ **using** *tny* **by** (*simp add*: *less-le*)
qed

lemma *lin-dense-gt*: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t{<}\ u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge t < x \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow t < y)$
proof(*clarsimp*)
fix *x l u y*
assume *tU*: $t \in U$ **and** *noU*: $\forall t.\ l < t \wedge t < u \longrightarrow t \notin U$ **and** *lx*: $l < x$ **and** *xu*: $x{<}u$
and *px*: $t < x$ **and** *ly*: $l{<}y$ **and** *yu*:$y < u$
from *tU noU ly yu* **have** *tny*: $t{\neq}y$ **by** *auto*
{assume *H*: $y{<}\ t$
from *less-trans*[*OF ly H*] *less-trans*[*OF px xu*] **have** $l < t \wedge t < u$ **by** *simp*
with *tU noU* **have** *False* **by** *auto***}**
hence $\neg\ y{<}t$ **by** *auto* **hence** $t \leq y$ **by** (*auto simp add*: *not-less*)
thus $t < y$ **using** *tny* **by** (*simp add*:*less-le*)
qed

lemma *lin-dense-le*: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t{<}\ u \longrightarrow t \notin U) \wedge l{<}\ x \wedge x < u \wedge x \leq t \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow y{\leq}\ t)$
proof(*clarsimp*)
fix *x l u y*
assume *tU*: $t \in U$ **and** *noU*: $\forall t.\ l < t \wedge t < u \longrightarrow t \notin U$ **and** *lx*: $l < x$ **and** *xu*: $x{<}u$
and *px*: $x \leq t$ **and** *ly*: $l{<}y$ **and** *yu*:$y < u$
from *tU noU ly yu* **have** *tny*: $t{\neq}y$ **by** *auto*
{assume *H*: $t < y$
from *less-le-trans*[*OF lx px*] *less-trans*[*OF H yu*]
have $l < t \wedge t < u$ **by** *simp*
with *tU noU* **have** *False* **by** *auto***}**
hence $\neg\ t < y$ **by** *auto* **thus** $y \leq t$ **by** (*simp add*: *not-less*)
qed

lemma *lin-dense-ge*: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge t \leq x \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow t \leq y)$
proof(*clarsimp*)
fix *x l u y*
assume *tU*: $t \in U$ **and** *noU*: $\forall t.\ l < t \wedge t < u \longrightarrow t \notin U$ **and** *lx*: $l < x$ **and** *xu*: $x<u$
and *px*: $t \leq x$ **and** *ly*: $l<y$ **and** *yu*:$y < u$
from *tU noU ly yu* **have** *tny*: $t \neq y$ **by** *auto*
{assume *H*: $y< t$
from *less-trans*[*OF ly H*] *le-less-trans*[*OF px xu*]
have $l < t \wedge t < u$ **by** *simp*
with *tU noU* **have** *False* **by** *auto***}**
hence $\neg\ y<t$ **by** *auto* **thus** $t \leq y$ **by** (*simp add*: *not-less*)
qed
lemma *lin-dense-eq*: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge x = t \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow y = t)$ **by** *auto*
lemma *lin-dense-neq*: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge x \neq t \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow y \neq t)$ **by** *auto*
lemma *lin-dense-P*: $\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow P)$ **by** *auto*

lemma *lin-dense-conj*:
$[\![\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P1\ x \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow P1\ y)$;
$\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P2\ x \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow P2\ y)]\!] \Longrightarrow$
$\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge (P1\ x \wedge P2\ x) \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow (P1\ y \wedge P2\ y))$
by *blast*
lemma *lin-dense-disj*:
$[\![\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P1\ x \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow P1\ y)$;
$\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P2\ x \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow P2\ y)]\!] \Longrightarrow$
$\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge (P1\ x \vee P2\ x) \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow (P1\ y \vee P2\ y))$
by *blast*

lemma *npmibnd*: $[\![\forall x.\ \neg\ MP \wedge P\ x \longrightarrow (\exists\ u \in U.\ u \leq x); \forall x.\ \neg PP \wedge P\ x \longrightarrow (\exists\ u \in U.\ x \leq u)]\!]$
$\Longrightarrow \forall x.\ \neg\ MP \wedge \neg PP \wedge P\ x \longrightarrow (\exists\ u \in U.\ \exists\ u' \in U.\ u \leq x \wedge x \leq u')$
by *auto*

lemma *finite-set-intervals*:
assumes *px*: $P\ x$ **and** *lx*: $l \leq x$ **and** *xu*: $x \leq u$ **and** *linS*: $l \in S$
and *uinS*: $u \in S$ **and** *fS*:*finite S* **and** *lS*: $\forall\ x \in S.\ l \leq x$ **and** *Su*: $\forall\ x \in S.\ x \leq u$
shows $\exists\ a \in S.\ \exists\ b \in S.\ (\forall\ y.\ a < y \wedge y < b \longrightarrow y \notin S) \wedge a \leq x \wedge x \leq b \wedge P\ x$

proof−
let *?Mx* = {*y*. *y*∈ *S* ∧ *y* ≤ *x*}
let *?xM* = {*y*. *y*∈ *S* ∧ *x* ≤ *y*}
let *?a* = *Max ?Mx*
let *?b* = *Min ?xM*
have *MxS*: *?Mx* ⊆ *S* **by** *blast*
hence *fMx*: *finite ?Mx* **using** *fS finite-subset* **by** *auto*
from *lx linS* **have** *linMx*: *l* ∈ *?Mx* **by** *blast*
hence *Mxne*: *?Mx* ≠ {} **by** *blast*
have *xMS*: *?xM* ⊆ *S* **by** *blast*
hence *fxM*: *finite ?xM* **using** *fS finite-subset* **by** *auto*
from *xu uinS* **have** *linxM*: *u* ∈ *?xM* **by** *blast*
hence *xMne*: *?xM* ≠ {} **by** *blast*
have *ax*:*?a* ≤ *x* **using** *Mxne fMx* **by** *auto*
have *xb*:*x* ≤ *?b* **using** *xMne fxM* **by** *auto*
have *?a* ∈ *?Mx* **using** *Max-in*[*OF fMx Mxne*] **by** *simp* **hence** *ainS*: *?a* ∈ *S* **using** *MxS* **by** *blast*
have *?b* ∈ *?xM* **using** *Min-in*[*OF fxM xMne*] **by** *simp* **hence** *binS*: *?b* ∈ *S* **using** *xMS* **by** *blast*
have *noy*:∀ *y*. *?a* < *y* ∧ *y* < *?b* ⟶ *y* ∉ *S*
proof(*clarsimp*)
fix *y* **assume** *ay*: *?a* < *y* **and** *yb*: *y* < *?b* **and** *yS*: *y* ∈ *S*
from *yS* **have** *y*∈ *?Mx* ∨ *y*∈ *?xM* **by** (*auto simp add*: *linear*)
moreover {**assume** *y* ∈ *?Mx* **hence** *y* ≤ *?a* **using** *Mxne fMx* **by** *auto* **with** *ay* **have** *False* **by** (*simp add*: *not-le*[*symmetric*])}
moreover {**assume** *y* ∈ *?xM* **hence** *?b* ≤ *y* **using** *xMne fxM* **by** *auto* **with** *yb* **have** *False* **by** (*simp add*: *not-le*[*symmetric*])}
ultimately show *False* **by** *blast*
qed
from *ainS binS noy ax xb px* **show** *?thesis* **by** *blast*
qed

lemma *finite-set-intervals2*:
assumes *px*: *P x* **and** *lx*: *l* ≤ *x* **and** *xu*: *x* ≤ *u* **and** *linS*: *l*∈ *S*
and *uinS*: *u* ∈ *S* **and** *fS*:*finite S* **and** *lS*: ∀ *x*∈ *S*. *l* ≤ *x* **and** *Su*: ∀ *x*∈ *S*. *x* ≤ *u*
shows (∃ *s*∈ *S*. *P s*) ∨ (∃ *a* ∈ *S*. ∃ *b* ∈ *S*. (∀ *y*. *a* < *y* ∧ *y* < *b* ⟶ *y* ∉ *S*) ∧ *a* < *x* ∧ *x* < *b* ∧ *P x*)
proof−
from *finite-set-intervals*[**where** *P*=*P*, *OF px lx xu linS uinS fS lS Su*]
obtain *a* **and** *b* **where**
as: *a*∈ *S* **and** *bs*: *b*∈ *S* **and** *noS*:∀ *y*. *a* < *y* ∧ *y* < *b* ⟶ *y* ∉ *S*
and *axb*: *a* ≤ *x* ∧ *x* ≤ *b* ∧ *P x* **by** *auto*
from *axb* **have** *x*= *a* ∨ *x*= *b* ∨ (*a* < *x* ∧ *x* < *b*) **by** (*auto simp add*: *le-less*)
thus *?thesis* **using** *px as bs noS* **by** *blast*
qed

end

22 The classical QE after Langford for dense linear orders

context *dense-linear-order*
begin

lemma *dlo-qe-bnds*:
assumes *ne*: $L \neq \{\}$ **and** *neU*: $U \neq \{\}$ **and** *fL*: *finite L* **and** *fU*: *finite U*
shows $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y)) \equiv (\forall\ l \in L.\ \forall u \in U.\ l < u)$
proof (*simp only*: *atomize-eq*, *rule iffI*)
assume *H*: $\exists x.\ (\forall y{\in}L.\ y < x) \wedge (\forall y{\in}U.\ x < y)$
then obtain *x* **where** *xL*: $\forall y{\in}L.\ y < x$ **and** *xU*: $\forall y{\in}U.\ x < y$ **by** *blast*
{fix *l u* **assume** *l*: $l \in L$ **and** *u*: $u \in U$
have $l < x$ **using** *xL l* **by** *blast*
also have $x < u$ **using** *xU u* **by** *blast*
finally (*less-trans*) **have** $l < u$ **.}**
thus $\forall l{\in}L.\ \forall u{\in}U.\ l < u$ **by** *blast*
next
assume *H*: $\forall l{\in}L.\ \forall u{\in}U.\ l < u$
let *?ML* = *Max L*
let *?MU* = *Min U*
from *fL ne* **have** *th1*: *?ML* $\in L$ **and** *th1'*: $\forall l{\in}L.\ l \leq$ *?ML* **by** *auto*
from *fU neU* **have** *th2*: *?MU* $\in U$ **and** *th2'*: $\forall u{\in}U.$ *?MU* $\leq u$ **by** *auto*
from *th1 th2 H* **have** *?ML* $<$ *?MU* **by** *auto*
with *dense* **obtain** *w* **where** *th3*: *?ML* $< w$ **and** *th4*: $w <$ *?MU* **by** *blast*
from *th3 th1'* **have** $\forall l \in L.\ l < w$ **by** *auto*
moreover from *th4 th2'* **have** $\forall u \in U.\ w < u$ **by** *auto*
ultimately show $\exists x.\ (\forall y{\in}L.\ y < x) \wedge (\forall y{\in}U.\ x < y)$ **by** *auto*
qed

lemma *dlo-qe-noub*:
assumes *ne*: $L \neq \{\}$ **and** *fL*: *finite L*
shows $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in \{\}.\ x < y)) \equiv$ *True*
proof(*simp add*: *atomize-eq*)
from *gt-ex*[*of Max L*] **obtain** *M* **where** *M*: *Max L* $<$ *M* **by** *blast*
from *ne fL* **have** $\forall x \in L.\ x \leq$ *Max L* **by** *simp*
with *M* **have** $\forall x{\in}L.\ x < M$ **by** (*auto intro*: *le-less-trans*)
thus $\exists x.\ \forall y{\in}L.\ y < x$ **by** *blast*
qed

lemma *dlo-qe-nolb*:
assumes *ne*: $U \neq \{\}$ **and** *fU*: *finite U*
shows $(\exists x.\ (\forall y \in \{\}.\ y < x) \wedge (\forall y \in U.\ x < y)) \equiv$ *True*
proof(*simp add*: *atomize-eq*)
from *lt-ex*[*of Min U*] **obtain** *M* **where** *M*: $M <$ *Min U* **by** *blast*
from *ne fU* **have** $\forall x \in U.$ *Min U* $\leq x$ **by** *simp*
with *M* **have** $\forall x{\in}U.\ M < x$ **by** (*auto intro*: *less-le-trans*)
thus $\exists x.\ \forall y{\in}U.\ x < y$ **by** *blast*
qed

lemma *exists-neq*: $\exists (x::'a).\ x \neq t\ \exists (x::'a).\ t \neq x$
using *gt-ex*[*of t*] **by** *auto*

lemmas *dlo-simps* = *order-refl less-irrefl not-less not-le exists-neq*
le-less neq-iff linear less-not-permute

lemma *axiom*: *dense-linear-order* (*op* $\leq$) (*op* $<$) **by** (*rule dense-linear-order-axioms*)
lemma *atoms*:
includes *meta-term-syntax*
shows *TERM* (*less* :: $'a \Rightarrow$ -)
and *TERM* (*less-eq* :: $'a \Rightarrow$ -)
and *TERM* (*op* = :: $'a \Rightarrow$ -) **.**

declare *axiom*[*langford qe*: *dlo-qe-bnds dlo-qe-nolb dlo-qe-noub gather*: *gather-start gather-simps atoms*: *atoms*]
declare *dlo-simps*[*langfordsimp*]

end

lemma *dnf*:
$(P\ \&\ (Q \mid R)) = ((P\&Q) \mid (P\&R))$
$((Q \mid R)\ \&\ P) = ((Q\&P) \mid (R\&P))$
by *blast+*

lemmas *weak-dnf-simps* = *simp-thms dnf*

lemma *nnf-simps*:
$(\neg(P \wedge Q)) = (\neg P \vee \neg Q)\ (\neg(P \vee Q)) = (\neg P \wedge \neg Q)\ (P \longrightarrow Q) = (\neg P \vee Q)$
$(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q))\ (\neg \neg(P)) = P$
by *blast+*

lemma *ex-distrib*: $(\exists x.\ P\ x \vee Q\ x) \longleftrightarrow ((\exists x.\ P\ x) \vee (\exists x.\ Q\ x))$ **by** *blast*

lemmas *dnf-simps* = *weak-dnf-simps nnf-simps ex-distrib*

use ~~/*src/HOL/Tools/Qelim/langford.ML*
method-setup *dlo* = ⟪
Method.ctxt-args (*Method.SIMPLE-METHOD' o LangfordQE.dlo-tac*)
⟫ *Langford's algorithm for quantifier elimination in dense linear orders*

23 Contructive dense linear orders yield QE for linear arithmetic over ordered Fields – see *Arith-Tools.thy*

Linear order without upper bounds

locale *linorder-stupid-syntax* = *linorder*
begin

notation
less-eq (*op* $\sqsubseteq$) **and**
less-eq ((-/ $\sqsubseteq$ -) [*51*, *51*] *50*) **and**
less (*op* $\sqsubset$) **and**
less ((-/ $\sqsubset$ -) [*51*, *51*] *50*)

end

locale *linorder-no-ub* = *linorder-stupid-syntax* +
assumes *gt-ex*: $\exists\, y.$ *less x y*
begin
lemma *ge-ex*: $\exists\, y.\ x \sqsubseteq y$ **using** *gt-ex* **by** *auto*

Theorems for $\exists\, z.\ \forall\, x.\ z \sqsubset x \longrightarrow (P\ x \longleftrightarrow P_{+\infty})$

lemma *pinf-conj*:
assumes *ex1*: $\exists\, z1.\ \forall\, x.\ z1 \sqsubset x \longrightarrow (P1\ x \longleftrightarrow P1')$
and *ex2*: $\exists\, z2.\ \forall\, x.\ z2 \sqsubset x \longrightarrow (P2\ x \longleftrightarrow P2')$
shows $\exists\, z.\ \forall\, x.\ z \sqsubset\ x \longrightarrow ((P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2'))$
proof−
from *ex1 ex2* **obtain** *z1* **and** *z2* **where** *z1*: $\forall\, x.\ z1 \sqsubset x \longrightarrow (P1\ x \longleftrightarrow P1')$
and *z2*: $\forall\, x.\ z2 \sqsubset x \longrightarrow (P2\ x \longleftrightarrow P2')$ **by** *blast*
from *gt-ex* **obtain** *z* **where** *z*:*ord.max less-eq z1 z2* $\sqsubset z$ **by** *blast*
from *z* **have** *zz1*: *z1* $\sqsubset z$ **and** *zz2*: *z2* $\sqsubset z$ **by** *simp-all*
{fix *x* **assume** *H*: $z \sqsubset x$
from *less-trans*[*OF zz1 H*] *less-trans*[*OF zz2 H*]
have $(P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2')$ **using** *z1 zz1 z2 zz2* **by** *auto*
}
thus *?thesis* **by** *blast*
qed

lemma *pinf-disj*:
assumes *ex1*: $\exists\, z1.\ \forall\, x.\ z1 \sqsubset x \longrightarrow (P1\ x \longleftrightarrow P1')$
and *ex2*: $\exists\, z2.\ \forall\, x.\ z2 \sqsubset x \longrightarrow (P2\ x \longleftrightarrow P2')$
shows $\exists\, z.\ \forall\, x.\ z \sqsubset\ x \longrightarrow ((P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2'))$
proof−
from *ex1 ex2* **obtain** *z1* **and** *z2* **where** *z1*: $\forall\, x.\ z1 \sqsubset x \longrightarrow (P1\ x \longleftrightarrow P1')$
and *z2*: $\forall\, x.\ z2 \sqsubset x \longrightarrow (P2\ x \longleftrightarrow P2')$ **by** *blast*
from *gt-ex* **obtain** *z* **where** *z*:*ord.max less-eq z1 z2* $\sqsubset z$ **by** *blast*
from *z* **have** *zz1*: *z1* $\sqsubset z$ **and** *zz2*: *z2* $\sqsubset z$ **by** *simp-all*
{fix *x* **assume** *H*: $z \sqsubset x$
from *less-trans*[*OF zz1 H*] *less-trans*[*OF zz2 H*]
have $(P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2')$ **using** *z1 zz1 z2 zz2* **by** *auto*
}
thus *?thesis* **by** *blast*
qed

lemma *pinf-ex*: **assumes** *ex*:$\exists\, z.\ \forall\, x.\ z \sqsubset x \longrightarrow (P\ x \longleftrightarrow P1)$ **and** *p1*: *P1* **shows** $\exists\ x.\ P\ x$
proof−

from *ex* **obtain** *z* **where** *z*: $\forall x.\ z \sqsubset x \longrightarrow (P\ x \longleftrightarrow P1)$ **by** *blast*
from *gt-ex* **obtain** *x* **where** *x*: $z \sqsubset x$ **by** *blast*
from *z x p1* **show** *?thesis* **by** *blast*
qed

end

Linear order without upper bounds

locale *linorder-no-lb* = *linorder-stupid-syntax* +
assumes *lt-ex*: $\exists y.\ less\ y\ x$
begin
lemma *le-ex*: $\exists y.\ y \sqsubseteq x$ **using** *lt-ex* **by** *auto*

Theorems for $\exists z.\ \forall x.\ x \sqsubset z \longrightarrow (P\ x \longleftrightarrow P_{-\infty})$

lemma *minf-conj*:
assumes *ex1*: $\exists z1.\ \forall x.\ x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$
and *ex2*: $\exists z2.\ \forall x.\ x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$
shows $\exists z.\ \forall x.\ x \sqsubset\ z \longrightarrow ((P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2'))$
proof−
from *ex1 ex2* **obtain** *z1* **and** *z2* **where** *z1*: $\forall x.\ x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$**and** *z2*: $\forall x.\ x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$ **by** *blast*
from *lt-ex* **obtain** *z* **where** *z*:$z \sqsubset$ *ord.min less-eq z1 z2* **by** *blast*
from *z* **have** *zz1*: $z \sqsubset z1$ **and** *zz2*: $z \sqsubset z2$ **by** *simp-all*
{fix *x* **assume** *H*: $x \sqsubset z$
from *less-trans*[*OF H zz1*] *less-trans*[*OF H zz2*]
have $(P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2')$ **using** *z1 zz1 z2 zz2* **by** *auto*
}
thus *?thesis* **by** *blast*
qed

lemma *minf-disj*:
assumes *ex1*: $\exists z1.\ \forall x.\ x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$
and *ex2*: $\exists z2.\ \forall x.\ x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$
shows $\exists z.\ \forall x.\ x \sqsubset\ z \longrightarrow ((P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2'))$
proof−
from *ex1 ex2* **obtain** *z1* **and** *z2* **where** *z1*: $\forall x.\ x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$**and** *z2*: $\forall x.\ x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$ **by** *blast*
from *lt-ex* **obtain** *z* **where** *z*:$z \sqsubset$ *ord.min less-eq z1 z2* **by** *blast*
from *z* **have** *zz1*: $z \sqsubset z1$ **and** *zz2*: $z \sqsubset z2$ **by** *simp-all*
{fix *x* **assume** *H*: $x \sqsubset z$
from *less-trans*[*OF H zz1*] *less-trans*[*OF H zz2*]
have $(P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2')$ **using** *z1 zz1 z2 zz2* **by** *auto*
}
thus *?thesis* **by** *blast*
qed

lemma *minf-ex*: **assumes** *ex*:$\exists z.\ \forall x.\ x \sqsubset z \longrightarrow (P\ x \longleftrightarrow P1)$ **and** *p1*: *P1*
shows $\exists\ x.\ P\ x$
proof−

from *ex* **obtain** *z* **where** *z*: $\forall x.\ x \sqsubset z \longrightarrow (P\ x \longleftrightarrow P1)$ **by** *blast*
from *lt-ex* **obtain** *x* **where** *x*: $x \sqsubset z$ **by** *blast*
from *z x p1* **show** *?thesis* **by** *blast*
qed

end

locale *constr-dense-linear-order* = *linorder-no-lb* + *linorder-no-ub* +
fixes *between*
assumes *between-less*: *less x y* $\Longrightarrow$ *less x* (*between x y*) $\wedge$ *less* (*between x y*) *y*
and *between-same*: *between x x* = *x*

interpretation *constr-dense-linear-order* < *dense-linear-order*
apply *unfold-locales*
using *gt-ex lt-ex between-less*
by (*auto*, *rule-tac x*=*between x y* **in** *exI*, *simp*)

context *constr-dense-linear-order*
begin

lemma *rinf-U*:
assumes *fU*: *finite U*
and *lin-dense*: $\forall x\ l\ u.\ (\forall\ t.\ l \sqsubset t \wedge t \sqsubset u \longrightarrow t \notin U) \wedge l \sqsubset x \wedge x \sqsubset u \wedge P\ x$
$\longrightarrow (\forall\ y.\ l \sqsubset y \wedge y \sqsubset u \longrightarrow P\ y\ )$
and *nmpiU*: $\forall x.\ \neg\ MP \wedge \neg PP \wedge P\ x \longrightarrow (\exists\ u \in U.\ \exists\ u' \in U.\ u \sqsubseteq x \wedge x \sqsubseteq u')$
and *nmi*: $\neg\ MP$ **and** *npi*: $\neg\ PP$ **and** *ex*: $\exists\ x.\ \ P\ x$
shows $\exists\ u \in U.\ \exists\ u' \in U.\ P$ (*between u u'*)
proof−
from *ex* **obtain** *x* **where** *px*: *P x* **by** *blast*
from *px nmi npi nmpiU* **have** $\exists\ u \in U.\ \exists\ u' \in U.\ u \sqsubseteq x \wedge x \sqsubseteq u'$ **by** *auto*
then obtain *u* **and** *u'* **where** *uU*:$u \in U$ **and** *uU'*: $u' \in U$ **and** *ux*:$u \sqsubseteq x$ **and** *xu'*:$x \sqsubseteq u'$ **by** *auto*
from *uU* **have** *Une*: $U \neq \{\}$ **by** *auto*
term *linorder.Min less-eq*
let *?l* = *linorder.Min less-eq U*
let *?u* = *linorder.Max less-eq U*
have *linM*: *?l* $\in U$ **using** *fU Une* **by** *simp*
have *uinM*: *?u* $\in U$ **using** *fU Une* **by** *simp*
have *lM*: $\forall\ t \in U.$ *?l* $\sqsubseteq t$ **using** *Une fU* **by** *auto*
have *Mu*: $\forall\ t \in U.\ t \sqsubseteq$ *?u* **using** *Une fU* **by** *auto*
have *th*:*?l* $\sqsubseteq u$ **using** *uU Une lM* **by** *auto*
from *order-trans*[*OF th ux*] **have** *lx*: *?l* $\sqsubseteq x$.
have *th*: $u' \sqsubseteq$ *?u* **using** *uU' Une Mu* **by** *simp*
from *order-trans*[*OF xu' th*] **have** *xu*: $x \sqsubseteq$ *?u* .
from *finite-set-intervals2*[**where** *P*=*P*,*OF px lx xu linM uinM fU lM Mu*]
have $(\exists\ s \in U.\ P\ s) \vee$
$(\exists\ t1 \in U.\ \exists\ t2 \in U.\ (\forall\ y.\ t1 \sqsubset y \wedge y \sqsubset t2 \longrightarrow y \notin U) \wedge t1 \sqsubset x \wedge x \sqsubset$

$t2 \wedge P\ x$) **.**
moreover { fix u **assume** um: $u \in U$ **and** pu: $P\ u$
have $between\ u\ u = u$ **by** ($simp\ add$: $between\text{-}same$)
with $um\ pu$ **have** P ($between\ u\ u$) **by** $simp$
with um **have** $?thesis$ **by** $blast$**}**
moreover{
assume $\exists\ t1 \in U.\ \exists\ t2 \in U.\ (\forall\ y.\ t1 \sqsubset y \wedge y \sqsubset t2 \longrightarrow y \notin U) \wedge t1 \sqsubset x \wedge x \sqsubset t2 \wedge P\ x$
then obtain $t1$ **and** $t2$ **where** $t1M$: $t1 \in U$ **and** $t2M$: $t2 \in U$
and noM: $\forall\ y.\ t1 \sqsubset y \wedge y \sqsubset t2 \longrightarrow y \notin U$ **and** $t1x$: $t1 \sqsubset x$ **and** $xt2$: $x \sqsubset t2$ **and** px: $P\ x$
by $blast$
from $less\text{-}trans[OF\ t1x\ xt2]$ **have** $t1t2$: $t1 \sqsubset t2$ **.**
let $?u = between\ t1\ t2$
from $between\text{-}less\ t1t2$ **have** $t1lu$: $t1 \sqsubset ?u$ **and** $ut2$: $?u \sqsubset t2$ **by** $auto$
from $lin\text{-}dense\ noM\ t1x\ xt2\ px\ t1lu\ ut2$ **have** $P\ ?u$ **by** $blast$
with $t1M\ t2M$ **have** $?thesis$ **by** $blast$**}**
ultimately show $?thesis$ **by** $blast$
qed

theorem $fr\text{-}eq$:
assumes fU: $finite\ U$
and $lin\text{-}dense$: $\forall x\ l\ u.\ (\forall\ t.\ l \sqsubset t \wedge t \sqsubset u \longrightarrow t \notin U) \wedge l \sqsubset x \wedge x \sqsubset u \wedge P\ x \longrightarrow (\forall\ y.\ l \sqsubset y \wedge y \sqsubset u \longrightarrow P\ y\ )$
and $nmibnd$: $\forall x.\ \neg\ MP \wedge P\ x \longrightarrow (\exists\ u \in U.\ u \sqsubseteq x)$
and $npibnd$: $\forall x.\ \neg PP \wedge P\ x \longrightarrow (\exists\ u \in U.\ x \sqsubseteq u)$
and mi: $\exists z.\ \forall x.\ x \sqsubset z \longrightarrow (P\ x = MP)$ **and** pi: $\exists z.\ \forall x.\ z \sqsubset x \longrightarrow (P\ x = PP)$
shows $(\exists\ x.\ P\ x) \equiv (MP \vee PP \vee (\exists\ u \in U.\ \exists\ u' \in U.\ P\ (between\ u\ u')))$
(**is** - $\equiv$ (- $\vee$ - $\vee$ $?F$) **is** $?E \equiv ?D$)
proof−
{
assume px: $\exists\ x.\ P\ x$
have $MP \vee PP \vee (\neg\ MP \wedge \neg\ PP)$ **by** $blast$
moreover {assume $MP \vee PP$ **hence** $?D$ **by** $blast$**}**
moreover {assume nmi: $\neg\ MP$ **and** npi: $\neg\ PP$
from $npmibnd[OF\ nmibnd\ npibnd]$
have $nmpiU$: $\forall x.\ \neg\ MP \wedge \neg PP \wedge P\ x \longrightarrow (\exists\ u \in U.\ \exists\ u' \in U.\ u \sqsubseteq x \wedge x \sqsubseteq u')$ **.**
from $rinf\text{-}U[OF\ fU\ lin\text{-}dense\ nmpiU\ nmi\ npi\ px]$ **have** $?D$ **by** $blast$**}**
ultimately have $?D$ **by** $blast$**}**
moreover
{ assume $?D$
moreover {assume m:MP **from** $minf\text{-}ex[OF\ mi\ m]$ **have** $?E$ **.}**
moreover {assume p: PP **from** $pinf\text{-}ex[OF\ pi\ p]$ **have** $?E$ **. }**
moreover {assume f:$?F$ **hence** $?E$ **by** $blast$**}**
ultimately have $?E$ **by** $blast$**}**
ultimately have $?E = ?D$ **by** $blast$ **thus** $?E \equiv ?D$ **by** $simp$
qed

lemmas *minf-thms = minf-conj minf-disj minf-eq minf-neq minf-lt minf-le minf-gt minf-ge minf-P*
lemmas *pinf-thms = pinf-conj pinf-disj pinf-eq pinf-neq pinf-lt pinf-le pinf-gt pinf-ge pinf-P*

lemmas *nmi-thms = nmi-conj nmi-disj nmi-eq nmi-neq nmi-lt nmi-le nmi-gt nmi-ge nmi-P*
lemmas *npi-thms = npi-conj npi-disj npi-eq npi-neq npi-lt npi-le npi-gt npi-ge npi-P*
lemmas *lin-dense-thms = lin-dense-conj lin-dense-disj lin-dense-eq lin-dense-neq lin-dense-lt lin-dense-le lin-dense-gt lin-dense-ge lin-dense-P*

lemma *ferrack-axiom*: *constr-dense-linear-order less-eq less between*
by (*rule constr-dense-linear-order-axioms*)
lemma *atoms*:
includes *meta-term-syntax*
shows *TERM* (*less* :: *'a* ⇒ -)
and *TERM* (*less-eq* :: *'a* ⇒ -)
and *TERM* (*op =* :: *'a* ⇒ -) **.**

declare *ferrack-axiom* [*ferrack minf*: *minf-thms pinf*: *pinf-thms*
nmi: *nmi-thms npi*: *npi-thms lindense*:
lin-dense-thms qe: *fr-eq atoms*: *atoms*]

declaration ⟪
let
fun simps phi = map (*Morphism.thm phi*) [@{*thm not-less*}, @{*thm not-le*}]
fun generic-whatis phi =
let
val [*lt*, *le*] = *map* (*Morphism.term phi*) [@{*term op* ⊏}, @{*term op* ⊑}]
fun h x t =
case term-of t of
Const(*op =*, -)$*y*$*z* => *if term-of x aconv y then Ferrante-Rackoff-Data.Eq*
else Ferrante-Rackoff-Data.Nox
| @{*term Not*}$(*Const*(*op =*, -)$*y*$*z*) => *if term-of x aconv y then Ferrante-Rackoff-Data.NEq*
else Ferrante-Rackoff-Data.Nox
| *b*$*y*$*z* => *if Term.could-unify* (*b*, *lt*) *then*
if term-of x aconv y then Ferrante-Rackoff-Data.Lt
else if term-of x aconv z then Ferrante-Rackoff-Data.Gt
else Ferrante-Rackoff-Data.Nox
else if Term.could-unify (*b*, *le*) *then*
if term-of x aconv y then Ferrante-Rackoff-Data.Le
else if term-of x aconv z then Ferrante-Rackoff-Data.Ge
else Ferrante-Rackoff-Data.Nox
else Ferrante-Rackoff-Data.Nox
| - => *Ferrante-Rackoff-Data.Nox*
in h end
fun ss phi = HOL-ss addsimps (*simps phi*)

in
Ferrante-Rackoff-Data.funs @{*thm ferrack-axiom*}
{*isolate-conv* = *K* (*K* (*K Thm.reflexive*)), *whatis* = *generic-whatis*, *simpset* = *ss*}
end
⟩⟩

end

use ~~/*src*/*HOL*/*Tools*/*Qelim*/*ferrante-rackoff.ML*

method-setup *ferrack* = ⟪
Method.ctxt-args (*Method.SIMPLE-METHOD'* o *FerranteRackoff.dlo-tac*)
⟫ *Ferrante and Rackoff's algorithm for quantifier elimination in dense linear orders*

23.1 Ferrante and Rackoff algorithm over ordered fields

lemma *neg-prod-lt*:(*c*::'*a*::*ordered-field*) < *0* ⟹ ((*c*∗*x* < *0*) == (*x* > *0*))
proof−
assume *H*: *c* < *0*
have *c*∗*x* < *0* = (*0*/*c* < *x*) **by** (*simp only*: *neg-divide-less-eq*[*OF H*] *ring-simps*)
also have ... = (*0* < *x*) **by** *simp*
finally show (*c*∗*x* < *0*) == (*x* > *0*) **by** *simp*
qed

lemma *pos-prod-lt*:(*c*::'*a*::*ordered-field*) > *0* ⟹ ((*c*∗*x* < *0*) == (*x* < *0*))
proof−
assume *H*: *c* > *0*
hence *c*∗*x* < *0* = (*0*/*c* > *x*) **by** (*simp only*: *pos-less-divide-eq*[*OF H*] *ring-simps*)
also have ... = (*0* > *x*) **by** *simp*
finally show (*c*∗*x* < *0*) == (*x* < *0*) **by** *simp*
qed

lemma *neg-prod-sum-lt*: (*c*::'*a*::*ordered-field*) < *0* ⟹ ((*c*∗*x* + *t*< *0*) == (*x* > (− *1*/*c*)∗*t*))
proof−
assume *H*: *c* < *0*
have *c*∗*x* + *t*< *0* = (*c*∗*x* < −*t*) **by** (*subst less-iff-diff-less-0* [*of c*∗*x* −*t*], *simp*)
also have ... = (−*t*/*c* < *x*) **by** (*simp only*: *neg-divide-less-eq*[*OF H*] *ring-simps*)
also have ... = ((− *1*/*c*)∗*t* < *x*) **by** *simp*
finally show (*c*∗*x* + *t* < *0*) == (*x* > (− *1*/*c*)∗*t*) **by** *simp*
qed

lemma *pos-prod-sum-lt*:(*c*::'*a*::*ordered-field*) > *0* ⟹ ((*c*∗*x* + *t* < *0*) == (*x* < (− *1*/*c*)∗*t*))
proof−
assume *H*: *c* > *0*
have *c*∗*x* + *t*< *0* = (*c*∗*x* < −*t*) **by** (*subst less-iff-diff-less-0* [*of c*∗*x* −*t*], *simp*)
also have ... = (−*t*/*c* > *x*) **by** (*simp only*: *pos-less-divide-eq*[*OF H*] *ring-simps*)

also have ... = $((-\ 1/c)*t > x)$ **by** *simp*
finally show $(c*x + t < 0) == (x < (-\ 1/c)*t)$ **by** *simp*
qed

lemma *sum-lt*:$((x::'a::pordered\text{-}ab\text{-}group\text{-}add) + t < 0) == (x < -\ t)$
using *less-diff-eq*[**where** $a= x$ **and** $b=t$ **and** $c=0$] **by** *simp*

lemma *neg-prod-le*:$(c::'a::ordered\text{-}field) < 0 \Longrightarrow ((c*x <= 0) == (x >= 0))$
proof−
assume *H*: $c < 0$
have $c*x <= 0 = (0/c <= x)$ **by** (*simp only*: *neg-divide-le-eq*[*OF H*] *ring-simps*)
also have ... = $(0 <= x)$ **by** *simp*
finally show $(c*x <= 0) == (x >= 0)$ **by** *simp*
qed

lemma *pos-prod-le*:$(c::'a::ordered\text{-}field) > 0 \Longrightarrow ((c*x <= 0) == (x <= 0))$
proof−
assume *H*: $c > 0$
hence $c*x <= 0 = (0/c >= x)$ **by** (*simp only*: *pos-le-divide-eq*[*OF H*] *ring-simps*)
also have ... = $(0 >= x)$ **by** *simp*
finally show $(c*x <= 0) == (x <= 0)$ **by** *simp*
qed

lemma *neg-prod-sum-le*: $(c::'a::ordered\text{-}field) < 0 \Longrightarrow ((c*x + t <= 0) == (x >= (-\ 1/c)*t))$
proof−
assume *H*: $c < 0$
have $c*x + t <= 0 = (c*x <= -t)$ **by** (*subst le-iff-diff-le-0* [*of* $c*x\ -t$], *simp*)
also have ... = $(-t/c <= x)$ **by** (*simp only*: *neg-divide-le-eq*[*OF H*] *ring-simps*)
also have ... = $((-\ 1/c)*t <= x)$ **by** *simp*
finally show $(c*x + t <= 0) == (x >= (-\ 1/c)*t)$ **by** *simp*
qed

lemma *pos-prod-sum-le*:$(c::'a::ordered\text{-}field) > 0 \Longrightarrow ((c*x + t <= 0) == (x <= (-\ 1/c)*t))$
proof−
assume *H*: $c > 0$
have $c*x + t <= 0 = (c*x <= -t)$ **by** (*subst le-iff-diff-le-0* [*of* $c*x\ -t$], *simp*)
also have ... = $(-t/c >= x)$ **by** (*simp only*: *pos-le-divide-eq*[*OF H*] *ring-simps*)
also have ... = $((-\ 1/c)*t >= x)$ **by** *simp*
finally show $(c*x + t <= 0) == (x <= (-\ 1/c)*t)$ **by** *simp*
qed

lemma *sum-le*:$((x::'a::pordered\text{-}ab\text{-}group\text{-}add) + t <= 0) == (x <= -\ t)$
using *le-diff-eq*[**where** $a= x$ **and** $b=t$ **and** $c=0$] **by** *simp*

lemma *nz-prod-eq*:$(c::'a::ordered\text{-}field) \neq 0 \Longrightarrow ((c*x = 0) == (x = 0))$ **by** *simp*
lemma *nz-prod-sum-eq*: $(c::'a::ordered\text{-}field) \neq 0 \Longrightarrow ((c*x + t = 0) == (x = (-\ 1/c)*t))$

proof−
assume *H*: $c \neq 0$
have $c*x + t = 0 = (c*x = -t)$ **by** (*subst eq-iff-diff-eq-0* [*of* $c*x$ $-t$], *simp*)
also have $\ldots = (x = -t/c)$ **by** (*simp only*: *nonzero-eq-divide-eq*[*OF H*] *ring-simps*)
finally show $(c*x + t = 0) == (x = (-1/c)*t)$ **by** *simp*
qed
lemma *sum-eq*:$((x::'a::pordered\text{-}ab\text{-}group\text{-}add) + t = 0) == (x = -t)$
using *eq-diff-eq*[**where** *a*= *x* **and** *b*=*t* **and** *c*=*0*] **by** *simp*

interpretation *class-ordered-field-dense-linear-order*: *constr-dense-linear-order*
[*op* <= *op* <
λ *x y*. *1/2* $*$ (($x::'a::\{ordered\text{-}field,recpower,number\text{-}ring\}$) + *y*)]
proof (*unfold-locales*, *dlo*, *dlo*, *auto*)
fix *x y*::′*a* **assume** *lt*: $x < y$
from *less-half-sum*[*OF lt*] **show** $x < (x + y)$ */2* **by** *simp*
next
fix *x y*::′*a* **assume** *lt*: $x < y$
from *gt-half-sum*[*OF lt*] **show** $(x + y)$ */2* $< y$ **by** *simp*
qed

declaration⟪
let
fun earlier [] *x y* = *false*
| *earlier* (*h*::*t*) *x y* =
if h aconvc y then false else if h aconvc x then true else earlier t x y;

fun dest-frac ct = *case term-of ct of*
Const (@{*const-name HOL.divide*},-) $ *a* $ *b*=>
Rat.rat-of-quotient (*snd* (*HOLogic.dest-number a*), *snd* (*HOLogic.dest-number b*))
| *t* => *Rat.rat-of-int* (*snd* (*HOLogic.dest-number t*))

fun mk-frac phi cT x =
let val (*a*, *b*) = *Rat.quotient-of-rat x*
in if b = *1 then Numeral.mk-cnumber cT a*
else Thm.capply
(*Thm.capply* (*Drule.cterm-rule* (*instantiate'* [*SOME cT*] []) @{*cpat op* /})
(*Numeral.mk-cnumber cT a*))
(*Numeral.mk-cnumber cT b*)
end

fun whatis x ct = *case term-of ct of*
Const(@{*const-name HOL.plus*}, -)$(*Const*(@{*const-name HOL.times*},-)$-$*y*)$-
=>
if y aconv term-of x then (*c*∗*x*+*t*,[(*funpow 2 Thm.dest-arg1*) *ct*, *Thm.dest-arg ct*])
else (*Nox*,[])
| *Const*(@{*const-name HOL.plus*}, -)$*y*$- =>

```
    if y aconv term-of x then (x+t,[Thm.dest-arg ct])
    else (Nox,[])
| Const(@{const-name HOL.times}, -)$-$y =>
    if y aconv term-of x then (c*x,[Thm.dest-arg1 ct])
    else (Nox,[])
| t => if t aconv term-of x then (x,[]) else (Nox,[]);

fun xnormalize-conv ctxt [] ct = reflexive ct
| xnormalize-conv ctxt (vs as (x::-)) ct =
  case term-of ct of
  Const(@{const-name HOL.less},-)$-$Const(@{const-name HOL.zero},-) =>
   (case whatis x (Thm.dest-arg1 ct) of
   (c*x+t,[c,t]) =>
     let
      val cr = dest-frac c
      val clt = Thm.dest-fun2 ct
      val cz = Thm.dest-arg ct
      val neg = cr </ Rat.zero
      val cthp = Simplifier.rewrite (local-simpset-of ctxt)
            (Thm.capply @{cterm Trueprop}
              (if neg then Thm.capply (Thm.capply clt c) cz
               else Thm.capply (Thm.capply clt cz) c))
      val cth = equal-elim (symmetric cthp) TrueI
       val th = implies-elim (instantiate' [SOME (ctyp-of-term x)] (map SOME
[c,x,t])
           (if neg then @{thm neg-prod-sum-lt} else @{thm pos-prod-sum-lt})) cth
      val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
             (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
    in rth end
  | (x+t,[t]) =>
     let
      val T = ctyp-of-term x
      val th = instantiate' [SOME T] [SOME x, SOME t] @{thm sum-lt}
      val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
           (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
     in  rth end
  | (c*x,[c]) =>
     let
      val cr = dest-frac c
      val clt = Thm.dest-fun2 ct
      val cz = Thm.dest-arg ct
      val neg = cr </ Rat.zero
      val cthp = Simplifier.rewrite (local-simpset-of ctxt)
            (Thm.capply @{cterm Trueprop}
              (if neg then Thm.capply (Thm.capply clt c) cz
               else Thm.capply (Thm.capply clt cz) c))
      val cth = equal-elim (symmetric cthp) TrueI
       val th = implies-elim (instantiate' [SOME (ctyp-of-term x)] (map SOME
[c,x])
```

```
        (if neg then @{thm neg-prod-lt} else @{thm pos-prod-lt})) cth
    val rth = th
  in rth end
 | - => reflexive ct)


| Const(@{const-name HOL.less-eq},-)$-$Const(@{const-name HOL.zero},-) =>
 (case whatis x (Thm.dest-arg1 ct) of
  (c*x+t,[c,t]) =>
    let
     val T = ctyp-of-term x
     val cr = dest-frac c
     val clt = Drule.cterm-rule (instantiate' [SOME T] []) @{cpat op <}
     val cz = Thm.dest-arg ct
     val neg = cr </ Rat.zero
     val cthp = Simplifier.rewrite (local-simpset-of ctxt)
           (Thm.capply @{cterm Trueprop}
             (if neg then Thm.capply (Thm.capply clt c) cz
               else Thm.capply (Thm.capply clt cz) c))
     val cth = equal-elim (symmetric cthp) TrueI
     val th = implies-elim (instantiate' [SOME T] (map SOME [c,x,t])
        (if neg then @{thm neg-prod-sum-le} else @{thm pos-prod-sum-le})) cth
     val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
            (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
   in rth end
 | (x+t,[t]) =>
    let
     val T = ctyp-of-term x
     val th = instantiate' [SOME T] [SOME x, SOME t] @{thm sum-le}
     val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
          (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
    in  rth end
 | (c*x,[c]) =>
    let
     val T = ctyp-of-term x
     val cr = dest-frac c
     val clt = Drule.cterm-rule (instantiate' [SOME T] []) @{cpat op <}
     val cz = Thm.dest-arg ct
     val neg = cr </ Rat.zero
     val cthp = Simplifier.rewrite (local-simpset-of ctxt)
           (Thm.capply @{cterm Trueprop}
             (if neg then Thm.capply (Thm.capply clt c) cz
               else Thm.capply (Thm.capply clt cz) c))
     val cth = equal-elim (symmetric cthp) TrueI
      val th = implies-elim (instantiate' [SOME (ctyp-of-term x)] (map SOME
[c,x])
        (if neg then @{thm neg-prod-le} else @{thm pos-prod-le})) cth
     val rth = th
   in rth end
```

```
    | - => reflexive ct)

|  Const(op =,-)$-$Const(@{const-name HOL.zero},-) =>
   (case whatis x (Thm.dest-arg1 ct) of
    (c*x+t,[c,t]) =>
      let
       val T = ctyp-of-term x
       val cr = dest-frac c
       val ceq = Thm.dest-fun2 ct
       val cz = Thm.dest-arg ct
       val cthp = Simplifier.rewrite (local-simpset-of ctxt)
           (Thm.capply @{cterm Trueprop}
            (Thm.capply @{cterm Not} (Thm.capply (Thm.capply ceq c) cz)))
       val cth = equal-elim (symmetric cthp) TrueI
       val th = implies-elim
             (instantiate' [SOME T] (map SOME [c,x,t]) @{thm nz-prod-sum-eq})
cth
       val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
                 (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
      in rth end
   | (x+t,[t]) =>
      let
       val T = ctyp-of-term x
       val th = instantiate' [SOME T] [SOME x, SOME t] @{thm sum-eq}
       val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
            (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
      in  rth end
   | (c*x,[c]) =>
      let
       val T = ctyp-of-term x
       val cr = dest-frac c
       val ceq = Thm.dest-fun2 ct
       val cz = Thm.dest-arg ct
       val cthp = Simplifier.rewrite (local-simpset-of ctxt)
           (Thm.capply @{cterm Trueprop}
            (Thm.capply @{cterm Not} (Thm.capply (Thm.capply ceq c) cz)))
       val cth = equal-elim (symmetric cthp) TrueI
       val rth = implies-elim
             (instantiate' [SOME T] (map SOME [c,x]) @{thm nz-prod-eq}) cth
      in rth end
   | - => reflexive ct);

local
  val less-iff-diff-less-0 = mk-meta-eq @{thm less-iff-diff-less-0}
  val le-iff-diff-le-0 = mk-meta-eq @{thm le-iff-diff-le-0}
  val eq-iff-diff-eq-0 = mk-meta-eq @{thm eq-iff-diff-eq-0}
in
fun field-isolate-conv phi ctxt vs ct = case term-of ct of
  Const(@{const-name HOL.less},-)$a$b =>
```

```
  let val (ca,cb) = Thm.dest-binop ct
      val T = ctyp-of-term ca
      val th = instantiate' [SOME T] [SOME ca, SOME cb] less-iff-diff-less-0
      val nth = Conv.fconv-rule
        (Conv.arg-conv (Conv.arg1-conv
            (Normalizer.semiring-normalize-ord-conv @{context} (earlier vs)))) th
      val rth = transitive nth (xnormalize-conv ctxt vs (Thm.rhs-of nth))
  in rth end
| Const(@{const-name HOL.less-eq},-)$a$b =>
  let val (ca,cb) = Thm.dest-binop ct
      val T = ctyp-of-term ca
      val th = instantiate' [SOME T] [SOME ca, SOME cb] le-iff-diff-le-0
      val nth = Conv.fconv-rule
        (Conv.arg-conv (Conv.arg1-conv
            (Normalizer.semiring-normalize-ord-conv @{context} (earlier vs)))) th
      val rth = transitive nth (xnormalize-conv ctxt vs (Thm.rhs-of nth))
  in rth end

| Const(op =,-)$a$b =>
  let val (ca,cb) = Thm.dest-binop ct
      val T = ctyp-of-term ca
      val th = instantiate' [SOME T] [SOME ca, SOME cb] eq-iff-diff-eq-0
      val nth = Conv.fconv-rule
        (Conv.arg-conv (Conv.arg1-conv
            (Normalizer.semiring-normalize-ord-conv @{context} (earlier vs)))) th
      val rth = transitive nth (xnormalize-conv ctxt vs (Thm.rhs-of nth))
  in rth end
| @{term Not} $(Const(op =,-)$a$b) => Conv.arg-conv (field-isolate-conv phi ctxt
vs) ct
| - => reflexive ct
end;

fun classfield-whatis phi =
 let
  fun h x t =
   case term-of t of
     Const(op =, -)$y$z => if term-of x aconv y then Ferrante-Rackoff-Data.Eq
                           else Ferrante-Rackoff-Data.Nox
  | @{term Not}$(Const(op =, -)$y$z) => if term-of x aconv y then Ferrante-Rackoff-Data.NEq
                           else Ferrante-Rackoff-Data.Nox
   | Const(@{const-name HOL.less},-)$y$z =>
      if term-of x aconv y then Ferrante-Rackoff-Data.Lt
       else if term-of x aconv z then Ferrante-Rackoff-Data.Gt
       else Ferrante-Rackoff-Data.Nox
   | Const (@{const-name HOL.less-eq},-)$y$z =>
        if term-of x aconv y then Ferrante-Rackoff-Data.Le
        else if term-of x aconv z then Ferrante-Rackoff-Data.Ge
        else Ferrante-Rackoff-Data.Nox
   | - => Ferrante-Rackoff-Data.Nox
```

```
 in h end;
fun class-field-ss phi =
   HOL-basic-ss addsimps ([@{thm linorder-not-less}, @{thm linorder-not-le}])
   addsplits [@{thm abs-split},@{thm split-max}, @{thm split-min}]

in
Ferrante-Rackoff-Data.funs @{thm class-ordered-field-dense-linear-order.ferrack-axiom}
 {isolate-conv = field-isolate-conv, whatis = classfield-whatis, simpset = class-field-ss}
end
⟩⟩
```

end

24 Efficient-Nat: Implementation of natural numbers by target-language integers

theory *Efficient-Nat*
imports *Code-Integer Code-Index*
begin

When generating code for functions on natural numbers, the canonical representation using *0* and *Suc* is unsuitable for computations involving large numbers. The efficiency of the generated code can be improved drastically by implementing natural numbers by target-language integers. To do this, just include this theory.

24.1 Basic arithmetic

Most standard arithmetic functions on natural numbers are implemented using their counterparts on the integers:

code-datatype *number-nat-inst.number-of-nat*

lemma *zero-nat-code* [*code*, *code unfold*]:
  0 = (*Numeral0* :: *nat*)
  by *simp*
lemmas [*code post*] = *zero-nat-code* [*symmetric*]

lemma *one-nat-code* [*code*, *code unfold*]:
  1 = (*Numeral1* :: *nat*)
  by *simp*
lemmas [*code post*] = *one-nat-code* [*symmetric*]

lemma *Suc-code* [*code*]:
  Suc n = *n* + *1*
  by *simp*

lemma *plus-nat-code* [*code*]:
n + m = nat (of-nat n + of-nat m)
by *simp*

lemma *minus-nat-code* [*code*]:
n − m = nat (of-nat n − of-nat m)
by *simp*

lemma *times-nat-code* [*code*]:
*n * m = nat (of-nat n * of-nat m)*
unfolding *of-nat-mult* [*symmetric*] **by** *simp*

Specialized *op div* and *op mod* operations.

definition
divmod-aux :: *nat ⇒ nat ⇒ nat × nat*
where
[*code func del*]: *divmod-aux = divmod*

lemma [*code func*]:
divmod n m = (if m = 0 then (0, n) else divmod-aux n m)
unfolding *divmod-aux-def divmod-div-mod* **by** *simp*

lemma *divmod-aux-code* [*code*]:
divmod-aux n m = (nat (of-nat n div of-nat m), nat (of-nat n mod of-nat m))
unfolding *divmod-aux-def divmod-div-mod zdiv-int* [*symmetric*] *zmod-int* [*symmetric*]
by *simp*

lemma *eq-nat-code* [*code*]:
n = m ⟷ (of-nat n :: int) = of-nat m
by *simp*

lemma *less-eq-nat-code* [*code*]:
n ≤ m ⟷ (of-nat n :: int) ≤ of-nat m
by *simp*

lemma *less-nat-code* [*code*]:
n < m ⟷ (of-nat n :: int) < of-nat m
by *simp*

24.2 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code func*, *code unfold*]:
nat-case = (λf g n. if n = 0 then f else g (n − 1))
by (*auto simp add*: *expand-fun-eq dest*!: *gr0-implies-Suc*)

24.3 Preprocessors

In contrast to *Suc n*, the term *n + 1* is no longer a constructor term. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a recursion equation or in the arguments of an inductive relation in an introduction rule) must be eliminated. This can be accomplished by applying the following transformation rules:

lemma *Suc-if-eq*: $(\bigwedge n.\ f\ (Suc\ n) = h\ n) \Longrightarrow f\ 0 = g \Longrightarrow$
$f\ n = (if\ n = 0\ then\ g\ else\ h\ (n - 1))$
by (*case-tac n*) *simp-all*

lemma *Suc-clause*: $(\bigwedge n.\ P\ n\ (Suc\ n)) \Longrightarrow n \neq 0 \Longrightarrow P\ (n - 1)\ n$
by (*case-tac n*) *simp-all*

The rules above are built into a preprocessor that is plugged into the code generator. Since the preprocessor for introduction rules does not know anything about modes, some of the modes that worked for the canonical representation of natural numbers may no longer work.

24.4 Target language setup

For ML, we map *nat* to target language integers, where we assert that values are always non-negative.

code-type *nat*
(*SML int*)
(*OCaml Big'-int.big'-int*)

types-code
nat (*int*)
attach (*term-of*) ⟪
val term-of-nat = HOLogic.mk-number HOLogic.natT;
⟫
attach (*test*) ⟪
fun gen-nat i =
let val n = random-range 0 i
in (*n*, *fn* () => *term-of-nat n*) *end*;
⟫

For Haskell we define our own *nat* type. The reason is that we have to distinguish type class instances for *nat* and *int*.

code-include *Haskell Nat* ⟪
newtype Nat = Nat Integer deriving (*Show*, *Eq*);

instance Num Nat where {
fromInteger k = Nat (*if k >= 0 then k else 0*);
Nat n + Nat m = Nat (*n + m*);
Nat n - Nat m = fromInteger (*n - m*);
*Nat n * Nat m = Nat* (*n * m*);

```
  abs n = n;
  signum - = 1;
  negate n = error negate Nat;
};

instance Ord Nat where {
  Nat n <= Nat m = n <= m;
  Nat n < Nat m = n < m;
};

instance Real Nat where {
  toRational (Nat n) = toRational n;
};

instance Enum Nat where {
  toEnum k = fromInteger (toEnum k);
  fromEnum (Nat n) = fromEnum n;
};

instance Integral Nat where {
  toInteger (Nat n) = n;
  divMod n m = quotRem n m;
  quotRem (Nat n) (Nat m) = (Nat k, Nat l) where (k, l) = quotRem n m;
};
⟩⟩
```

code-reserved *Haskell Nat*

code-type *nat*
  (*Haskell Nat*)

code-instance *nat* :: *eq*
  (*Haskell* −)

Natural numerals.

lemma [*code inline*, *symmetric*, *code post*]:
  nat (*number-of i*) = *number-nat-inst.number-of-nat i*
  — this interacts as desired with *number-of ?v* = *nat* (*number-of ?v*)
  by (*simp add*: *number-nat-inst.number-of-nat*)

setup ⟨⟨
  fold (*Numeral.add-code* @{*const-name number-nat-inst.number-of-nat*}
    true false) [*SML*, *OCaml*, *Haskell*]
⟩⟩

Since natural numbers are implemented using integers in ML, the coercion function *of-nat* of type *nat* ⇒ *int* is simply implemented by the identity function. For the *nat* function for converting an integer to a natural number, we give a specific implementation using an ML function that returns

its input value, provided that it is non-negative, and otherwise returns *0*.

definition
int :: *nat* ⇒ *int*
where
[*code func del*]: *int* = *of-nat*

lemma *int-code'* [*code func*]:
int (*number-of l*) = (*if neg* (*number-of l* :: *int*) *then 0 else number-of l*)
unfolding *int-nat-number-of* [*folded int-def*] **..**

lemma *nat-code'* [*code func*]:
nat (*number-of l*) = (*if neg* (*number-of l* :: *int*) *then 0 else number-of l*)
by *auto*

lemma *of-nat-int* [*code unfold*]:
of-nat = *int* **by** (*simp add*: *int-def*)
declare *of-nat-int* [*symmetric*, *code post*]

code-const *int*
(*SML* -)
(*OCaml* -)

consts-code
int ((-))
nat (⟨**module**⟩*nat*)
attach ⟪
fun nat i = *if i* < *0 then 0 else i*;
⟫

code-const *nat*
(*SML IntInf.max*/ (/*0*,/ -))
(*OCaml Big'-int.max'-big'-int*/ *Big'-int.zero'-big'-int*)

For Haskell, things are slightly different again.

code-const *int* **and** *nat*
(*Haskell toInteger* **and** *fromInteger*)

Conversion from and to indices.

code-const *index-of-nat*
(*SML IntInf.toInt*)
(*OCaml Big'-int.int'-of'-big'-int*)
(*Haskell toEnum*)

code-const *nat-of-index*
(*SML IntInf.fromInt*)
(*OCaml Big'-int.big'-int'-of'-int*)
(*Haskell fromEnum*)

Using target language arithmetic operations whenever appropriate

code-const *op* + :: *nat* ⇒ *nat* ⇒ *nat*
(*SML IntInf*.+ ((-), (-)))
(*OCaml Big'-int.add'-big'-int*)
(*Haskell* **infixl** *6* +)

code-const *op* ∗ :: *nat* ⇒ *nat* ⇒ *nat*
(*SML IntInf*.∗ ((-), (-)))
(*OCaml Big'-int.mult'-big'-int*)
(*Haskell* **infixl** *7* ∗)

code-const *divmod-aux*
(*SML IntInf.divMod*/ ((-),/ (-)))
(*OCaml Big'-int.quomod'-big'-int*)
(*Haskell divMod*)

code-const *op* = :: *nat* ⇒ *nat* ⇒ *bool*
(*SML* !((- : *IntInf.int*) = -))
(*OCaml Big'-int.eq'-big'-int*)
(*Haskell* **infixl** *4* ==)

code-const *op* ≤ :: *nat* ⇒ *nat* ⇒ *bool*
(*SML IntInf*.<= ((-), (-)))
(*OCaml Big'-int.le'-big'-int*)
(*Haskell* **infix** *4* <=)

code-const *op* < :: *nat* ⇒ *nat* ⇒ *bool*
(*SML IntInf*.< ((-), (-)))
(*OCaml Big'-int.lt'-big'-int*)
(*Haskell* **infix** *4* <)

consts-code
0 (*0*)
Suc ((- +/ *1*))
op + :: *nat* ⇒ *nat* ⇒ *nat* ((- +/ -))
op ∗ :: *nat* ⇒ *nat* ⇒ *nat* ((- ∗/ -))
op ≤ :: *nat* ⇒ *nat* ⇒ *bool* ((- <=/ -))
op < :: *nat* ⇒ *nat* ⇒ *bool* ((- </ -))

Module names

code-modulename *SML*
Nat Integer
Divides Integer
Efficient-Nat Integer

code-modulename *OCaml*
Nat Integer
Divides Integer
Efficient-Nat Integer

code-modulename *Haskell*
Nat Integer
Divides Integer
Efficient-Nat Integer

hide *const int*

end

25 Enum: Finite types as explicit enumerations

theory *Enum*
imports *Main*
begin

25.1 Class *enum*

class *enum* = *itself* +
fixes *enum* :: *'a list*
assumes *UNIV-enum* [*code func*]: *UNIV* = *set enum*
and *enum-distinct*: *distinct enum*
begin

lemma *finite-enum*: *finite* (*UNIV* :: *'a set*)
unfolding *UNIV-enum* **..**

lemma *enum-all*: *set enum* = *UNIV* **unfolding** *UNIV-enum* **..**

lemma *in-enum* [*intro*]: *x* ∈ *set enum*
unfolding *enum-all* **by** *auto*

lemma *enum-eq-I*:
assumes ⋀*x*. *x* ∈ *set xs*
shows *set enum* = *set xs*
proof −
from *assms UNIV-eq-I* **have** *UNIV* = *set xs* **by** *auto*
with *enum-all* **show** *?thesis* **by** *simp*
qed

end

25.2 Equality and order on functions

instantiation *fun* :: (*enum*, *eq*) *eq*
begin

definition
eq-class.eq f g ⟷ (∀ *x* ∈ *set enum*. *f x* = *g x*)

instance by *default*
(*simp-all add*: *eq-fun-def enum-all expand-fun-eq*)

end

lemma *order-fun* [*code func*]:
fixes $f\ g :: 'a{::}enum \Rightarrow 'b{::}order$
shows $f \leq g \longleftrightarrow$ *list-all* $(\lambda x.\ f\ x \leq g\ x)$ *enum*
and $f < g \longleftrightarrow f \leq g \wedge \neg$ *list-all* $(\lambda x.\ f\ x = g\ x)$ *enum*
by (*simp-all add*: *list-all-iff enum-all expand-fun-eq le-fun-def less-fun-def order-less-le*)

25.3 Quantifiers

lemma *all-code* [*code func*]: $(\forall x.\ P\ x) \longleftrightarrow$ *list-all P enum*
by (*simp add*: *list-all-iff enum-all*)

lemma *exists-code* [*code func*]: $(\exists x.\ P\ x) \longleftrightarrow \neg$ *list-all* (*Not o P*) *enum*
by (*simp add*: *list-all-iff enum-all*)

25.4 Default instances

primrec *n-lists* :: *nat* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list list* **where**
n-lists 0 xs = [[]]
| *n-lists* (*Suc n*) *xs* = *concat* (*map* $(\lambda ys.$ *map* $(\lambda y.\ y\ \#\ ys)$ *xs*) (*n-lists n xs*))

lemma *n-lists-Nil* [*simp*]: *n-lists n* [] = (*if n* = *0 then* [[]] *else* [])
by (*induct n*) *simp-all*

lemma *length-n-lists*: *length* (*n-lists n xs*) = *length xs* ^ *n*
by (*induct n*) (*auto simp add*: *length-concat map-compose* [*symmetric*] *o-def listsum-triv*)

lemma *length-n-lists-elem*: *ys* $\in$ *set* (*n-lists n xs*) $\Longrightarrow$ *length ys* = *n*
by (*induct n arbitrary*: *ys*) *auto*

lemma *set-n-lists*: *set* (*n-lists n xs*) = $\{ys.$ *length ys* = *n* $\wedge$ *set ys* $\subseteq$ *set xs*$\}$
proof (*rule set-ext*)
fix *ys* :: $'a$ *list*
show *ys* $\in$ *set* (*n-lists n xs*) $\longleftrightarrow$ *ys* $\in \{ys.$ *length ys* = *n* $\wedge$ *set ys* $\subseteq$ *set xs*$\}$
proof −
have *ys* $\in$ *set* (*n-lists n xs*) $\Longrightarrow$ *length ys* = *n*
by (*induct n arbitrary*: *ys*) *auto*
moreover have $\bigwedge x.$ *ys* $\in$ *set* (*n-lists n xs*) $\Longrightarrow x \in$ *set ys* $\Longrightarrow x \in$ *set xs*
by (*induct n arbitrary*: *ys*) *auto*
moreover have *set ys* $\subseteq$ *set xs* $\Longrightarrow$ *ys* $\in$ *set* (*n-lists* (*length ys*) *xs*)
by (*induct ys*) *auto*
ultimately show *?thesis* **by** *auto*
qed
qed

lemma *distinct-n-lists*:
assumes *distinct xs*
shows *distinct* (*n-lists n xs*)
proof (*rule card-distinct*)
from *assms* **have** *card-length*: *card* (*set xs*) = *length xs* **by** (*rule distinct-card*)
have *card* (*set* (*n-lists n xs*)) = *card* (*set xs*) ^ *n*
proof (*induct n*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc n*)
moreover have *card* (⋃ *ys*∈*set* (*n-lists n xs*). (λ*y*. *y* # *ys*) ‘ *set xs*)
= (∑ *ys*∈*set* (*n-lists n xs*). *card* ((λ*y*. *y* # *ys*) ‘ *set xs*))
by (*rule card-UN-disjoint*) *auto*
moreover have ⋀*ys*. *card* ((λ*y*. *y* # *ys*) ‘ *set xs*) = *card* (*set xs*)
by (*rule card-image*) (*simp add*: *inj-on-def*)
ultimately show *?case* **by** *auto*
qed
also have ... = *length xs* ^ *n* **by** (*simp add*: *card-length*)
finally show *card* (*set* (*n-lists n xs*)) = *length* (*n-lists n xs*)
by (*simp add*: *length-n-lists*)
qed

lemma *map-of-zip-map*:
fixes *f* :: *'a*::*enum* ⇒ *'b*::*enum*
shows *map-of* (*zip xs* (*map f xs*)) = (λ*x*. *if x* ∈ *set xs then Some* (*f x*) *else None*)
by (*induct xs*) (*simp-all add*: *expand-fun-eq*)

lemma *map-of-zip-enum-is-Some*:
assumes *length ys* = *length* (*enum* :: *'a*::*enum list*)
shows ∃ *y*. *map-of* (*zip* (*enum* :: *'a*::*enum list*) *ys*) *x* = *Some y*
proof −
from *assms* **have** *x* ∈ *set* (*enum* :: *'a*::*enum list*) ⟷
(∃ *y*. *map-of* (*zip* (*enum* :: *'a*::*enum list*) *ys*) *x* = *Some y*)
by (*auto intro*!: *map-of-zip-is-Some*)
then show *?thesis* **using** *enum-all* **by** *auto*
qed

lemma *map-of-zip-enum-inject*:
fixes *xs ys* :: *'b*::*enum list*
assumes *length*: *length xs* = *length* (*enum* :: *'a*::*enum list*)
length ys = *length* (*enum* :: *'a*::*enum list*)
and *map-of*: *the* ∘ *map-of* (*zip* (*enum* :: *'a*::*enum list*) *xs*) = *the* ∘ *map-of* (*zip* (*enum* :: *'a*::*enum list*) *ys*)
shows *xs* = *ys*
proof −
have *map-of* (*zip* (*enum* :: *'a list*) *xs*) = *map-of* (*zip* (*enum* :: *'a list*) *ys*)
proof

```
    fix x :: 'a
    from length map-of-zip-enum-is-Some obtain y1 y2
      where map-of (zip (enum :: 'a list) xs) x = Some y1
        and map-of (zip (enum :: 'a list) ys) x = Some y2 by blast
    moreover from map-of have the (map-of (zip (enum :: 'a::enum list) xs) x)
= the (map-of (zip (enum :: 'a::enum list) ys) x)
      by (auto dest: fun-cong)
     ultimately show map-of (zip (enum :: 'a::enum list) xs) x = map-of (zip
(enum :: 'a::enum list) ys) x
      by simp
  qed
  with length enum-distinct show xs = ys by (rule map-of-zip-inject)
qed

instantiation fun :: (enum, enum) enum
begin

definition
  [code func del]: enum = map (λys. the o map-of (zip (enum::'a list) ys)) (n-lists
(length (enum::'a::enum list)) enum)

instance proof
  show UNIV = set (enum :: ('a ⇒ 'b) list)
  proof (rule UNIV-eq-I)
    fix f :: 'a ⇒ 'b
    have f = the ∘ map-of (zip (enum :: 'a::enum list) (map f enum))
      by (auto simp add: map-of-zip-map expand-fun-eq)
    then show f ∈ set enum
      by (auto simp add: enum-fun-def set-n-lists)
  qed
next
  from map-of-zip-enum-inject
  show distinct (enum :: ('a ⇒ 'b) list)
    by (auto intro!: inj-onI simp add: enum-fun-def
      distinct-map distinct-n-lists enum-distinct set-n-lists enum-all)
qed

end

lemma [code func]:
  enum = map (λys. the o map-of (zip (enum::('a::{enum, eq}) list) ys)) (n-lists
(length (enum::'a::{enum, eq} list)) enum)
  unfolding enum-fun-def ..

instantiation unit :: enum
begin

definition
  enum = [()]
```

instance by *default*
(*simp-all add*: *enum-unit-def UNIV-unit*)

end

instantiation *bool* :: *enum*
begin

definition
enum = [*False*, *True*]

instance by *default*
(*simp-all add*: *enum-bool-def UNIV-bool*)

end

primrec *product* :: *'a list* ⇒ *'b list* ⇒ (*'a* × *'b*) *list* **where**
product [] - = []
| *product* (*x*#*xs*) *ys* = *map* (*Pair x*) *ys* @ *product xs ys*

lemma *product-list-set*:
set (*product xs ys*) = *set xs* × *set ys*
by (*induct xs*) *auto*

lemma *distinct-product*:
assumes *distinct xs* **and** *distinct ys*
shows *distinct* (*product xs ys*)
using *assms* **by** (*induct xs*)
(*auto intro*: *inj-onI simp add*: *product-list-set distinct-map*)

instantiation ∗ :: (*enum*, *enum*) *enum*
begin

definition
enum = *product enum enum*

instance by *default*
(*simp-all add*: *enum-prod-def product-list-set distinct-product enum-all enum-distinct*)

end

instantiation + :: (*enum*, *enum*) *enum*
begin

definition
enum = *map Inl enum* @ *map Inr enum*

instance by *default*

(*auto simp add*: *enum-all enum-sum-def*, *case-tac x*, *auto intro*: *inj-onI simp add*: *distinct-map enum-distinct*)

end

primrec *sublists* :: *'a list* ⇒ *'a list list* **where**
sublists [] = [[]]
| *sublists* (*x*#*xs*) = (*let xss* = *sublists xs in map* (*Cons x*) *xss* @ *xss*)

lemma *length-sublists*:
length (*sublists xs*) = *Suc* (*Suc* (*0*::*nat*)) ^ *length xs*
by (*induct xs*) (*simp-all add*: *Let-def*)

lemma *sublists-powset*:
set ‘ *set* (*sublists xs*) = *Pow* (*set xs*)
proof −
have *aux*: ⋀*x A*. *set* ‘ *Cons x* ‘ *A* = *insert x* ‘ *set* ‘ *A*
by (*auto simp add*: *image-def*)
have *set* (*map set* (*sublists xs*)) = *Pow* (*set xs*)
by (*induct xs*)
(*simp-all add*: *aux Let-def Pow-insert Un-commute*)
then show *?thesis* **by** *simp*
qed

lemma *distinct-set-sublists*:
assumes *distinct xs*
shows *distinct* (*map set* (*sublists xs*))
proof (*rule card-distinct*)
have *finite* (*set xs*) **by** *rule*
then have *card* (*Pow* (*set xs*)) = *Suc* (*Suc 0*) ^ *card* (*set xs*) **by** (*rule card-Pow*)
with *assms distinct-card* [*of xs*]
have *card* (*Pow* (*set xs*)) = *Suc* (*Suc 0*) ^ *length xs* **by** *simp*
then show *card* (*set* (*map set* (*sublists xs*))) = *length* (*map set* (*sublists xs*))
by (*simp add*: *sublists-powset length-sublists*)
qed

instantiation *nibble* :: *enum*
begin

definition
enum = [*Nibble0*, *Nibble1*, *Nibble2*, *Nibble3*, *Nibble4*, *Nibble5*, *Nibble6*, *Nibble7*,
Nibble8, *Nibble9*, *NibbleA*, *NibbleB*, *NibbleC*, *NibbleD*, *NibbleE*, *NibbleF*]

instance by *default*
(*simp-all add*: *enum-nibble-def UNIV-nibble*)

end

instantiation *char* :: *enum*

begin

definition
[*code func del*]: *enum* = *map* (*split Char*) (*product enum enum*)

lemma *enum-char* [*code func*]:
enum = [*Char Nibble0 Nibble0*, *Char Nibble0 Nibble1*, *Char Nibble0 Nibble2*,
Char Nibble0 Nibble3, *Char Nibble0 Nibble4*, *Char Nibble0 Nibble5*,
Char Nibble0 Nibble6, *Char Nibble0 Nibble7*, *Char Nibble0 Nibble8*,
Char Nibble0 Nibble9, *Char Nibble0 NibbleA*, *Char Nibble0 NibbleB*,
Char Nibble0 NibbleC, *Char Nibble0 NibbleD*, *Char Nibble0 NibbleE*,
Char Nibble0 NibbleF, *Char Nibble1 Nibble0*, *Char Nibble1 Nibble1*,
Char Nibble1 Nibble2, *Char Nibble1 Nibble3*, *Char Nibble1 Nibble4*,
Char Nibble1 Nibble5, *Char Nibble1 Nibble6*, *Char Nibble1 Nibble7*,
Char Nibble1 Nibble8, *Char Nibble1 Nibble9*, *Char Nibble1 NibbleA*,
Char Nibble1 NibbleB, *Char Nibble1 NibbleC*, *Char Nibble1 NibbleD*,
Char Nibble1 NibbleE, *Char Nibble1 NibbleF*, *CHR '' ''*, *CHR ''!''*,
Char Nibble2 Nibble2, *CHR ''#''*, *CHR ''$''*, *CHR ''%''*, *CHR ''&''*,
Char Nibble2 Nibble7, *CHR ''('*, *CHR '')''*, *CHR ''*''*, *CHR ''+''*, *CHR '',''*,
CHR ''−'', *CHR ''.''*, *CHR ''/''*, *CHR ''0''*, *CHR ''1''*, *CHR ''2''*, *CHR ''3''*,
CHR ''4'', *CHR ''5''*, *CHR ''6''*, *CHR ''7''*, *CHR ''8''*, *CHR ''9''*, *CHR '':''*,
CHR '';'', *CHR ''<''*, *CHR ''=''*, *CHR ''>''*, *CHR ''?''*, *CHR ''@''*, *CHR ''A''*,
CHR ''B'', *CHR ''C''*, *CHR ''D''*, *CHR ''E''*, *CHR ''F''*, *CHR ''G''*, *CHR ''H''*,
CHR ''I'', *CHR ''J''*, *CHR ''K''*, *CHR ''L''*, *CHR ''M''*, *CHR ''N''*, *CHR ''O''*,
CHR ''P'', *CHR ''Q''*, *CHR ''R''*, *CHR ''S''*, *CHR ''T''*, *CHR ''U''*, *CHR ''V''*,
CHR ''W'', *CHR ''X''*, *CHR ''Y''*, *CHR ''Z''*, *CHR ''[''*, *Char Nibble5 NibbleC*,
CHR '']'', *CHR ''^''*, *CHR ''-''*, *Char Nibble6 Nibble0*, *CHR ''a''*, *CHR ''b''*,
CHR ''c'', *CHR ''d''*, *CHR ''e''*, *CHR ''f''*, *CHR ''g''*, *CHR ''h''*, *CHR ''i''*,
CHR ''j'', *CHR ''k''*, *CHR ''l''*, *CHR ''m''*, *CHR ''n''*, *CHR ''o''*, *CHR ''p''*,
CHR ''q'', *CHR ''r''*, *CHR ''s''*, *CHR ''t''*, *CHR ''u''*, *CHR ''v''*, *CHR ''w''*,
CHR ''x'', *CHR ''y''*, *CHR ''z''*, *CHR ''{''*, *CHR ''|''*, *CHR ''}''*, *CHR ''~''*,
Char Nibble7 NibbleF, *Char Nibble8 Nibble0*, *Char Nibble8 Nibble1*,
Char Nibble8 Nibble2, *Char Nibble8 Nibble3*, *Char Nibble8 Nibble4*,
Char Nibble8 Nibble5, *Char Nibble8 Nibble6*, *Char Nibble8 Nibble7*,
Char Nibble8 Nibble8, *Char Nibble8 Nibble9*, *Char Nibble8 NibbleA*,
Char Nibble8 NibbleB, *Char Nibble8 NibbleC*, *Char Nibble8 NibbleD*,
Char Nibble8 NibbleE, *Char Nibble8 NibbleF*, *Char Nibble9 Nibble0*,
Char Nibble9 Nibble1, *Char Nibble9 Nibble2*, *Char Nibble9 Nibble3*,
Char Nibble9 Nibble4, *Char Nibble9 Nibble5*, *Char Nibble9 Nibble6*,
Char Nibble9 Nibble7, *Char Nibble9 Nibble8*, *Char Nibble9 Nibble9*,
Char Nibble9 NibbleA, *Char Nibble9 NibbleB*, *Char Nibble9 NibbleC*,
Char Nibble9 NibbleD, *Char Nibble9 NibbleE*, *Char Nibble9 NibbleF*,
Char NibbleA Nibble0, *Char NibbleA Nibble1*, *Char NibbleA Nibble2*,
Char NibbleA Nibble3, *Char NibbleA Nibble4*, *Char NibbleA Nibble5*,
Char NibbleA Nibble6, *Char NibbleA Nibble7*, *Char NibbleA Nibble8*,
Char NibbleA Nibble9, *Char NibbleA NibbleA*, *Char NibbleA NibbleB*,
Char NibbleA NibbleC, *Char NibbleA NibbleD*, *Char NibbleA NibbleE*,
Char NibbleA NibbleF, *Char NibbleB Nibble0*, *Char NibbleB Nibble1*,
Char NibbleB Nibble2, *Char NibbleB Nibble3*, *Char NibbleB Nibble4*,

Char NibbleB Nibble5, *Char NibbleB Nibble6*, *Char NibbleB Nibble7*,
Char NibbleB Nibble8, *Char NibbleB Nibble9*, *Char NibbleB NibbleA*,
Char NibbleB NibbleB, *Char NibbleB NibbleC*, *Char NibbleB NibbleD*,
Char NibbleB NibbleE, *Char NibbleB NibbleF*, *Char NibbleC Nibble0*,
Char NibbleC Nibble1, *Char NibbleC Nibble2*, *Char NibbleC Nibble3*,
Char NibbleC Nibble4, *Char NibbleC Nibble5*, *Char NibbleC Nibble6*,
Char NibbleC Nibble7, *Char NibbleC Nibble8*, *Char NibbleC Nibble9*,
Char NibbleC NibbleA, *Char NibbleC NibbleB*, *Char NibbleC NibbleC*,
Char NibbleC NibbleD, *Char NibbleC NibbleE*, *Char NibbleC NibbleF*,
Char NibbleD Nibble0, *Char NibbleD Nibble1*, *Char NibbleD Nibble2*,
Char NibbleD Nibble3, *Char NibbleD Nibble4*, *Char NibbleD Nibble5*,
Char NibbleD Nibble6, *Char NibbleD Nibble7*, *Char NibbleD Nibble8*,
Char NibbleD Nibble9, *Char NibbleD NibbleA*, *Char NibbleD NibbleB*,
Char NibbleD NibbleC, *Char NibbleD NibbleD*, *Char NibbleD NibbleE*,
Char NibbleD NibbleF, *Char NibbleE Nibble0*, *Char NibbleE Nibble1*,
Char NibbleE Nibble2, *Char NibbleE Nibble3*, *Char NibbleE Nibble4*,
Char NibbleE Nibble5, *Char NibbleE Nibble6*, *Char NibbleE Nibble7*,
Char NibbleE Nibble8, *Char NibbleE Nibble9*, *Char NibbleE NibbleA*,
Char NibbleE NibbleB, *Char NibbleE NibbleC*, *Char NibbleE NibbleD*,
Char NibbleE NibbleE, *Char NibbleE NibbleF*, *Char NibbleF Nibble0*,
Char NibbleF Nibble1, *Char NibbleF Nibble2*, *Char NibbleF Nibble3*,
Char NibbleF Nibble4, *Char NibbleF Nibble5*, *Char NibbleF Nibble6*,
Char NibbleF Nibble7, *Char NibbleF Nibble8*, *Char NibbleF Nibble9*,
Char NibbleF NibbleA, *Char NibbleF NibbleB*, *Char NibbleF NibbleC*,
Char NibbleF NibbleD, *Char NibbleF NibbleE*, *Char NibbleF NibbleF*]
unfolding *enum-char-def enum-nibble-def* **by** *simp*

instance by *default*
(*auto intro*: *char.exhaust injI simp add*: *enum-char-def product-list-set enum-all full-SetCompr-eq* [*symmetric*]
distinct-map distinct-product enum-distinct)

end

end

26 RType: Reflecting Pure types into HOL

theory *RType*
imports *Main Code-Message Code-Index*
begin

datatype *rtype* = *RType message-string rtype list*

class *rtype* =
fixes *rtype* :: *'a*::{} *itself* ⇒ *rtype*
begin

definition

```
  rtype-of :: 'a ⇒ rtype
where
  [simp]: rtype-of x = rtype TYPE('a)

end

setup ⟪
let
  fun rtype-tr (*-RTYPE*) [ty] =
        Lexicon.const @{const-syntax rtype} $ (Lexicon.const -constrain $ Lexi-
con.const TYPE $
        (Lexicon.const itself $ ty))
    | rtype-tr (*-RTYPE*) ts = raise TERM (rtype-tr, ts);
  fun rtype-tr' show-sorts (*rtype*)
        (Type (fun, [Type (itself, [T]), -])) (Const (@{const-syntax TYPE}, -) ::
ts) =
      Term.list-comb (Lexicon.const -RTYPE $ Syntax.term-of-typ show-sorts T,
ts)
    | rtype-tr' - T ts = raise Match;
in
  Sign.add-syntax-i
   [(-RTYPE, SimpleSyntax.read-typ type => logic, Delimfix (1RTYPE/(1'(-'))))]
  #> Sign.add-trfuns ([], [(-RTYPE, rtype-tr)], [], [])
  #> Sign.add-trfunsT [(@{const-syntax rtype}, rtype-tr')]
end
⟫

ML ⟪
structure RType =
struct

fun mk f (Type (tyco, tys)) =
      @{term RType} $ Message-String.mk tyco
        $ HOLogic.mk-list @{typ rtype} (map (mk f) tys)
  | mk f (TFree v) =
      f v;

fun rtype ty =
  Const (@{const-name rtype}, Term.itselfT ty --> @{typ rtype})
    $ Logic.mk-type ty;

fun add-def tyco thy =
  let
    val sorts = replicate (Sign.arity-number thy tyco) @{sort rtype};
    val vs = Name.names Name.context 'a sorts;
    val ty = Type (tyco, map TFree vs);
    val lhs = Const (@{const-name rtype}, Term.itselfT ty --> @{typ rtype})
      $ Free (T, Term.itselfT ty);
    val rhs = mk (rtype o TFree) ty;
```

```
    val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs));
  in
    thy
    |> TheoryTarget.instantiation ([tyco], vs, @{sort rtype})
    |> `(fn lthy => Syntax.check-term lthy eq)
    |-> (fn eq => Specification.definition (NONE, ((, []), eq)))
    |> snd
    |> Class.prove-instantiation-instance (K (Class.intro-classes-tac []))
    |> LocalTheory.exit
    |> ProofContext.theory-of
  end;

fun perhaps-add-def tyco thy =
  let
    val inst = can (Sorts.mg-domain (Sign.classes-of thy) tyco) @{sort rtype}
  in if inst then thy else add-def tyco thy end;

end;
⟫

setup ⟪
  RType.add-def @{type-name prop}
  #> RType.add-def @{type-name fun}
  #> RType.add-def @{type-name itself}
  #> RType.add-def @{type-name bool}
  #> TypedefPackage.interpretation RType.perhaps-add-def
⟫

lemma [code func]:
  RType tyco1 tys1 = RType tyco2 tys2 ⟷ tyco1 = tyco2
    ∧ list-all2 (op =) tys1 tys2
  by (auto simp add: list-all2-eq [symmetric])

code-type rtype
  (SML Term.typ)

code-const RType
  (SML Term.Type/ (-, -))

code-reserved SML Term

hide (open) const rtype RType

end
```

27 Eval: A simple term evaluation mechanism

theory *Eval*

imports
RType
Code-Index
begin

27.1 Term representation

27.1.1 Terms and class *term-of*

datatype *term* = *dummy-term*

definition
Const :: *message-string* ⇒ *rtype* ⇒ *term*
where
Const - - = dummy-term

definition
App :: *term* ⇒ *term* ⇒ *term*
where
App - - = dummy-term

code-datatype *Const App*

class *term-of* = *rtype* +
fixes *term-of* :: *'a* ⇒ *term*

lemma *term-of-anything*: *term-of x* ≡ *t*
by (*rule eq-reflection*) (*cases term-of x*, *cases t*, *simp*)

ML ⟪
structure Eval =
struct

fun mk-term f g (*Const* (*c*, *ty*)) =
@{*term Const*} $ *Message-String.mk c* $ *g ty*
| *mk-term f g* (*t1* $ *t2*) =
@{*term App*} $ *mk-term f g t1* $ *mk-term f g t2*
| *mk-term f g* (*Free v*) = *f v*
| *mk-term f g* (*Bound i*) = *Bound i*
| *mk-term f g* (*Abs* (*v*, -, *t*)) = *Abs* (*v*, @{*typ term*}, *mk-term f g t*);

fun mk-term-of ty t = *Const* (@{*const-name term-of*}, *ty* −−> @{*typ term*}) $ *t*;

end;
⟫

27.1.2 *term-of* instances

setup ⟪
let

```
  fun add-term-of-def ty vs tyco thy =
    let
      val lhs = Const (@{const-name term-of}, ty --> @{typ term})
        $ Free (x, ty);
      val rhs = @{term undefined :: term};
      val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs));
    in
      thy
      |> TheoryTarget.instantiation ([tyco], vs, @{sort term-of})
      |> `(fn lthy => Syntax.check-term lthy eq)
      |-> (fn eq => Specification.definition (NONE, ((, []), eq)))
      |> snd
      |> Class.prove-instantiation-instance (K (Class.intro-classes-tac []))
      |> LocalTheory.exit
      |> ProofContext.theory-of
    end;
  fun interpretator (tyco, (raw-vs, -)) thy =
    let
       val has-inst = can (Sorts.mg-domain (Sign.classes-of thy) tyco) @{sort
term-of};
      val constrain-sort =
        curry (Sorts.inter-sort (Sign.classes-of thy)) @{sort term-of};
      val vs = (map o apsnd) constrain-sort raw-vs;
      val ty = Type (tyco, map TFree vs);
    in
      thy
      |> RType.perhaps-add-def tyco
      |> not has-inst ? add-term-of-def ty vs tyco
    end;
in
  Code.type-interpretation interpretator
end
⟩⟩

setup ⟨⟨
let
  fun mk-term-of-eq ty vs tyco (c, tys) =
    let
      val t = list-comb (Const (c, tys ---> ty),
        map Free (Name.names Name.context a tys));
    in (map-aterms (fn Free (v, ty) => Var ((v, 0), ty) | t => t) t, Eval.mk-term
      (fn (v, ty) => Eval.mk-term-of ty (Var ((v, 0), ty)))
      (RType.mk (fn (v, sort) => RType.rtype (TFree (v, sort)))) t)
    end;
  fun prove-term-of-eq ty eq thy =
    let
      val cty = Thm.ctyp-of thy ty;
      val (arg, rhs) = pairself (Thm.cterm-of thy) eq;
      val thm = @{thm term-of-anything}
```

```
      |> Drule.instantiate' [SOME cty] [SOME arg, SOME rhs]
      |> Thm.varifyT;
  in
    thy
    |> Code.add-func thm
  end;
fun interpretator (tyco, (raw-vs, raw-cs)) thy =
  let
    val constrain-sort =
      curry (Sorts.inter-sort (Sign.classes-of thy)) @{sort term-of};
    val vs = (map o apsnd) constrain-sort raw-vs;
    val cs = (map o apsnd o map o map-atyps)
      (fn TFree (v, sort) => TFree (v, constrain-sort sort)) raw-cs;
    val ty = Type (tyco, map TFree vs);
    val eqs = map (mk-term-of-eq ty vs tyco) cs;
    val const = AxClass.param-of-inst thy (@{const-name term-of}, tyco);
  in
    thy
    |> Code.del-funcs const
    |> fold (prove-term-of-eq ty) eqs
  end;
in
  Code.type-interpretation interpretator
end
⟫
```

27.1.3 Code generator setup

lemmas [*code func del*] = *term.recs term.cases term.size*
lemma [*code func*, *code func del*]: (*t1*::*term*) = *t2* ⟷ *t1* = *t2* **..**

lemma [*code func*, *code func del*]: (*term-of* :: *rtype* ⇒ *term*) = *term-of* **..**
lemma [*code func*, *code func del*]: (*term-of* :: *term* ⇒ *term*) = *term-of* **..**
lemma [*code func*, *code func del*]: (*term-of* :: *index* ⇒ *term*) = *term-of* **..**
lemma [*code func*, *code func del*]: (*term-of* :: *message-string* ⇒ *term*) = *term-of* **..**

code-type *term*
(*SML Term.term*)

code-const *Const* **and** *App*
(*SML Term.Const*/ (-, -) **and** *Term.*$/ (-, -))

code-const *term-of* :: *index* ⇒ *term*
(*SML HOLogic.mk'-number*/ *HOLogic.indexT*)

code-const *term-of* :: *message-string* ⇒ *term*
(*SML Message'-String.mk*)

27.1.4 Syntax

print-translation ⟪
let
  val term = Const (<TERM>, dummyT);
  fun tr1' [-, -] = term;
  fun tr2' [] = term;
in
  [(@{const-syntax Const}, tr1'),
    (@{const-syntax App}, tr1'),
    (@{const-syntax dummy-term}, tr2')]
end
⟫
setup ⟪
  Sign.declare-const [] (rterm-of, @{typ 'a ⇒ 'b}, NoSyn)
  #> snd
⟫

notation (**output**)
  rterm-of (≪-≫)

locale (**open**) *rterm-syntax =*
  fixes *rterm-of-syntax :: 'a ⇒ 'b (≪-≫)*

parse-translation ⟪
let
  fun rterm-of-tr [t] = Lexicon.const @{const-name rterm-of} $ t
    | rterm-of-tr ts = raise TERM (rterm-of-tr, ts);
in
  [(Syntax.fixedN ^ rterm-of-syntax, rterm-of-tr)]
end
⟫

setup ⟪
let
  val subst-rterm-of = Eval.mk-term
    (fn (v, -) => error (illegal free variable in term quotation: ^ quote v))
    (RType.mk (fn (v, sort) => RType.rtype (TFree (v, sort))));
  fun subst-rterm-of' (Const (@{const-name rterm-of}, -), [t]) = subst-rterm-of t
    | subst-rterm-of' (Const (@{const-name rterm-of}, -), -) =
        error (illegal number of arguments for ^ quote @{const-name rterm-of})
    | subst-rterm-of' (t, ts) = list-comb (t, map (subst-rterm-of' o strip-comb) ts);
  fun subst-rterm-of'' t =
    let
      val t' = subst-rterm-of' (strip-comb t);
    in if t aconv t'
      then NONE
      else SOME t'
    end;
  fun check-rterm-of ts ctxt =

```
    let
      val ts' = map subst-rterm-of'' ts;
    in if exists is-some ts'
      then SOME (map2 the-default ts ts', ctxt)
      else NONE
    end;
in
  Context.theory-map (Syntax.add-term-check 0 rterm-of check-rterm-of)
end;
⟩⟩

hide const dummy-term
hide (open) const Const App
hide (open) const term-of
```

27.2 Evaluation setup

```
ML ⟨⟨
signature EVAL =
sig
  val mk-term: ((string * typ) -> term) -> (typ -> term) -> term -> term
  val eval-ref: (unit -> term) option ref
  val eval-term: theory -> term -> term
  val evaluate: Proof.context -> term -> unit
  val evaluate': string -> Proof.context -> term -> unit
  val evaluate-cmd: string option -> string -> Toplevel.state -> unit
end;

structure Eval : EVAL =
struct

open Eval;

val eval-ref = ref (NONE : (unit -> term) option);

fun eval-term thy t =
  t
  |> Eval.mk-term-of (fastype-of t)
  |> (fn t => CodePackage.eval-term (Eval.eval-ref, eval-ref) thy t [])
  |> Code.postprocess-term thy;

val evaluators = [
  (code, eval-term),
  (SML, Codegen.eval-term),
  (normal-form, Nbe.norm-term)
];

fun gen-evaluate evaluators ctxt t =
  let
```

```
    val thy = ProofContext.theory-of ctxt;
    val (evls, evl) = split-last evaluators;
    val t' = case get-first (fn f => try (f thy) t) evls
     of SOME t' => t'
      | NONE => evl thy t;
    val ty' = Term.type-of t';
    val p = Pretty.block [Pretty.quote (Syntax.pretty-term ctxt t'),
      Pretty.fbrk, Pretty.str ::, Pretty.brk 1,
      Pretty.quote (Syntax.pretty-typ ctxt ty')];
  in Pretty.writeln p end;

val evaluate = gen-evaluate (map snd evaluators);

fun evaluate' name = gen-evaluate
  [(the o AList.lookup (op =) evaluators) name];

fun evaluate-cmd some-name raw-t state =
  let
    val ctxt = Toplevel.context-of state;
    val t = Syntax.read-term ctxt raw-t;
  in case some-name
   of NONE => evaluate ctxt t
    | SOME name => evaluate' name ctxt t
  end;

end;
⟩⟩

ML ⟨⟨
  OuterSyntax.improper-command value read, evaluate and print term OuterKeyword.diag
    (Scan.option (OuterParse.$$$ ( |-- OuterParse.name --| OuterParse.$$$ ))
    -- OuterParse.term
      >> (fn (some-name, t) => Toplevel.no-timing o Toplevel.keep
          (Eval.evaluate-cmd some-name t)));
⟩⟩

end
```

28 Eval-Witness: Evaluation Oracle with ML witnesses

theory *Eval-Witness*
imports *List*
begin

We provide an oracle method similar to "eval", but with the possibility to provide ML values as witnesses for existential statements.

Our oracle can prove statements of the form $\exists\, x.\ P\ x$ where P is an executable predicate that can be compiled to ML. The oracle generates code for P and applies it to a user-specified ML value. If the evaluation returns true, this is effectively a proof of $\exists\, x.\ P\ x$.

However, this is only sound if for every ML value of the given type there exists a corresponding HOL value, which could be used in an explicit proof. Unfortunately this is not true for function types, since ML functions are not equivalent to the pure HOL functions. Thus, the oracle can only be used on first-order types.

We define a type class to mark types that can be safely used with the oracle.

class *ml-equiv* = *type*

Instances of *ml-equiv* should only be declared for those types, where the universe of ML values coincides with the HOL values.

Since this is essentially a statement about ML, there is no logical characterization.

instance *nat* :: *ml-equiv* **..**
instance *bool* :: *ml-equiv* **..**
instance *list* :: (*ml-equiv*) *ml-equiv* **..**

oracle *eval-witness-oracle* (*term* ∗ *string list*) = ⟪ *fn thy* => *fn* (*goal*, *ws*) =>
let
  fun check-type T =
    if Sorts.of-sort (*Sign.classes-of thy*) (*T*, [*Eval-Witness.ml-equiv*])
    then T
    else error (*Type* ^ *quote* (*Syntax.string-of-typ-global thy T*) ^ *not allowed for ML witnesses*)

  fun dest-exs 0 *t* = *t*
    | *dest-exs n* (*Const* (*Ex*, -) $ *Abs* (*v*,*T*,*b*)) =
      Abs (*v*, *check-type T*, *dest-exs* (*n* − 1) *b*)
    | *dest-exs* - - = *sys-error dest-exs*;
  val t = *dest-exs* (*length ws*) (*HOLogic.dest-Trueprop goal*);
in
  if CodePackage.satisfies thy t ws
  then goal
  else HOLogic.Trueprop $ *HOLogic.true-const* (∗*dummy*∗)
end
⟫

method-setup *eval-witness* = ⟪
let

fun eval-tac ws thy =
  SUBGOAL (*fn* (*t*, *i*) => *rtac* (*eval-witness-oracle thy* (*t*, *ws*)) *i*)

```
in
  Method.simple-args (Scan.repeat Args.name) (fn ws => fn ctxt =>
    Method.SIMPLE-METHOD' (eval-tac ws (ProofContext.theory-of ctxt)))
end
⟩ Evaluation with ML witnesses
```

28.1 Toy Examples

Note that we must use the generated data structure for the naturals, since ML integers are different.

Since polymorphism is not allowed, we must specify the type explicitly:

```
lemma ∃ l. length (l::bool list) = 3
apply (eval-witness [true,true,true])
done
```

Multiple witnesses

```
lemma ∃ k l. length (k::bool list) = length (l::bool list)
apply (eval-witness [] [])
done
```

28.2 Discussion

28.2.1 Conflicts

This theory conflicts with EfficientNat, since the *ml-equiv* instance for natural numbers is not valid when they are mapped to ML integers. With that theory loaded, we could use our oracle to prove $\exists\, n.\ n < (0::'a)$ by providing $^{\sim}1$ as a witness.

This shows that *ml-equiv* declarations have to be used with care, taking the configuration of the code generator into account.

28.2.2 Haskell

If we were able to run generated Haskell code, the situation would be much nicer, since Haskell functions are pure and could be used as witnesses for HOL functions: Although Haskell functions are partial, we know that if the evaluation terminates, they are "sufficiently defined" and could be completed arbitrarily to a total (HOL) function.

This would allow us to provide access to very efficient data structures via lookup functions coded in Haskell and provided to HOL as witnesses.

end

29 Executable-Set: Implementation of finite sets by lists

theory *Executable-Set*
imports *List*
begin

29.1 Definitional rewrites

lemma [*code target*: *Set*]:
  $A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
  by *blast*

declare *subset-eq* [*code*]

lemma [*code*]:
  $a \in A \longleftrightarrow (\exists\, x{\in}A.\ x = a)$
  unfolding *bex-triv-one-point1* **..**

definition
  filter-set :: $('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where**
  filter-set $P\ xs = \{x{\in}xs.\ P\ x\}$

29.2 Operations on lists

29.2.1 Basic definitions

definition
  flip :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$ **where**
  flip $f\ a\ b = f\ b\ a$

definition
  member :: $'a\ list \Rightarrow 'a \Rightarrow bool$ **where**
  member $xs\ x \longleftrightarrow x \in set\ xs$

definition
  insertl :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
  insertl $x\ xs = (if\ member\ xs\ x\ then\ xs\ else\ x\#xs)$

lemma [*code target*: *List*]: *member* $[]\ y \longleftrightarrow False$
  and [*code target*: *List*]: *member* $(x\#xs)\ y \longleftrightarrow y = x \vee member\ xs\ y$
  unfolding *member-def* **by** (*induct xs*) *simp-all*

fun
  drop-first :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
  drop-first $f\ [] = []$
| *drop-first* $f\ (x\#xs) = (if\ f\ x\ then\ xs\ else\ x\ \#\ drop\text{-}first\ f\ xs)$
declare *drop-first.simps* [*code del*]
declare *drop-first.simps* [*code target*: *List*]

declare *remove1.simps* [*code del*]
lemma [*code target*: *List*]:
  remove1 x xs = (*if member xs x then drop-first* (λ*y*. *y* = *x*) *xs else xs*)
proof (*cases member xs x*)
  case *False* **thus** *?thesis* **unfolding** *member-def* **by** (*induct xs*) *auto*
next
  case *True*
  have *remove1 x xs* = *drop-first* (λ*y*. *y* = *x*) *xs* **by** (*induct xs*) *simp-all*
  with *True* **show** *?thesis* **by** *simp*
qed

lemma *member-nil* [*simp*]:
  member [] = (λ*x*. *False*)
proof (*rule ext*)
  fix *x*
  show *member* [] *x* = *False* **unfolding** *member-def* **by** *simp*
qed

lemma *member-insertl* [*simp*]:
  x ∈ *set* (*insertl x xs*)
  unfolding *insertl-def member-def mem-iff* **by** *simp*

lemma *insertl-member* [*simp*]:
  fixes *xs x*
  assumes *member*: *member xs x*
  shows *insertl x xs* = *xs*
  using *member* **unfolding** *insertl-def* **by** *simp*

lemma *insertl-not-member* [*simp*]:
  fixes *xs x*
  assumes *member*: ¬ (*member xs x*)
  shows *insertl x xs* = *x* # *xs*
  using *member* **unfolding** *insertl-def* **by** *simp*

lemma *foldr-remove1-empty* [*simp*]:
  foldr remove1 xs [] = []
  by (*induct xs*) *simp-all*

29.2.2 Derived definitions

function *unionl* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
  unionl [] *ys* = *ys*
| *unionl xs ys* = *foldr insertl xs ys*
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *unionl-eq* = *unionl.simps*(*2*)

function *intersect* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
intersect [] *ys* = []
| *intersect xs* [] = []
| *intersect xs ys* = *filter* (*member xs*) *ys*
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *intersect-eq* = *intersect.simps*(*3*)

function *subtract* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
subtract [] *ys* = *ys*
| *subtract xs* [] = []
| *subtract xs ys* = *foldr remove1 xs ys*
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *subtract-eq* = *subtract.simps*(*3*)

function *map-distinct* :: (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list*
where
map-distinct f [] = []
| *map-distinct f xs* = *foldr* (*insertl o f*) *xs* []
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *map-distinct-eq* = *map-distinct.simps*(*2*)

function *unions* :: *'a list list* ⇒ *'a list*
where
unions [] = []
| *unions xs* = *foldr unionl xs* []
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *unions-eq* = *unions.simps*(*2*)

consts *intersects* :: *'a list list* ⇒ *'a list*
primrec
intersects (*x*#*xs*) = *foldr intersect xs x*

definition
map-union :: *'a list* ⇒ (*'a* ⇒ *'b list*) ⇒ *'b list* **where**
map-union xs f = *unions* (*map f xs*)

definition
map-inter :: *'a list* ⇒ (*'a* ⇒ *'b list*) ⇒ *'b list* **where**
map-inter xs f = *intersects* (*map f xs*)

29.3 Isomorphism proofs

lemma *iso-member*:
member xs x $\longleftrightarrow$ *x* $\in$ *set xs*
unfolding *member-def mem-iff* **..**

lemma *iso-insert*:
set (*insertl x xs*) = *insert x* (*set xs*)
unfolding *insertl-def iso-member* **by** (*simp add*: *Set.insert-absorb*)

lemma *iso-remove1*:
assumes *distnct*: *distinct xs*
shows *set* (*remove1 x xs*) = *set xs* − {*x*}
using *distnct set-remove1-eq* **by** *auto*

lemma *iso-union*:
set (*unionl xs ys*) = *set xs* ∪ *set ys*
unfolding *unionl-eq*
by (*induct xs arbitrary*: *ys*) (*simp-all add*: *iso-insert*)

lemma *iso-intersect*:
set (*intersect xs ys*) = *set xs* ∩ *set ys*
unfolding *intersect-eq Int-def* **by** (*simp add*: *Int-def iso-member*) *auto*

definition
subtract′ :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
subtract′ = *flip subtract*

lemma *iso-subtract*:
fixes *ys*
assumes *distnct*: *distinct ys*
shows *set* (*subtract′ ys xs*) = *set ys* − *set xs*
and *distinct* (*subtract′ ys xs*)
unfolding *subtract′-def flip-def subtract-eq*
using *distnct* **by** (*induct xs arbitrary*: *ys*) *auto*

lemma *iso-map-distinct*:
set (*map-distinct f xs*) = *image f* (*set xs*)
unfolding *map-distinct-eq* **by** (*induct xs*) (*simp-all add*: *iso-insert*)

lemma *iso-unions*:
set (*unions xss*) = $\bigcup$ *set* (*map set xss*)
unfolding *unions-eq*
proof (*induct xss*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons xs xss*) **thus** *?case* **by** (*induct xs*) (*simp-all add*: *iso-insert*)
qed

lemma *iso-intersects*:

set (*intersects* (*xs*#*xss*)) = ⋂ *set* (*map set* (*xs*#*xss*))
by (*induct xss*) (*simp-all add*: *Int-def iso-member*, *auto*)

lemma *iso-UNION*:
set (*map-union xs f*) = *UNION* (*set xs*) (*set o f*)
unfolding *map-union-def iso-unions* **by** *simp*

lemma *iso-INTER*:
set (*map-inter* (*x*#*xs*) *f*) = *INTER* (*set* (*x*#*xs*)) (*set o f*)
unfolding *map-inter-def iso-intersects* **by** (*induct xs*) (*simp-all add*: *iso-member*, *auto*)

definition
Blall :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Blall = *flip list-all*
definition
Blex :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Blex = *flip list-ex*

lemma *iso-Ball*:
Blall xs f = *Ball* (*set xs*) *f*
unfolding *Blall-def flip-def* **by** (*induct xs*) *simp-all*

lemma *iso-Bex*:
Blex xs f = *Bex* (*set xs*) *f*
unfolding *Blex-def flip-def* **by** (*induct xs*) *simp-all*

lemma *iso-filter*:
set (*filter P xs*) = *filter-set P* (*set xs*)
unfolding *filter-set-def* **by** (*induct xs*) *auto*

29.4 code generator setup

ML ⟪
nonfix inter;
nonfix union;
nonfix subset;
⟫

29.4.1 const serializations

consts-code
{} ({∗[]∗})
insert ({∗*insertl*∗})
op ∪ ({∗*unionl*∗})
op ∩ ({∗*intersect*∗})
op − :: *'a set* ⇒ *'a set* ⇒ *'a set* ({∗ *flip subtract* ∗})
image ({∗*map-distinct*∗})
Union ({∗*unions*∗})
Inter ({∗*intersects*∗})

UNION ({∗*map-union*∗})
INTER ({∗*map-inter*∗})
Ball ({∗*Ball*∗})
Bex ({∗*Bex*∗})
filter-set ({∗*filter*∗})

end

30 FuncSet: Pi and Function Sets

theory *FuncSet*
imports *Main*
begin

definition
Pi :: [*'a set*, *'a* => *'b set*] => (*'a* => *'b*) *set* **where**
Pi A B = {*f*. ∀ *x*. *x* ∈ *A* −−> *f x* ∈ *B x*}

definition
extensional :: *'a set* => (*'a* => *'b*) *set* **where**
extensional A = {*f*. ∀ *x*. *x*~:*A* −−> *f x* = *arbitrary*}

definition
restrict :: [*'a* => *'b*, *'a set*] => (*'a* => *'b*) **where**
restrict f A = (%*x*. *if x* ∈ *A then f x else arbitrary*)

abbreviation
funcset :: [*'a set*, *'b set*] => (*'a* => *'b*) *set*
(**infixr** −> *60*) **where**
A −> *B* == *Pi A* (%-. *B*)

notation (*xsymbols*)
funcset (**infixr** → *60*)

syntax
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a* => *'b*) *set* ((*3PI* -:-./ -) *10*)
-lam :: [*pttrn*, *'a set*, *'a* => *'b*] => (*'a*=>*'b*) ((*3*%-:-./ -) [*0*,*0*,*3*] *3*)

syntax (*xsymbols*)
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a* => *'b*) *set* ((*3*Π -∈-./ -) *10*)
-lam :: [*pttrn*, *'a set*, *'a* => *'b*] => (*'a*=>*'b*) ((*3*λ-∈-./ -) [*0*,*0*,*3*] *3*)

syntax (*HTML* **output**)
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a* => *'b*) *set* ((*3*Π -∈-./ -) *10*)
-lam :: [*pttrn*, *'a set*, *'a* => *'b*] => (*'a*=>*'b*) ((*3*λ-∈-./ -) [*0*,*0*,*3*] *3*)

translations
PI x:*A*. *B* == *CONST Pi A* (%*x*. *B*)

%x:A. f == CONST restrict (%x. f) A

definition
compose :: ['a set, 'b => 'c, 'a => 'b] => ('a => 'c) **where**
compose A g f = (λx∈A. *g* (*f x*))

30.1 Basic Properties of *Pi*

lemma *Pi-I*: (!!x. x ∈ A ==> f x ∈ B x) ==> f ∈ Pi A B
by (*simp add*: *Pi-def*)

lemma *funcsetI*: (!!x. x ∈ A ==> f x ∈ B) ==> f ∈ A -> B
by (*simp add*: *Pi-def*)

lemma *Pi-mem*: [|f: Pi A B; x ∈ A|] ==> f x ∈ B x
by (*simp add*: *Pi-def*)

lemma *funcset-mem*: [|f ∈ A -> B; x ∈ A|] ==> f x ∈ B
by (*simp add*: *Pi-def*)

lemma *funcset-image*: f ∈ A→B ==> f ‘ A ⊆ B
by (*auto simp add*: *Pi-def*)

lemma *Pi-eq-empty*: ((PI x: A. B x) = {}) = (∃x∈A. B(x) = {})
apply (*simp add*: *Pi-def*, *auto*)

Converse direction requires Axiom of Choice to exhibit a function picking an element from each non-empty *B x*

apply (*drule-tac x* = %u. *SOME y*. y ∈ B u **in** *spec*, *auto*)
apply (*cut-tac P*= %y. y ∈ B x **in** *some-eq-ex*, *auto*)
done

lemma *Pi-empty* [*simp*]: *Pi* {} *B* = *UNIV*
by (*simp add*: *Pi-def*)

lemma *Pi-UNIV* [*simp*]: *A* -> *UNIV* = *UNIV*
by (*simp add*: *Pi-def*)

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: (!!x. x ∈ A ==> B x <= C x) ==> *Pi A B* <= *Pi A C*
by (*simp add*: *Pi-def*, *blast*)

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: A' <= A ==> *Pi A B* <= *Pi A' B*
by (*simp add*: *Pi-def*, *blast*)

30.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*:
[| f ∈ A -> B; g ∈ B -> C |]==> *compose A g f* ∈ A -> C

by (*simp add*: *Pi-def compose-def restrict-def*)

lemma *compose-assoc*:
[| $f \in A \to B$; $g \in B \to C$; $h \in C \to D$ |]
==> *compose A h* (*compose A g f*) = *compose A* (*compose B h g*) *f*
by (*simp add*: *expand-fun-eq Pi-def compose-def restrict-def*)

lemma *compose-eq*: $x \in A$ ==> *compose A g f x* = $g(f(x))$
by (*simp add*: *compose-def restrict-def*)

lemma *surj-compose*: [| *f* ' *A* = *B*; *g* ' *B* = *C* |] ==> *compose A g f* ' *A* = *C*
by (*auto simp add*: *image-def compose-eq*)

30.3 Bounded Abstraction: *restrict*

lemma *restrict-in-funcset*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B*) ==> (λ*x*∈*A*. *f x*) ∈ *A* -> *B*
by (*simp add*: *Pi-def restrict-def*)

lemma *restrictI*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B x*) ==> (λ*x*∈*A*. *f x*) ∈ *Pi A B*
by (*simp add*: *Pi-def restrict-def*)

lemma *restrict-apply* [*simp*]:
(λ*y*∈*A*. *f y*) *x* = (*if x* ∈ *A then f x else arbitrary*)
by (*simp add*: *restrict-def*)

lemma *restrict-ext*:
(!!*x*. *x* ∈ *A* ==> *f x* = *g x*) ==> (λ*x*∈*A*. *f x*) = (λ*x*∈*A*. *g x*)
by (*simp add*: *expand-fun-eq Pi-def Pi-def restrict-def*)

lemma *inj-on-restrict-eq* [*simp*]: *inj-on* (*restrict f A*) *A* = *inj-on f A*
by (*simp add*: *inj-on-def restrict-def*)

lemma *Id-compose*:
[|*f* ∈ *A* -> *B*; *f* ∈ *extensional A*|] ==> *compose A* (λ*y*∈*B*. *y*) *f* = *f*
by (*auto simp add*: *expand-fun-eq compose-def extensional-def Pi-def*)

lemma *compose-Id*:
[|*g* ∈ *A* -> *B*; *g* ∈ *extensional A*|] ==> *compose A g* (λ*x*∈*A*. *x*) = *g*
by (*auto simp add*: *expand-fun-eq compose-def extensional-def Pi-def*)

lemma *image-restrict-eq* [*simp*]: (*restrict f A*) ' *A* = *f* ' *A*
by (*auto simp add*: *restrict-def*)

30.4 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-betw-imp-funcset*: *bij-betw f A B* ⟹ $f \in A \to B$

by (*auto simp add*: *bij-betw-def inj-on-Inv Pi-def*)

lemma *inj-on-compose*:
[| *bij-betw f A B*; *inj-on g B* |] ==> *inj-on* (*compose A g f*) *A*
by (*auto simp add*: *bij-betw-def inj-on-def compose-eq*)

lemma *bij-betw-compose*:
[| *bij-betw f A B*; *bij-betw g B C* |] ==> *bij-betw* (*compose A g f*) *A C*
apply (*simp add*: *bij-betw-def compose-eq inj-on-compose*)
apply (*auto simp add*: *compose-def image-def*)
done

lemma *bij-betw-restrict-eq* [*simp*]:
bij-betw (*restrict f A*) *A B* = *bij-betw f A B*
by (*simp add*: *bij-betw-def*)

30.5 Extensionality

lemma *extensional-arb*: [|*f* ∈ *extensional A*; *x*∉ *A*|] ==> *f x* = *arbitrary*
by (*simp add*: *extensional-def*)

lemma *restrict-extensional* [*simp*]: *restrict f A* ∈ *extensional A*
by (*simp add*: *restrict-def extensional-def*)

lemma *compose-extensional* [*simp*]: *compose A f g* ∈ *extensional A*
by (*simp add*: *compose-def*)

lemma *extensionalityI*:
[| *f* ∈ *extensional A*; *g* ∈ *extensional A*;
!!*x*. *x*∈*A* ==> *f x* = *g x* |] ==> *f* = *g*
by (*force simp add*: *expand-fun-eq extensional-def*)

lemma *Inv-funcset*: *f* ‘ *A* = *B* ==> (λ*x*∈*B*. *Inv A f x*) : *B* -> *A*
by (*unfold Inv-def*) (*fast intro*: *restrict-in-funcset someI2*)

lemma *compose-Inv-id*:
bij-betw f A B ==> *compose A* (λ*y*∈*B*. *Inv A f y*) *f* = (λ*x*∈*A*. *x*)
apply (*simp add*: *bij-betw-def compose-def*)
apply (*rule restrict-ext*, *auto*)
apply (*erule subst*)
apply (*simp add*: *Inv-f-f*)
done

lemma *compose-id-Inv*:
f ‘ *A* = *B* ==> *compose B f* (λ*y*∈*B*. *Inv A f y*) = (λ*x*∈*B*. *x*)
apply (*simp add*: *compose-def*)
apply (*rule restrict-ext*)
apply (*simp add*: *f-Inv-f*)
done

30.6 Cardinality

lemma *card-inj*: [|$f \in A \to B$; *inj-on f A*; *finite B*|] ==> $card(A) \leq card(B)$
apply (*rule card-inj-on-le*)
apply (*auto simp add*: *Pi-def*)
done

lemma *card-bij*:
[|$f \in A \to B$; *inj-on f A*;
$g \in B \to A$; *inj-on g B*; *finite A*; *finite B*|] ==> $card(A) = card(B)$
by (*blast intro*: *card-inj order-antisym*)

declare *FuncSet.Pi-I* [*skolem*]
declare *FuncSet.Pi-mono* [*skolem*]
declare *FuncSet.extensionalityI* [*skolem*]
declare *FuncSet.funcsetI* [*skolem*]
declare *FuncSet.restrictI* [*skolem*]
declare *FuncSet.restrict-in-funcset* [*skolem*]

end

31 Heap: A polymorphic heap based on cantor encodings

theory *Heap*
imports *Main Countable RType*
begin

31.1 Representable types

The type class of representable types

class *heap* = *rtype* + *countable*

Instances for common HOL types

instance *nat* :: *heap* **..**

instance $*$ :: (*heap*, *heap*) *heap* **..**

instance + :: (*heap*, *heap*) *heap* **..**

instance *list* :: (*heap*) *heap* **..**

instance *option* :: (*heap*) *heap* **..**

instance *int* :: *heap* **..**

instance *message-string* :: *countable*
by (*rule countable-classI* [*of message-string-case to-nat*])
(*auto split*: *message-string.splits*)

instance *message-string* :: *heap* **..**

Reflected types themselves are heap-representable

instantiation *rtype* :: *countable*
begin

lemma *list-size-size-append*:
list-size size (*xs* @ *ys*) = *list-size size xs* + *list-size size ys*
by (*induct xs*, *auto*)

lemma *rtype-size*: *t* = *RType.RType c ts* ⟹ *t'* ∈ *set ts* ⟹ *size t'* < *size t*
by (*frule split-list*) (*auto simp add*: *list-size-size-append*)

function *to-nat-rtype* :: *rtype* ⇒ *nat* **where**
to-nat-rtype (*RType.RType c ts*) = *to-nat* (*to-nat c*, *to-nat* (*map to-nat-rtype ts*))
by *pat-completeness auto*

termination by (*relation measure* (λ*x*. *size x*))
(*simp*, *simp only*: *in-measure rtype-size*)

instance
proof (*rule countable-classI*)
fix *t t'* :: *rtype* **and** *ts*
have (∀ *t'*. *to-nat-rtype t* = *to-nat-rtype t'* ⟶ *t* = *t'*)
∧ (∀ *ts'*. *map to-nat-rtype ts* = *map to-nat-rtype ts'* ⟶ *ts* = *ts'*)
proof (*induct rule*: *rtype.induct*)
case (*RType c ts*) **show** *?case*
proof (*rule allI*, *rule impI*)
fix *t'*
assume *hyp*: *to-nat-rtype* (*rtype.RType c ts*) = *to-nat-rtype t'*
then obtain *c' ts'* **where** *t'*: *t'* = (*rtype.RType c' ts'*)
by (*cases t'*) *auto*
with *RType hyp* **have** *c* = *c'* **and** *ts* = *ts'* **by** *simp-all*
with *t'* **show** *rtype.RType c ts* = *t'* **by** *simp*
qed
next
case *Nil-rtype* **then show** *?case* **by** *simp*
next
case (*Cons-rtype t ts*) **then show** *?case* **by** *auto*
qed
then have *to-nat-rtype t* = *to-nat-rtype t'* ⟹ *t* = *t'* **by** *auto*
moreover assume *to-nat-rtype t* = *to-nat-rtype t'*
ultimately show *t* = *t'* **by** *simp*

qed

end

instance *rtype* :: *heap* **..**

31.2 A polymorphic heap with dynamic arrays and references

types *addr* = *nat* — untyped heap references

datatype *'a array* = *Array addr*
datatype *'a ref* = *Ref addr* — note the phantom type 'a

primrec *addr-of-array* :: *'a array* ⇒ *addr* **where**
addr-of-array (*Array x*) = *x*

primrec *addr-of-ref* :: *'a ref* ⇒ *addr* **where**
addr-of-ref (*Ref x*) = *x*

lemma *addr-of-array-inj* [*simp*]:
addr-of-array a = *addr-of-array a'* ⟷ *a* = *a'*
by (*cases a*, *cases a'*) *simp-all*

lemma *addr-of-ref-inj* [*simp*]:
addr-of-ref r = *addr-of-ref r'* ⟷ *r* = *r'*
by (*cases r*, *cases r'*) *simp-all*

instance *array* :: (*type*) *countable*
by (*rule countable-classI* [*of addr-of-array*]) *simp*

instance *ref* :: (*type*) *countable*
by (*rule countable-classI* [*of addr-of-ref*]) *simp*

setup ⟪
Sign.add-const-constraint (@{*const-name Array*}, *SOME* @{*typ nat* ⇒ *'a*::*heap array*})
#> *Sign.add-const-constraint* (@{*const-name Ref*}, *SOME* @{*typ nat* ⇒ *'a*::*heap ref*})
#> *Sign.add-const-constraint* (@{*const-name addr-of-array*}, *SOME* @{*typ 'a*::*heap array* ⇒ *nat*})
#> *Sign.add-const-constraint* (@{*const-name addr-of-ref*}, *SOME* @{*typ 'a*::*heap ref* ⇒ *nat*})
⟫

types *heap-rep* = *nat* — representable values

record *heap* =
arrays :: *rtype* ⇒ *addr* ⇒ *heap-rep list*

refs :: *rtype* ⇒ *addr* ⇒ *heap-rep*
lim :: *addr*

definition *empty* :: *heap* **where**
empty = (|*arrays* = (λ-. *arbitrary*), *refs* = (λ-. *arbitrary*), *lim* = *0*|) — why arbitrary?

31.3 Imperative references and arrays

References and arrays are developed in parallel, but keeping them separate makes some later proofs simpler.

31.3.1 Primitive operations

definition
new-ref :: *heap* ⇒ ('*a*::*heap*) *ref* × *heap* **where**
new-ref h = (*let l* = *lim h in* (*Ref l*, *h*(|*lim* := *l* + *1*|)))

definition
new-array :: *heap* ⇒ ('*a*::*heap*) *array* × *heap* **where**
new-array h = (*let l* = *lim h in* (*Array l*, *h*(|*lim* := *l* + *1*|)))

definition
ref-present :: '*a*::*heap ref* ⇒ *heap* ⇒ *bool* **where**
ref-present r h ⟷ *addr-of-ref r* < *lim h*

definition
array-present :: '*a*::*heap array* ⇒ *heap* ⇒ *bool* **where**
array-present a h ⟷ *addr-of-array a* < *lim h*

definition
get-ref :: '*a*::*heap ref* ⇒ *heap* ⇒ '*a* **where**
get-ref r h = *from-nat* (*refs h* (*RTYPE*('*a*)) (*addr-of-ref r*))

definition
get-array :: '*a*::*heap array* ⇒ *heap* ⇒ '*a list* **where**
get-array a h = *map from-nat* (*arrays h* (*RTYPE*('*a*)) (*addr-of-array a*))

definition
set-ref :: '*a*::*heap ref* ⇒ '*a* ⇒ *heap* ⇒ *heap* **where**
set-ref r x =
refs-update (λ*h*. *h*(*RTYPE*('*a*) := ((*h* (*RTYPE*('*a*))) (*addr-of-ref r*:=*to-nat x*))))

definition
set-array :: '*a*::*heap array* ⇒ '*a list* ⇒ *heap* ⇒ *heap* **where**
set-array a x =
arrays-update (λ*h*. *h*(*RTYPE*('*a*) := ((*h* (*RTYPE*('*a*))) (*addr-of-array a*:=*map to-nat x*))))

31.3.2 Interface operations

definition
ref :: *'a* ⇒ *heap* ⇒ *'a*::*heap ref* × *heap* **where**
ref x h = (*let* (*r*, *h'*) = *new-ref h*;
h'' = *set-ref r x h'*
in (*r*, *h''*))

definition
array :: *nat* ⇒ *'a* ⇒ *heap* ⇒ *'a*::*heap array* × *heap* **where**
array n x h = (*let* (*r*, *h'*) = *new-array h*;
h'' = *set-array r* (*replicate n x*) *h'*
in (*r*, *h''*))

definition
array-of-list :: *'a list* ⇒ *heap* ⇒ *'a*::*heap array* × *heap* **where**
array-of-list xs h = (*let* (*r*, *h'*) = *new-array h*;
h'' = *set-array r xs h'*
in (*r*, *h''*))

definition
upd :: *'a*::*heap array* ⇒ *nat* ⇒ *'a* ⇒ *heap* ⇒ *heap* **where**
upd a i x h = *set-array a* ((*get-array a h*)[*i*:=*x*]) *h*

definition
length :: *'a*::*heap array* ⇒ *heap* ⇒ *nat* **where**
length a h = *size* (*get-array a h*)

definition
array-ran :: (*'a*::*heap*) *option array* ⇒ *heap* ⇒ *'a set* **where**
array-ran a h = {*e*. *Some e* ∈ *set* (*get-array a h*)}
— FIXME

31.3.3 Reference equality

The following relations are useful for comparing arrays and references.

definition
noteq-refs :: (*'a*::*heap*) *ref* ⇒ (*'b*::*heap*) *ref* ⇒ *bool* (**infix** =!= *70*)
where
r =!= *s* ⟷ *RTYPE*(*'a*) ≠ *RTYPE*(*'b*) ∨ *addr-of-ref r* ≠ *addr-of-ref s*

definition
noteq-arrs :: (*'a*::*heap*) *array* ⇒ (*'b*::*heap*) *array* ⇒ *bool* (**infix** =!!= *70*)
where
r =!!= *s* ⟷ *RTYPE*(*'a*) ≠ *RTYPE*(*'b*) ∨ *addr-of-array r* ≠ *addr-of-array s*

lemma *noteq-refs-sym*: *r* =!= *s* ⟹ *s* =!= *r*
and *noteq-arrs-sym*: *a* =!!= *b* ⟹ *b* =!!= *a*
and *unequal-refs* [*simp*]: *r* ≠ *r'* ⟷ *r* =!= *r'* — same types!

and *unequal-arrs* [*simp*]: $a \neq a' \longleftrightarrow a =!!= a'$
unfolding *noteq-refs-def noteq-arrs-def* **by** *auto*

lemma *present-new-ref*: *ref-present r h* $\Longrightarrow$ $r =!= fst\ (ref\ v\ h)$
by (*simp add*: *ref-present-def new-ref-def ref-def Let-def noteq-refs-def*)

lemma *present-new-arr*: *array-present a h* $\Longrightarrow$ $a =!!= fst\ (array\ v\ x\ h)$
by (*simp add*: *array-present-def noteq-arrs-def new-array-def array-def Let-def*)

31.3.4 Properties of heap containers

Properties of imperative arrays

FIXME: Does there exist a "canonical" array axiomatisation in the literature?

lemma *array-get-set-eq* [*simp*]: *get-array r* (*set-array r x h*) = *x*
by (*simp add*: *get-array-def set-array-def*)

lemma *array-get-set-neq* [*simp*]: $r =!!= s \Longrightarrow$ *get-array r* (*set-array s x h*) = *get-array r h*
by (*simp add*: *noteq-arrs-def get-array-def set-array-def*)

lemma *set-array-same* [*simp*]:
set-array r x (*set-array r y h*) = *set-array r x h*
by (*simp add*: *set-array-def*)

lemma *array-set-set-swap*:
$r =!!= r' \Longrightarrow$ *set-array r x* (*set-array r' x' h*) = *set-array r' x'* (*set-array r x h*)
by (*simp add*: *Let-def expand-fun-eq noteq-arrs-def set-array-def*)

lemma *array-ref-set-set-swap*:
set-array r x (*set-ref r' x' h*) = *set-ref r' x'* (*set-array r x h*)
by (*simp add*: *Let-def expand-fun-eq set-array-def set-ref-def*)

lemma *get-array-upd-eq* [*simp*]:
get-array a (*upd a i v h*) = (*get-array a h*) [*i* := *v*]
by (*simp add*: *upd-def*)

lemma *nth-upd-array-neq-array* [*simp*]:
$a =!!= b \Longrightarrow$ *get-array a* (*upd b j v h*) ! *i* = *get-array a h* ! *i*
by (*simp add*: *upd-def noteq-arrs-def*)

lemma *get-arry-array-upd-elem-neqIndex* [*simp*]:
$i \neq j \Longrightarrow$ *get-array a* (*upd a j v h*) ! *i* = *get-array a h* ! *i*
by *simp*

lemma *length-upd-eq* [*simp*]:
length a (*upd a i v h*) = *length a h*
by (*simp add*: *length-def upd-def*)

lemma *length-upd-neq* [*simp*]:
length a (*upd b i v h*) = *length a h*
by (*simp add*: *upd-def length-def set-array-def get-array-def*)

lemma *upd-swap-neqArray*:
a =!!= *a'* ⟹
upd a i v (*upd a' i' v' h*)
= *upd a' i' v'* (*upd a i v h*)
apply (*unfold upd-def*)
apply *simp*
apply (*subst array-set-set-swap*, *assumption*)
apply (*subst array-get-set-neq*)
apply (*erule noteq-arrs-sym*)
apply (*simp*)
done

lemma *upd-swap-neqIndex*:
⟦ *i* ≠ *i'* ⟧ ⟹ *upd a i v* (*upd a i' v' h*) = *upd a i' v'* (*upd a i v h*)
by (*auto simp add*: *upd-def array-set-set-swap list-update-swap*)

lemma *get-array-init-array-list*:
get-array (*fst* (*array-of-list ls h*)) (*snd* (*array-of-list ls' h*)) = *ls'*
by (*simp add*: *Let-def split-def array-of-list-def*)

lemma *set-array*:
set-array (*fst* (*array-of-list ls h*))
new-ls (*snd* (*array-of-list ls h*))
= *snd* (*array-of-list new-ls h*)
by (*simp add*: *Let-def split-def array-of-list-def*)

lemma *array-present-upd* [*simp*]:
array-present a (*upd b i v h*) = *array-present a h*
by (*simp add*: *upd-def array-present-def set-array-def get-array-def*)

lemma *array-of-list-replicate*:
array-of-list (*replicate n x*) = *array n x*
by (*simp add*: *expand-fun-eq array-of-list-def array-def*)

Properties of imperative references

lemma *next-ref-fresh* [*simp*]:
assumes (*r*, *h'*) = *new-ref h*
shows ¬ *ref-present r h*
using *assms* **by** (*cases h*) (*auto simp add*: *new-ref-def ref-present-def Let-def*)

lemma *next-ref-present* [*simp*]:
assumes (*r*, *h'*) = *new-ref h*
shows *ref-present r h'*
using *assms* **by** (*cases h*) (*auto simp add*: *new-ref-def ref-present-def Let-def*)

lemma *ref-get-set-eq* [*simp*]: *get-ref r* (*set-ref r x h*) = *x*
by (*simp add*: *get-ref-def set-ref-def*)

lemma *ref-get-set-neq* [*simp*]: *r* =!= *s* ⟹ *get-ref r* (*set-ref s x h*) = *get-ref r h*
by (*simp add*: *noteq-refs-def get-ref-def set-ref-def*)

lemma *ref-set-get*: *set-ref r* (*get-ref r h*) *h* = *h*
apply (*simp add*: *set-ref-def get-ref-def*)
oops

lemma *set-ref-same*[*simp*]:
set-ref r x (*set-ref r y h*) = *set-ref r x h*
by (*simp add*: *set-ref-def*)

lemma *ref-set-set-swap*:
r =!= *r'* ⟹ *set-ref r x* (*set-ref r' x' h*) = *set-ref r' x'* (*set-ref r x h*)
by (*simp add*: *Let-def expand-fun-eq noteq-refs-def set-ref-def*)

lemma *ref-new-set*: *fst* (*ref v* (*set-ref r v' h*)) = *fst* (*ref v h*)
by (*simp add*: *ref-def new-ref-def set-ref-def Let-def*)

lemma *ref-get-new* [*simp*]:
get-ref (*fst* (*ref v h*)) (*snd* (*ref v' h*)) = *v'*
by (*simp add*: *ref-def Let-def split-def*)

lemma *ref-set-new* [*simp*]:
set-ref (*fst* (*ref v h*)) *new-v* (*snd* (*ref v h*)) = *snd* (*ref new-v h*)
by (*simp add*: *ref-def Let-def split-def*)

lemma *ref-get-new-neq*: *r* =!= (*fst* (*ref v h*)) ⟹
get-ref r (*snd* (*ref v h*)) = *get-ref r h*
by (*simp add*: *get-ref-def set-ref-def ref-def Let-def new-ref-def noteq-refs-def*)

lemma *lim-set-ref* [*simp*]:
lim (*set-ref r v h*) = *lim h*
by (*simp add*: *set-ref-def*)

lemma *ref-present-new-ref* [*simp*]:
ref-present r h ⟹ *ref-present r* (*snd* (*ref v h*))
by (*simp add*: *new-ref-def ref-present-def ref-def Let-def*)

lemma *ref-present-set-ref* [*simp*]:
ref-present r (*set-ref r' v h*) = *ref-present r h*
by (*simp add*: *set-ref-def ref-present-def*)

lemma *array-ranI*: ⟦ *Some b* = *get-array a h* ! *i*; *i* < *Heap.length a h* ⟧ ⟹ *b* ∈

array-ran a h
unfolding *array-ran-def Heap.length-def* **by** *simp*

lemma *array-ran-upd-array-Some*:
assumes *cl* ∈ *array-ran a* (*Heap.upd a i* (*Some b*) *h*)
shows *cl* ∈ *array-ran a h* ∨ *cl* = *b*
proof −
have *set* (*get-array a h*[*i* := *Some b*]) ⊆ *insert* (*Some b*) (*set* (*get-array a h*))
by (*rule set-update-subset-insert*)
with *assms* **show** *?thesis*
unfolding *array-ran-def Heap.upd-def* **by** *fastsimp*
qed

lemma *array-ran-upd-array-None*:
assumes *cl* ∈ *array-ran a* (*Heap.upd a i None h*)
shows *cl* ∈ *array-ran a h*
proof −
have *set* (*get-array a h*[*i* := *None*]) ⊆ *insert None* (*set* (*get-array a h*)) **by** (*rule set-update-subset-insert*)
with *assms* **show** *?thesis*
unfolding *array-ran-def Heap.upd-def* **by** *auto*
qed

Non-interaction between imperative array and imperative references

lemma *get-array-set-ref* [*simp*]: *get-array a* (*set-ref r v h*) = *get-array a h*
by (*simp add*: *get-array-def set-ref-def*)

lemma *nth-set-ref* [*simp*]: *get-array a* (*set-ref r v h*) ! *i* = *get-array a h* ! *i*
by *simp*

lemma *get-ref-upd* [*simp*]: *get-ref r* (*upd a i v h*) = *get-ref r h*
by (*simp add*: *get-ref-def set-array-def upd-def*)

lemma *new-ref-upd*: *fst* (*ref v* (*upd a i v′ h*)) = *fst* (*ref v h*)
by (*simp add*: *set-array-def get-array-def Let-def ref-new-set upd-def ref-def new-ref-def*)

not actually true ???

lemma *upd-set-ref-swap*: *upd a i v* (*set-ref r v′ h*) = *set-ref r v′* (*upd a i v h*)
apply (*case-tac a*)
apply (*simp add*: *Let-def upd-def*)
apply *auto*
oops

lemma *length-new-ref*[*simp*]:
length a (*snd* (*ref v h*)) = *length a h*
by (*simp add*: *get-array-def set-ref-def length-def new-ref-def ref-def Let-def*)

lemma *get-array-new-ref* [*simp*]:
get-array a (*snd* (*ref v h*)) = *get-array a h*

by (*simp add*: *new-ref-def ref-def set-ref-def get-array-def Let-def*)

lemma *ref-present-upd* [*simp*]:
ref-present r (*upd a i v h*) = *ref-present r h*
by (*simp add*: *upd-def ref-present-def set-array-def get-array-def*)

lemma *array-present-set-ref* [*simp*]:
array-present a (*set-ref r v h*) = *array-present a h*
by (*simp add*: *array-present-def set-ref-def*)

lemma *array-present-new-ref* [*simp*]:
array-present a h ⟹ *array-present a* (*snd* (*ref v h*))
by (*simp add*: *array-present-def new-ref-def ref-def Let-def*)

hide (**open**) *const empty array array-of-list upd length ref*

end

32 Heap-Monad: A monad with a polymorphic heap

theory *Heap-Monad*
imports *Heap*
begin

32.1 The monad

32.1.1 Monad combinators

datatype *exception* = *Exn*

Monadic heap actions either produce values and transform the heap, or fail

datatype *'a Heap* = *Heap heap* ⇒ (*'a* + *exception*) × *heap*

primrec
execute :: *'a Heap* ⇒ *heap* ⇒ (*'a* + *exception*) × *heap* **where**
execute (*Heap f*) = *f*
lemmas [*code del*] = *execute.simps*

lemma *Heap-execute* [*simp*]:
Heap (*execute f*) = *f* **by** (*cases f*) *simp-all*

lemma *Heap-eqI*:
(⋀*h*. *execute f h* = *execute g h*) ⟹ *f* = *g*
by (*cases f*, *cases g*) (*auto simp*: *expand-fun-eq*)

lemma *Heap-eqI'*:

$(\bigwedge h.\ (\lambda x.\ execute\ (f\ x)\ h) = (\lambda y.\ execute\ (g\ y)\ h)) \Longrightarrow f = g$
by (*auto simp*: *expand-fun-eq intro*: *Heap-eqI*)

lemma *Heap-strip*: $(\bigwedge f.\ PROP\ P\ f) \equiv (\bigwedge g.\ PROP\ P\ (Heap\ g))$
proof
fix $g :: heap \Rightarrow ('a + exception) \times heap$
assume $\bigwedge f.\ PROP\ P\ f$
then show $PROP\ P\ (Heap\ g)$ **.**
next
fix $f :: 'a\ Heap$
assume *assm*: $\bigwedge g.\ PROP\ P\ (Heap\ g)$
then have $PROP\ P\ (Heap\ (execute\ f))$ **.**
then show $PROP\ P\ f$ **by** *simp*
qed

definition
$heap :: (heap \Rightarrow 'a \times heap) \Rightarrow 'a\ Heap$ **where**
[*code del*]: $heap\ f = Heap\ (\lambda h.\ apfst\ Inl\ (f\ h))$

lemma *execute-heap* [*simp*]:
$execute\ (heap\ f)\ h = apfst\ Inl\ (f\ h)$
by (*simp add*: *heap-def*)

definition
$run :: 'a\ Heap \Rightarrow 'a\ Heap$ **where**
run-drop [*code del*]: $run\ f = f$

definition
$bindM :: 'a\ Heap \Rightarrow ('a \Rightarrow 'b\ Heap) \Rightarrow 'b\ Heap$ (**infixl** $>>=\ 54$) **where**
[*code del*]: $f >>= g = Heap\ (\lambda h.\ case\ execute\ f\ h\ of$
$(Inl\ x,\ h') \Rightarrow execute\ (g\ x)\ h'$
$\mid r \Rightarrow r)$

notation
$bindM$ (**infixl** $\gg=\ 54$)

abbreviation
$chainM :: 'a\ Heap \Rightarrow 'b\ Heap \Rightarrow 'b\ Heap$ (**infixl** $>>\ 54$) **where**
$f >> g \equiv f >>= (\lambda\text{-}.\ g)$

notation
$chainM$ (**infixl** $\gg\ 54$)

definition
$return :: 'a \Rightarrow 'a\ Heap$ **where**
[*code del*]: $return\ x = heap\ (Pair\ x)$

lemma *execute-return* [*simp*]:
$execute\ (return\ x)\ h = apfst\ Inl\ (x,\ h)$

by (*simp add*: *return-def*)

definition
raise :: *string* ⇒ *'a Heap* **where** — the string is just decoration
[*code del*]: *raise s* = *Heap* (*Pair* (*Inr Exn*))

notation (*latex* **output**)
raise (*raise*)

lemma *execute-raise* [*simp*]:
execute (*raise s*) *h* = (*Inr Exn*, *h*)
by (*simp add*: *raise-def*)

32.1.2 do-syntax

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

nonterminals *do-expr*

syntax
-*do* :: *do-expr* ⇒ *'a*
((*do* (-)//*done*) [*12*] *100*)
-*bindM* :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(- <− -;//- [*1000*, *13*, *12*] *12*)
-*chainM* :: *'a* ⇒ *do-expr* ⇒ *do-expr*
(-;//- [*13*, *12*] *12*)
-*let* :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*let* - = -;//- [*1000*, *13*, *12*] *12*)
-*nil* :: *'a* ⇒ *do-expr*
(- [*12*] *12*)

syntax (*xsymbols*)
-*bindM* :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(- ← -;//- [*1000*, *13*, *12*] *12*)
syntax (*latex* **output**)
-*do* :: *do-expr* ⇒ *'a*
((*do* (-)) [*12*] *100*)
-*let* :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*let* - = -;//- [*1000*, *13*, *12*] *12*)
notation (*latex* **output**)
return (*return*)

translations
-*do f* => *CONST run f*
-*bindM x f g* => *f* ≫= (λ*x*. *g*)
-*chainM f g* => *f* ≫ *g*
-*let x t f* => *CONST Let t* (λ*x*. *f*)
-*nil f* => *f*

print-translation ⟪
let
  fun dest-abs-eta (Abs (abs as (-, ty, -))) =
      let
        val (v, t) = Syntax.variant-abs abs;
      in ((v, ty), t) end
    | dest-abs-eta t =
      let
        val (v, t) = Syntax.variant-abs (, dummyT, t $ Bound 0);
      in ((v, dummyT), t) end
  fun unfold-monad (Const (@{const-syntax bindM}, -) $ f $ g) =
      let
        val ((v, ty), g') = dest-abs-eta g;
        val v-used = fold-aterms
          (fn Free (w, -) => (fn s => s orelse v = w) | - => I) g' false;
      in if v-used then
        Const (-bindM, dummyT) $ Free (v, ty) $ f $ unfold-monad g'
      else
        Const (-chainM, dummyT) $ f $ unfold-monad g'
      end
    | unfold-monad (Const (@{const-syntax chainM}, -) $ f $ g) =
      Const (-chainM, dummyT) $ f $ unfold-monad g
    | unfold-monad (Const (@{const-syntax Let}, -) $ f $ g) =
      let
        val ((v, ty), g') = dest-abs-eta g;
      in Const (-let, dummyT) $ Free (v, ty) $ f $ unfold-monad g' end
    | unfold-monad (Const (@{const-syntax Pair}, -) $ f) =
      Const (return, dummyT) $ f
    | unfold-monad f = f;
  fun tr' (f::ts) =
    list-comb (Const (-do, dummyT) $ unfold-monad f, ts)
in [(@{const-syntax run}, tr')] end;
⟫

32.1.3 Plain evaluation

definition
  evaluate :: *'a Heap* ⇒ *'a*
where
  [*code del*]: *evaluate f = (case execute f Heap.empty*
    of (Inl x, -) ⇒ x)

32.2 Monad properties

32.2.1 Superfluous runs

run is just a doodle

lemma *run-simp* [*simp*]:
  ⋀*f*. *run (run f) = run f*

$\bigwedge f\ g.\ run\ f \gg\!= g = f \gg\!= g$
$\bigwedge f\ g.\ run\ f \gg g = f \gg g$
$\bigwedge f\ g.\ f \gg\!= (\lambda x.\ run\ g) = f \gg\!= (\lambda x.\ g)$
$\bigwedge f\ g.\ f \gg run\ g = f \gg g$
$\bigwedge f.\ f = run\ g \longleftrightarrow f = g$
$\bigwedge f.\ run\ f = g \longleftrightarrow f = g$
unfolding *run-drop* **by** *rule+*

32.2.2 Monad laws

lemma *return-bind*: $return\ x \gg\!= f = f\ x$
by (*simp add*: *bindM-def return-def*)

lemma *bind-return*: $f \gg\!= return = f$
proof (*rule Heap-eqI*)
fix h
show $execute\ (f \gg\!= return)\ h = execute\ f\ h$
by (*auto simp add*: *bindM-def return-def split*: *sum.splits prod.splits*)
qed

lemma *bind-bind*: $(f \gg\!= g) \gg\!= h = f \gg\!= (\lambda x.\ g\ x \gg\!= h)$
by (*rule Heap-eqI*) (*auto simp add*: *bindM-def split*: *split*: *sum.splits prod.splits*)

lemma *bind-bind'*: $f \gg\!= (\lambda x.\ g\ x \gg\!= h\ x) = f \gg\!= (\lambda x.\ g\ x \gg\!= (\lambda y.\ return\ (x, y))) \gg\!= (\lambda(x, y).\ h\ x\ y)$
by (*rule Heap-eqI*) (*auto simp add*: *bindM-def split*: *split*: *sum.splits prod.splits*)

lemma *raise-bind*: $raise\ e \gg\!= f = raise\ e$
by (*simp add*: *raise-def bindM-def*)

lemmas *monad-simp* = *return-bind bind-return bind-bind raise-bind*

32.3 Generic combinators

definition
$liftM :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b\ Heap$
where
$liftM\ f = return\ o\ f$

definition
$compM :: ('a \Rightarrow 'b\ Heap) \Rightarrow ('b \Rightarrow 'c\ Heap) \Rightarrow 'a \Rightarrow 'c\ Heap$ (**infixl** $>>==$ *54*)
where
$(f >>== g) = (\lambda x.\ f\ x \gg\!= g)$

notation
compM (**infixl** $\gg\!==$ *54*)

lemma *liftM-collapse*: $liftM\ f\ x = return\ (f\ x)$
by (*simp add*: *liftM-def*)

lemma *liftM-compM*: *liftM f ⟫== g = g o f*
by (*auto intro*: *Heap-eqI' simp add*: *expand-fun-eq liftM-def compM-def bindM-def*)

lemma *compM-return*: *f ⟫== return = f*
by (*simp add*: *compM-def monad-simp*)

lemma *compM-compM*: *(f ⟫== g) ⟫== h = f ⟫== (g ⟫== h)*
by (*simp add*: *compM-def monad-simp*)

lemma *liftM-bind*:
(λx. liftM f x ⟫= liftM g) = liftM (λx. g (f x))
by (*rule Heap-eqI'*) (*simp add*: *monad-simp liftM-def bindM-def*)

lemma *liftM-comp*:
liftM f o g = liftM (f o g)
by (*rule Heap-eqI'*) (*simp add*: *liftM-def*)

lemmas *monad-simp' = monad-simp liftM-compM compM-return*
compM-compM liftM-bind liftM-comp

primrec
mapM :: *('a ⇒ 'b Heap) ⇒ 'a list ⇒ 'b list Heap*
where
mapM f [] = return []
| mapM f (x#xs) = do y ← f x;
ys ← mapM f xs;
return (y # ys)
done

primrec
foldM :: *('a ⇒ 'b ⇒ 'b Heap) ⇒ 'a list ⇒ 'b ⇒ 'b Heap*
where
foldM f [] s = return s
| foldM f (x#xs) s = f x s ⟫= foldM f xs

hide (**open**) *const heap execute*

32.4 Code generator setup

32.4.1 Logical intermediate layer

definition
Fail :: *message-string ⇒ exception*
where
[*code func del*]: *Fail s = Exn*

definition
raise-exc :: *exception ⇒ 'a Heap*
where

[*code func del*]: *raise-exc e* = *raise* []

lemma *raise-raise-exc* [*code func*, *code inline*]:
raise s = *raise-exc* (*Fail* (*STR s*))
unfolding *Fail-def raise-exc-def raise-def* **..**

hide (**open**) *const Fail raise-exc*

32.4.2 SML

code-type *Heap* (*SML unit*/ −>/ -)
code-const *Heap* (*SML raise*/ (*Fail*/ *bare Heap*))
code-monad *run op* ≫= *return* () *SML*
code-const *run* (*SML* -)
code-const *return* (*SML* (*fn*/ ()/ =>/ -))
code-const *Heap-Monad.Fail* (*SML Fail*)
code-const *Heap-Monad.raise-exc* (*SML* (*fn*/ ()/ =>/ *raise*/ -))

32.4.3 OCaml

code-type *Heap* (*OCaml* -)
code-const *Heap* (*OCaml failwith*/ *bare Heap*)
code-monad *run op* ≫= *return* () *OCaml*
code-const *run* (*OCaml* -)
code-const *return* (*OCaml* (*fn*/ ()/ =>/ -))
code-const *Heap-Monad.Fail* (*OCaml Failure*)
code-const *Heap-Monad.raise-exc* (*OCaml* (*fn*/ ()/ =>/ *raise*/ -))

code-reserved *OCaml Failure raise*

32.4.4 Haskell

Adaption layer

code-include *Haskell STMonad*
⟨⟨*import qualified Control.Monad*;
import qualified Control.Monad.ST;
import qualified Data.STRef;
import qualified Data.Array.ST;

type ST s a = *Control.Monad.ST.ST s a*;
type STRef s a = *Data.STRef.STRef s a*;
type STArray s a = *Data.Array.ST.STArray s Integer a*;

runST :: (*forall s*. *ST s a*) −> *a*;
runST s = *Control.Monad.ST.runST s*;

newSTRef = *Data.STRef.newSTRef*;
readSTRef = *Data.STRef.readSTRef*;
writeSTRef = *Data.STRef.writeSTRef*;

newArray :: (*Integer*, *Integer*) -> *a* -> *ST s* (*STArray s a*);
newArray = *Data.Array.ST.newArray*;

newListArray :: (*Integer*, *Integer*) -> [*a*] -> *ST s* (*STArray s a*);
newListArray = *Data.Array.ST.newListArray*;

length :: *STArray s a* -> *ST s Integer*;
length a = *Control.Monad.liftM snd* (*Data.Array.ST.getBounds a*);

readArray :: *STArray s a* -> *Integer* -> *ST s a*;
readArray = *Data.Array.ST.readArray*;

writeArray :: *STArray s a* -> *Integer* -> *a* -> *ST s* ();
writeArray = *Data.Array.ST.writeArray*;⟩⟩

code-reserved *Haskell ST STRef Array*
runST
newSTRef reasSTRef writeSTRef
newArray newListArray bounds readArray writeArray

Monad

code-type *Heap* (*Haskell ST '-s -*)
code-const *Heap* (*Haskell error bare Heap*))
code-const *evaluate* (*Haskell runST*)
code-monad *run op* ≫= *Haskell*
code-const *return* (*Haskell return*)
code-const *Heap-Monad.Fail* (*Haskell -*)
code-const *Heap-Monad.raise-exc* (*Haskell error*)

end

33 Array: Monadic arrays

theory *Array*
imports *Heap-Monad Code-Index*
begin

33.1 Primitives

definition
new :: *nat* ⇒ *'a*::*heap* ⇒ *'a array Heap* **where**
[*code del*]: *new n x* = *Heap-Monad.heap* (*Heap.array n x*)

definition
of-list :: *'a*::*heap list* ⇒ *'a array Heap* **where**
[*code del*]: *of-list xs* = *Heap-Monad.heap* (*Heap.array-of-list xs*)

definition
length :: *'a*::*heap array* ⇒ *nat Heap* **where**
[*code del*]: *length arr* = *Heap-Monad.heap* (λ*h*. (*Heap.length arr h*, *h*))

definition
nth :: *'a*::*heap array* ⇒ *nat* ⇒ *'a Heap*
where
[*code del*]: *nth a i* = (*do len* ← *length a*;
(*if i* < *len*
then Heap-Monad.heap (λ*h*. (*get-array a h* ! *i*, *h*))
else raise (*''array lookup*: *index out of range''*))
done)

— FIXME adjustion for List theory
no-syntax
nth :: *'a list* ⇒ *nat* ⇒ *'a* (**infixl** ! *100*)

abbreviation
nth-list :: *'a list* ⇒ *nat* ⇒ *'a* (**infixl** ! *100*)
where
nth-list ≡ *List.nth*

definition
upd :: *nat* ⇒ *'a* ⇒ *'a*::*heap array* ⇒ *'a*::*heap array Heap*
where
[*code del*]: *upd i x a* = (*do len* ← *length a*;
(*if i* < *len*
then Heap-Monad.heap (λ*h*. (*a*, *Heap.upd a i x h*))
else raise (*''array update*: *index out of range''*))
done)

lemma *upd-return*:
upd i x a ≫ *return a* = *upd i x a*
proof (*rule Heap-eqI*)
fix *h*
obtain *len h'* **where** *Heap-Monad.execute* (*Array.length a*) *h* = (*len*, *h'*)
by (*cases Heap-Monad.execute* (*Array.length a*) *h*)
then show *Heap-Monad.execute* (*upd i x a* ≫ *return a*) *h* = *Heap-Monad.execute* (*upd i x a*) *h*
by (*auto simp add*: *upd-def bindM-def run-drop split*: *sum.split*)
qed

33.2 Derivates

definition
map-entry :: *nat* ⇒ (*'a*::*heap* ⇒ *'a*) ⇒ *'a array* ⇒ *'a array Heap*
where
map-entry i f a = (*do*
x ← *nth a i*;

```
    upd i (f x) a
  done)

definition
  swap :: nat ⇒ 'a ⇒ 'a::heap array ⇒ 'a Heap
where
  swap i x a = (do
    y ← nth a i;
    upd i x a;
    return x
  done)

definition
  make :: nat ⇒ (nat ⇒ 'a::heap) ⇒ 'a array Heap
where
  make n f = of-list (map f [0 ..< n])

definition
  freeze :: 'a::heap array ⇒ 'a list Heap
where
  freeze a = (do
    n ← length a;
    mapM (nth a) [0..<n]
  done)

definition
  map :: ('a::heap ⇒ 'a) ⇒ 'a array ⇒ 'a array Heap
where
  map f a = (do
    n ← length a;
    foldM (λn. map-entry n f) [0..<n] a
  done)

hide (open) const new map — avoid clashed with some popular names
```

33.3 Properties

```
lemma array-make [code func]:
  Array.new n x = make n (λ-. x)
  by (induct n) (simp-all add: make-def new-def Heap-Monad.heap-def
    monad-simp array-of-list-replicate [symmetric]
    map-replicate-trivial replicate-append-same
    of-list-def)

lemma array-of-list-make [code func]:
  of-list xs = make (List.length xs) (λn. xs ! n)
  unfolding make-def map-nth ..
```

33.4 Code generator setup

33.4.1 Logical intermediate layer

definition *new'* **where**
  [*code del*]: *new' = Array.new o nat-of-index*
hide (**open**) *const new'*
lemma [*code func*]:
  Array.new = Array.new' o index-of-nat
  by (*simp add*: *new'-def o-def*)

definition *of-list'* **where**
  [*code del*]: *of-list' i xs = Array.of-list (take (nat-of-index i) xs)*
hide (**open**) *const of-list'*
lemma [*code func*]:
  Array.of-list xs = Array.of-list' (index-of-nat (List.length xs)) xs
  by (*simp add*: *of-list'-def*)

definition *make'* **where**
  [*code del*]: *make' i f = Array.make (nat-of-index i) (f o index-of-nat)*
hide (**open**) *const make'*
lemma [*code func*]:
  Array.make n f = Array.make' (index-of-nat n) (f o nat-of-index)
  by (*simp add*: *make'-def o-def*)

definition *length'* **where**
  [*code del*]: *length' = Array.length ⤜== liftM index-of-nat*
hide (**open**) *const length'*
lemma [*code func*]:
  Array.length = Array.length' ⤜== liftM nat-of-index
  by (*simp add*: *length'-def monad-simp'*,
    simp add: *liftM-def comp-def monad-simp*,
    simp add: *monad-simp'*)

definition *nth'* **where**
  [*code del*]: *nth' a = Array.nth a o nat-of-index*
hide (**open**) *const nth'*
lemma [*code func*]:
  Array.nth a n = Array.nth' a (index-of-nat n)
  by (*simp add*: *nth'-def*)

definition *upd'* **where**
  [*code del*]: *upd' a i x = Array.upd (nat-of-index i) x a ⤜ return ()*
hide (**open**) *const upd'*
lemma [*code func*]:
  Array.upd i x a = Array.upd' a (index-of-nat i) x ⤜ return a
  by (*simp add*: *upd'-def monad-simp upd-return*)

33.4.2 SML

code-type *array* (*SML -/ array*)
code-const *Array* (*SML raise/* (*Fail/ bare Array*))
code-const *Array.new'* (*SML* (*fn/* ()*/ =>/ Array.array/* ((-),/ (-))))
code-const *Array.of-list* (*SML* (*fn/* ()*/ =>/ Array.fromList/* -))
code-const *Array.make'* (*SML* (*fn/* ()*/ =>/ Array.tabulate/* ((-),/ (-))))
code-const *Array.length'* (*SML* (*fn/* ()*/ =>/ Array.length/* -))
code-const *Array.nth'* (*SML* (*fn/* ()*/ =>/ Array.sub/* ((-),/ (-))))
code-const *Array.upd'* (*SML* (*fn/* ()*/ =>/ Array.update/* ((-),/ (-),/ (-))))

code-reserved *SML Array*

33.4.3 OCaml

code-type *array* (*OCaml -/ array*)
code-const *Array* (*OCaml failwith/ bare Array*)
code-const *Array.new'* (*OCaml* (*fn/* ()*/ =>/ Array.make/ -/* -))
code-const *Array.of-list* (*OCaml* (*fn/* ()*/ =>/ Array.of'-list/* -))
code-const *Array.make'* (*OCaml* (*fn/* ()*/ =>/ Array.init/ -/* -))
code-const *Array.length'* (*OCaml* (*fn/* ()*/ =>/ Array.length/* -))
code-const *Array.nth'* (*OCaml* (*fn/* ()*/ =>/ Array.get/ -/* -))
code-const *Array.upd'* (*OCaml* (*fn/* ()*/ =>/ Array.set/ -/ -/* -))

code-reserved *OCaml Array*

33.4.4 Haskell

code-type *array* (*Haskell STArray '-s -*)
code-const *Array* (*Haskell error/ bare Array*)
code-const *Array.new'* (*Haskell newArray/* (*0*,/ -))
code-const *Array.of-list'* (*Haskell newListArray/* (*0*,/ -))
code-const *Array.length'* (*Haskell length*)
code-const *Array.nth'* (*Haskell readArray*)
code-const *Array.upd'* (*Haskell writeArray*)

end

34 Ref: Monadic references

theory *Ref*
imports *Heap-Monad*
begin

Imperative reference operations; modeled after their ML counterparts.
See http://caml.inria.fr/pub/docs/manual-caml-light/node14.15.html and http://www.smlnj.org/doc
level-comparison.html

34.1 Primitives

definition
new :: *'a*::*heap* ⇒ *'a ref Heap* **where**
[*code del*]: *new v* = *Heap-Monad.heap* (*Heap.ref v*)

definition
lookup :: *'a*::*heap ref* ⇒ *'a Heap* (!- *61*) **where**
[*code del*]: *lookup r* = *Heap-Monad.heap* (λ*h*. (*get-ref r h*, *h*))

definition
update :: *'a ref* ⇒ (*'a*::*heap*) ⇒ *unit Heap* (- := - *62*) **where**
[*code del*]: *update r e* = *Heap-Monad.heap* (λ*h*. ((), *set-ref r e h*))

34.2 Derivates

definition
change :: (*'a*::*heap* ⇒ *'a*) ⇒ *'a ref* ⇒ *'a Heap*
where
change f r = (*do x* ← ! *r*;
let y = *f x*;
r := *y*;
return y
done)

hide (**open**) *const new lookup update change*

34.3 Properties

lemma *lookup-chain*:
(!*r* ≫ *f*) = *f*
by (*cases f*)
(*auto simp add*: *Let-def bindM-def lookup-def expand-fun-eq*)

lemma *update-change* [*code func*]:
r := *e* = *Ref.change* (λ-. *e*) *r* ≫ *return* ()
by (*auto simp add*: *monad-simp change-def lookup-chain*)

34.4 Code generator setup

34.4.1 SML

code-type *ref* (*SML* -/ *ref*)
code-const *Ref* (*SML raise*/ (*Fail*/ *bare Ref*))
code-const *Ref.new* (*SML* (*fn*/ ()/ =>/ *ref*/ -))
code-const *Ref.lookup* (*SML* (*fn*/ ()/ =>/ !/ -))
code-const *Ref.update* (*SML* (*fn*/ ()/ =>/ -/ :=/ -))

code-reserved *SML ref*

34.4.2 OCaml

code-type *ref* (*OCaml* -/ *ref*)
code-const *Ref* (*OCaml failwith*/ *bare Ref*))
code-const *Ref.new* (*OCaml* (*fn*/ ()/ =>/ *ref*/ -))
code-const *Ref.lookup* (*OCaml* (*fn*/ ()/ =>/ !/ -))
code-const *Ref.update* (*OCaml* (*fn*/ ()/ =>/ -/ :=/ -))

code-reserved *OCaml ref*

34.4.3 Haskell

code-type *ref* (*Haskell STRef* 'a *s* -)
code-const *Ref* (*Haskell error*/ *bare Ref*)
code-const *Ref.new* (*Haskell newSTRef*)
code-const *Ref.lookup* (*Haskell readSTRef*)
code-const *Ref.update* (*Haskell writeSTRef*)

end

35 Imperative-HOL: Entry point into monadic imperative HOL

theory *Imperative-HOL*
imports *Array Ref*
begin

end

36 Infinite-Set: Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *ATP-Linkup*
begin

36.1 Infinite Sets

Some elementary facts about infinite sets, mostly by Stefan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

abbreviation
infinite :: 'a *set* ⇒ *bool* **where**
infinite S == ¬ *finite S*

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

lemma *infinite-imp-nonempty*: *infinite* S ==> $S \neq \{\}$
by *auto*

lemma *infinite-remove*:
infinite $S \Longrightarrow$ *infinite* $(S - \{a\})$
by *simp*

lemma *Diff-infinite-finite*:
assumes T: *finite* T **and** S: *infinite* S
shows *infinite* $(S - T)$
using T
proof *induct*
from S
show *infinite* $(S - \{\})$ **by** *auto*
next
fix T x
assume *ih*: *infinite* $(S - T)$
have $S - (insert\ x\ T) = (S - T) - \{x\}$
by (*rule Diff-insert*)
with *ih*
show *infinite* $(S - (insert\ x\ T))$
by (*simp add*: *infinite-remove*)
qed

lemma *Un-infinite*: *infinite* $S \Longrightarrow$ *infinite* $(S \cup T)$
by *simp*

lemma *infinite-super*:
assumes T: $S \subseteq T$ **and** S: *infinite* S
shows *infinite* T
proof
assume *finite* T
with T **have** *finite* S **by** (*simp add*: *finite-subset*)
with S **show** *False* **by** *simp*
qed

As a concrete example, we prove that the set of natural numbers is infinite.

lemma *finite-nat-bounded*:
assumes S: *finite* $(S$::*nat set*$)$
shows $\exists k.\ S \subseteq \{..<k\}$ (**is** $\exists k.$ *?bounded* S k)
using S
proof *induct*
have *?bounded* $\{\}$ *0* **by** *simp*
then show $\exists k.$ *?bounded* $\{\}$ k **..**
next
fix S x
assume $\exists k.$ *?bounded* S k
then obtain k **where** k: *?bounded* S k **..**

```
  show ∃k. ?bounded (insert x S) k
  proof (cases x < k)
    case True
    with k show ?thesis by auto
  next
    case False
    with k have ?bounded S (Suc x) by auto
    then show ?thesis by auto
  qed
qed

lemma finite-nat-iff-bounded:
  finite (S::nat set) = (∃k. S ⊆ {..<k})  (is ?lhs = ?rhs)
proof
  assume ?lhs
  then show ?rhs by (rule finite-nat-bounded)
next
  assume ?rhs
  then obtain k where S ⊆ {..<k} ..
  then show finite S
    by (rule finite-subset) simp
qed

lemma finite-nat-iff-bounded-le:
  finite (S::nat set) = (∃k. S ⊆ {..k})  (is ?lhs = ?rhs)
proof
  assume ?lhs
  then obtain k where S ⊆ {..<k}
    by (blast dest: finite-nat-bounded)
  then have S ⊆ {..k} by auto
  then show ?rhs ..
next
  assume ?rhs
  then obtain k where S ⊆ {..k} ..
  then show finite S
    by (rule finite-subset) simp
qed

lemma infinite-nat-iff-unbounded:
  infinite (S::nat set) = (∀m. ∃n. m<n ∧ n∈S)
  (is ?lhs = ?rhs)
proof
  assume ?lhs
  show ?rhs
  proof (rule ccontr)
    assume ¬ ?rhs
    then obtain m where m: ∀n. m<n ⟶ n∉S by blast
    then have S ⊆ {..m}
      by (auto simp add: sym [OF linorder-not-less])
```

```
    with ‹?lhs› show False
      by (simp add: finite-nat-iff-bounded-le)
  qed
next
  assume ?rhs
  show ?lhs
  proof
    assume finite S
    then obtain m where S ⊆ {..m}
      by (auto simp add: finite-nat-iff-bounded-le)
    then have ∀ n. m<n ⟶ n∉S by auto
    with ‹?rhs› show False by blast
  qed
qed

lemma infinite-nat-iff-unbounded-le:
  infinite (S::nat set) = (∀ m. ∃ n. m≤n ∧ n∈S)
  (is ?lhs = ?rhs)
proof
  assume ?lhs
  show ?rhs
  proof
    fix m
    from ‹?lhs› obtain n where m<n ∧ n∈S
      by (auto simp add: infinite-nat-iff-unbounded)
    then have m≤n ∧ n∈S by simp
    then show ∃ n. m ≤ n ∧ n ∈ S ..
  qed
next
  assume ?rhs
  show ?lhs
  proof (auto simp add: infinite-nat-iff-unbounded)
    fix m
    from ‹?rhs› obtain n where Suc m ≤ n ∧ n∈S
      by blast
    then have m<n ∧ n∈S by simp
    then show ∃ n. m < n ∧ n ∈ S ..
  qed
qed
```

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:
  assumes k: ∀ m. k<m ⟶ (∃ n. m<n ∧ n∈S)
  shows infinite (S::nat set)
proof −
  {
    fix m have ∃ n. m<n ∧ n∈S
```

```
    proof (cases k<m)
      case True
      with k show ?thesis by blast
    next
      case False
      from k obtain n where Suc k < n ∧ n∈S by auto
      with False have m<n ∧ n∈S by auto
      then show ?thesis ..
    qed
  }
  then show ?thesis
    by (auto simp add: infinite-nat-iff-unbounded)
qed

lemma nat-infinite [simp]: infinite (UNIV :: nat set)
  by (auto simp add: infinite-nat-iff-unbounded)

lemma nat-not-finite [elim]: finite (UNIV::nat set) ⟹ R
  by simp
```

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S.

```
lemma range-inj-infinite:
  inj (f::nat ⇒ 'a) ⟹ infinite (range f)
proof
  assume inj f
    and  finite (range f)
  then have finite (UNIV::nat set)
    by (auto intro: finite-imageD simp del: nat-infinite)
  then show False by simp
qed

lemma int-infinite [simp]:
  shows infinite (UNIV::int set)
proof −
  from inj-int have infinite (range int) by (rule range-inj-infinite)
  moreover
  have range int ⊆ (UNIV::int set) by simp
  ultimately show infinite (UNIV::int set) by (simp add: infinite-super)
qed
```

The "only if" direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

```
lemma linorder-injI:
  assumes hyp: !!x y. x < (y::'a::linorder) ==> f x ≠ f y
  shows inj f
```

proof (*rule inj-onI*)
fix x y
assume *f-eq*: $f\ x = f\ y$
show $x = y$
proof (*rule linorder-cases*)
assume $x < y$
with *hyp* **have** $f\ x \neq f\ y$ **by** *blast*
with *f-eq* **show** *?thesis* **by** *simp*
next
assume $x = y$
then show *?thesis* **.**
next
assume $y < x$
with *hyp* **have** $f\ y \neq f\ x$ **by** *blast*
with *f-eq* **show** *?thesis* **by** *simp*
qed
qed

lemma *infinite-countable-subset*:
assumes *inf*: *infinite* $(S::'a\ set)$
shows $\exists f.\ inj\ (f::nat \Rightarrow 'a) \wedge range\ f \subseteq S$
proof −
def *Sseq* $\equiv$ *nat-rec* $S\ (\lambda n\ T.\ T - \{SOME\ e.\ e \in T\})$
def *pick* $\equiv \lambda n.\ (SOME\ e.\ e \in Sseq\ n)$
have *Sseq-inf*: $\bigwedge n.$ *infinite* $(Sseq\ n)$
proof −
fix n
show *infinite* $(Sseq\ n)$
proof (*induct n*)
from *inf* **show** *infinite* $(Sseq\ 0)$
by (*simp add*: *Sseq-def*)
next
fix n
assume *infinite* $(Sseq\ n)$ **then show** *infinite* $(Sseq\ (Suc\ n))$
by (*simp add*: *Sseq-def infinite-remove*)
qed
qed
have *Sseq-S*: $\bigwedge n.$ *Sseq* $n \subseteq S$
proof −
fix n
show *Sseq* $n \subseteq S$
by (*induct n*) (*auto simp add*: *Sseq-def*)
qed
have *Sseq-pick*: $\bigwedge n.$ *pick* $n \in Sseq\ n$
proof −
fix n
show *pick* $n \in Sseq\ n$
proof (*unfold pick-def*, *rule someI-ex*)
from *Sseq-inf* **have** *infinite* $(Sseq\ n)$ **.**

```
    then have Sseq n ≠ {} by auto
    then show ∃x. x ∈ Sseq n by auto
   qed
  qed
  with Sseq-S have rng: range pick ⊆ S
   by auto
  have pick-Sseq-gt: ⋀n m. pick n ∉ Sseq (n + Suc m)
  proof −
   fix n m
   show pick n ∉ Sseq (n + Suc m)
    by (induct m) (auto simp add: Sseq-def pick-def)
  qed
  have pick-pick: ⋀n m. pick n ≠ pick (n + Suc m)
  proof −
   fix n m
   from Sseq-pick have pick (n + Suc m) ∈ Sseq (n + Suc m) .
   moreover from pick-Sseq-gt
   have pick n ∉ Sseq (n + Suc m) .
   ultimately show pick n ≠ pick (n + Suc m)
    by auto
  qed
  have inj: inj pick
  proof (rule linorder-injI)
   fix i j :: nat
   assume i < j
   show pick i ≠ pick j
   proof
    assume eq: pick i = pick j
    from ‹i < j› obtain k where j = i + Suc k
     by (auto simp add: less-iff-Suc-add)
    with pick-pick have pick i ≠ pick j by simp
    with eq show False by simp
   qed
  qed
  from rng inj show ?thesis by auto
qed

lemma infinite-iff-countable-subset:
   infinite S = (∃f. inj (f::nat ⇒ 'a) ∧ range f ⊆ S)
  by (auto simp add: infinite-countable-subset range-inj-infinite infinite-super)
```

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

```
lemma inf-img-fin-dom:
  assumes img: finite (f‘A) and dom: infinite A
  shows ∃y ∈ f‘A. infinite (f −‘ {y})
proof (rule ccontr)
```

assume ¬ *?thesis*
with *img* **have** *finite* (*UN y:f'A. f −' {y}*) **by** (*blast intro*: *finite-UN-I*)
moreover have $A \subseteq$ (*UN y:f'A. f −' {y}*) **by** *auto*
moreover note *dom*
ultimately show *False* **by** (*simp add*: *infinite-super*)
qed

lemma *inf-img-fin-domE*:
assumes *finite* (*f'A*) **and** *infinite A*
obtains *y* **where** $y \in$ *f'A* **and** *infinite* (*f −' {y}*)
using *assms* **by** (*blast dest*: *inf-img-fin-dom*)

36.2 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.

definition
Inf-many :: (*'a* ⇒ *bool*) ⇒ *bool* (**binder** *INFM* *10*) **where**
Inf-many P = *infinite* {*x*. *P x*}

definition
Alm-all :: (*'a* ⇒ *bool*) ⇒ *bool* (**binder** *MOST* *10*) **where**
Alm-all P = (¬ (*INFM x*. ¬ *P x*))

notation (*xsymbols*)
Inf-many (**binder** ∃$_\infty$ *10*) **and**
Alm-all (**binder** ∀$_\infty$ *10*)

notation (*HTML* **output**)
Inf-many (**binder** ∃$_\infty$ *10*) **and**
Alm-all (**binder** ∀$_\infty$ *10*)

lemma *INF-EX*:
(∃$_\infty$*x*. *P x*) ⟹ (∃ *x*. *P x*)
unfolding *Inf-many-def*
proof (*rule ccontr*)
assume *inf*: *infinite* {*x*. *P x*}
assume ¬ *?thesis* **then have** {*x*. *P x*} = {} **by** *simp*
then have *finite* {*x*. *P x*} **by** *simp*
with *inf* **show** *False* **by** *simp*
qed

lemma *MOST-iff-finiteNeg*: (∀$_\infty$*x*. *P x*) = *finite* {*x*. ¬ *P x*}
by (*simp add*: *Alm-all-def Inf-many-def*)

lemma *ALL-MOST*: ∀ *x*. *P x* ⟹ ∀$_\infty$*x*. *P x*
by (*simp add*: *MOST-iff-finiteNeg*)

lemma *INF-mono*:
assumes *inf*: $\exists_\infty x.\ P\ x$ **and** *q*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\exists_\infty x.\ Q\ x$
proof −
from *inf* **have** *infinite* $\{x.\ P\ x\}$ **unfolding** *Inf-many-def* **.**
moreover from *q* **have** $\{x.\ P\ x\} \subseteq \{x.\ Q\ x\}$ **by** *auto*
ultimately show *?thesis*
by (*simp add*: *Inf-many-def infinite-super*)
qed

lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$
unfolding *Alm-all-def* **by** (*blast intro*: *INF-mono*)

lemma *INF-nat*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall\, m.\ \exists\, n.\ m<n \wedge P\ n)$
by (*simp add*: *Inf-many-def infinite-nat-iff-unbounded*)

lemma *INF-nat-le*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall\, m.\ \exists\, n.\ m\leq n \wedge P\ n)$
by (*simp add*: *Inf-many-def infinite-nat-iff-unbounded-le*)

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n::nat)) = (\exists\, m.\ \forall\, n.\ m<n \longrightarrow P\ n)$
by (*simp add*: *Alm-all-def INF-nat*)

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n::nat)) = (\exists\, m.\ \forall\, n.\ m\leq n \longrightarrow P\ n)$
by (*simp add*: *Alm-all-def INF-nat-le*)

36.3 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

consts
enumerate :: $'a::wellorder\ set \Rightarrow (nat \Rightarrow 'a::wellorder)$
primrec
enumerate-0: *enumerate S 0* $= (LEAST\ n.\ n \in S)$
enumerate-Suc: *enumerate S* $(Suc\ n) =$ *enumerate* $(S - \{LEAST\ n.\ n \in S\})\ n$

lemma *enumerate-Suc′*:
enumerate S $(Suc\ n) =$ *enumerate* $(S - \{$*enumerate S 0*$\})\ n$
by *simp*

lemma *enumerate-in-set*: *infinite S* $\Longrightarrow$ *enumerate S n* $: S$
apply (*induct n arbitrary*: *S*)
apply (*fastsimp intro*: *LeastI dest*!: *infinite-imp-nonempty*)
apply (*fastsimp iff*: *finite-Diff-singleton*)
done

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: *infinite S* $\Longrightarrow$ *enumerate S n* $<$ *enumerate S* $(Suc\ n)$
apply (*induct n arbitrary*: *S*)
apply (*rule order-le-neq-trans*)

apply (*simp add*: *enumerate-0 Least-le enumerate-in-set*)
apply (*simp only*: *enumerate-Suc'*)
apply (*subgoal-tac enumerate* (*S* − {*enumerate S 0*}) *0* : *S* − {*enumerate S 0*})
apply (*blast intro*: *sym*)
apply (*simp add*: *enumerate-in-set del*: *Diff-iff*)
apply (*simp add*: *enumerate-Suc'*)
done

lemma *enumerate-mono*: *m*<*n* ⟹ *infinite S* ⟹ *enumerate S m* < *enumerate S n*
apply (*erule less-Suc-induct*)
apply (*auto intro*: *enumerate-step*)
done

36.4 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

definition
atmost-one :: *'a set* ⇒ *bool* **where**
atmost-one S = (∀ *x y*. *x*∈*S* ∧ *y*∈*S* ⟶ *x*=*y*)

lemma *atmost-one-empty*: *S* = {} ⟹ *atmost-one S*
by (*simp add*: *atmost-one-def*)

lemma *atmost-one-singleton*: *S* = {*x*} ⟹ *atmost-one S*
by (*simp add*: *atmost-one-def*)

lemma *atmost-one-unique* [*elim*]: *atmost-one S* ⟹ *x* ∈ *S* ⟹ *y* ∈ *S* ⟹ *y* = *x*
by (*simp add*: *atmost-one-def*)

end

37 ListVector: 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*::*times*) ⇒ *'a list* ⇒ *'a list* (**infix** ∗*s* *70*)
where *x* ∗*s* *xs* ≡ *map* (*op* ∗ *x*) *xs*

lemma *scale1*[*simp*]: (*1*::'*a*::*monoid-mult*) $*_s$ *xs* = *xs*
by (*induct xs*) *simp-all*

37.1 + and −

fun *zipwith0* :: ('*a*::*zero* ⇒ '*b*::*zero* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list*
where
zipwith0 f [] [] = [] |
zipwith0 f (*x*#*xs*) (*y*#*ys*) = *f x y* # *zipwith0 f xs ys* |
zipwith0 f (*x*#*xs*) [] = *f x 0* # *zipwith0 f xs* [] |
zipwith0 f [] (*y*#*ys*) = *f 0 y* # *zipwith0 f* [] *ys*

instance *list* :: ({*zero*,*plus*})*plus*
list-add-def : *op* + ≡ *zipwith0* (*op* +) **..**

instance *list* :: ({*zero*,*uminus*})*uminus*
list-uminus-def: *uminus* ≡ *map uminus* **..**

instance *list* :: ({*zero*,*minus*})*minus*
list-diff-def: *op* − ≡ *zipwith0* (*op* −) **..**

lemma *zipwith0-Nil*[*simp*]: *zipwith0 f* [] *ys* = *map* (*f 0*) *ys*
by(*induct ys*) *simp-all*

lemma *list-add-Nil*[*simp*]: [] + *xs* = (*xs*::'*a*::*monoid-add list*)
by (*induct xs*) (*auto simp*:*list-add-def*)

lemma *list-add-Nil2*[*simp*]: *xs* + [] = (*xs*::'*a*::*monoid-add list*)
by (*induct xs*) (*auto simp*:*list-add-def*)

lemma *list-add-Cons*[*simp*]: (*x*#*xs*) + (*y*#*ys*) = (*x*+*y*)#(*xs*+*ys*)
by(*auto simp*:*list-add-def*)

lemma *list-diff-Nil*[*simp*]: [] − *xs* = −(*xs*::'*a*::*group-add list*)
by (*induct xs*) (*auto simp*:*list-diff-def list-uminus-def*)

lemma *list-diff-Nil2*[*simp*]: *xs* − [] = (*xs*::'*a*::*group-add list*)
by (*induct xs*) (*auto simp*:*list-diff-def*)

lemma *list-diff-Cons-Cons*[*simp*]: (*x*#*xs*) − (*y*#*ys*) = (*x*−*y*)#(*xs*−*ys*)
by (*induct xs*) (*auto simp*:*list-diff-def*)

lemma *list-uminus-Cons*[*simp*]: −(*x*#*xs*) = (−*x*)#(−*xs*)
by (*induct xs*) (*auto simp*:*list-uminus-def*)

lemma *self-list-diff*:
xs − *xs* = *replicate* (*length*(*xs*::'*a*::*group-add list*)) *0*
by(*induct xs*) *simp-all*

lemma *list-add-assoc*: **fixes** *xs* :: *'a*::*monoid-add list*
shows *(xs+ys)+zs = xs+(ys+zs)*
apply(*induct xs arbitrary*: *ys zs*)
apply *simp*
apply(*case-tac ys*)
apply(*simp*)
apply(*simp*)
apply(*case-tac zs*)
apply(*simp*)
apply(*simp add*:*add-assoc*)
done

37.2 Inner product

definition *iprod* :: *'a*::*ring list* ⇒ *'a list* ⇒ *'a* (⟨-,-⟩) **where**
⟨*xs*,*ys*⟩ = (∑ (*x*,*y*) ← *zip xs ys*. *x*∗*y*)

lemma *iprod-Nil*[*simp*]: ⟨[],*ys*⟩ = *0*
by(*simp add*:*iprod-def*)

lemma *iprod-Nil2*[*simp*]: ⟨*xs*,[]⟩ = *0*
by(*simp add*:*iprod-def*)

lemma *iprod-Cons*[*simp*]: ⟨*x*#*xs*,*y*#*ys*⟩ = *x*∗*y* + ⟨*xs*,*ys*⟩
by(*simp add*:*iprod-def*)

lemma *iprod0-if-coeffs0*: ∀ *c*∈*set cs*. *c* = *0* ⟹ ⟨*cs*,*xs*⟩ = *0*
apply(*induct cs arbitrary*:*xs*)
apply *simp*
apply(*case-tac xs*) **apply** *simp*
apply *auto*
done

lemma *iprod-uminus*[*simp*]: ⟨−*xs*,*ys*⟩ = −⟨*xs*,*ys*⟩
by(*simp add*: *iprod-def uminus-listsum-map o-def split-def map-zip-map list-uminus-def*)

lemma *iprod-left-add-distrib*: ⟨*xs* + *ys*,*zs*⟩ = ⟨*xs*,*zs*⟩ + ⟨*ys*,*zs*⟩
apply(*induct xs arbitrary*: *ys zs*)
apply (*simp add*: *o-def split-def*)
apply(*case-tac ys*)
apply *simp*
apply(*case-tac zs*)
apply (*simp*)
apply(*simp add*:*left-distrib*)
done

lemma *iprod-left-diff-distrib*: ⟨*xs* − *ys*, *zs*⟩ = ⟨*xs*,*zs*⟩ − ⟨*ys*,*zs*⟩
apply(*induct xs arbitrary*: *ys zs*)

apply (*simp add*: *o-def split-def*)
apply(*case-tac ys*)
apply *simp*
apply(*case-tac zs*)
apply (*simp*)
apply(*simp add*:*left-diff-distrib*)
done

lemma *iprod-assoc*: $\langle x *_s xs, ys\rangle = x * \langle xs,ys\rangle$
apply(*induct xs arbitrary*: *ys*)
apply *simp*
apply(*case-tac ys*)
apply (*simp*)
apply (*simp add*:*right-distrib mult-assoc*)
done

end

38 Multiset: Multisets

theory *Multiset*
imports *List*
begin

38.1 The type of multisets

typedef *'a multiset* = {*f*::*'a* => *nat*. *finite* {*x* . *f x* > *0*}}
proof
  show (λ*x*. *0*::*nat*) ∈ *?multiset* **by** *simp*
qed

lemmas *multiset-typedef* [*simp*] =
    Abs-multiset-inverse Rep-multiset-inverse Rep-multiset
  and [*simp*] = *Rep-multiset-inject* [*symmetric*]

definition
  Mempty :: *'a multiset* ({#}) **where**
  {#} = *Abs-multiset* (λ*a*. *0*)

definition
  single :: *'a* => *'a multiset* **where**
  single a = *Abs-multiset* (λ*b*. *if b* = *a then 1 else 0*)

declare
  Mempty-def[*code func del*] *single-def*[*code func del*]

definition
  count :: *'a multiset* => *'a* => *nat* **where**
  count = *Rep-multiset*

definition
MCollect :: *'a multiset => ('a => bool) => 'a multiset* **where**
MCollect M P = Abs-multiset (λx. if P x then Rep-multiset M x else 0)

abbreviation
Melem :: *'a => 'a multiset => bool* ((-/ :# -) [*50*, *51*] *50*) **where**
a :# M == 0 < count M a

notation (*xsymbols*)
Melem (**infix** ∈# *50*)

syntax
-MCollect :: *pttrn => 'a multiset => bool => 'a multiset* ((*1*{# - :# -./ -#}))
translations
{#*x* :# *M*. *P*#} == *CONST MCollect M* (λ*x*. *P*)

definition
set-of :: *'a multiset => 'a set* **where**
set-of M = {*x*. *x* :# *M*}

instantiation *multiset* :: (*type*) {*plus*, *minus*, *zero*, *size*}
begin

definition
union-def[*code func del*]:
M + N = Abs-multiset (λ*a*. *Rep-multiset M a + Rep-multiset N a*)

definition
diff-def: *M − N = Abs-multiset* (λ*a*. *Rep-multiset M a − Rep-multiset N a*)

definition
Zero-multiset-def [*simp*]: *0* = {#}

definition
size-def[*code func del*]: *size M = setsum* (*count M*) (*set-of M*)

instance ..

end

definition
multiset-inter :: *'a multiset ⇒ 'a multiset ⇒ 'a multiset* (**infixl** #∩ *70*) **where**
multiset-inter A B = A − (A − B)

Multiset Enumeration

syntax
-multiset :: *args => 'a multiset* ({#(-)#})
translations

$\{\#x,\ xs\#\} == \{\#x\#\} + \{\#xs\#\}$
$\{\#x\#\} == CONST\ single\ x$

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*: $(\lambda a.\ 0) \in multiset$
by (*simp add*: *multiset-def*)

lemma *only1-in-multiset*: $(\lambda b.\ if\ b = a\ then\ 1\ else\ 0) \in multiset$
by (*simp add*: *multiset-def*)

lemma *union-preserves-multiset*:
$M \in multiset ==> N \in multiset ==> (\lambda a.\ M\ a + N\ a) \in multiset$
apply (*simp add*: *multiset-def*)
apply (*drule* (*1*) *finite-UnI*)
apply (*simp del*: *finite-Un add*: *Un-def*)
done

lemma *diff-preserves-multiset*:
$M \in multiset ==> (\lambda a.\ M\ a - N\ a) \in multiset$
apply (*simp add*: *multiset-def*)
apply (*rule finite-subset*)
apply *auto*
done

lemma *MCollect-preserves-multiset*:
$M \in multiset ==> (\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
apply (*simp add*: *multiset-def*)
apply (*rule finite-subset*, *auto*)
done

lemmas *in-multiset* = *const0-in-multiset only1-in-multiset*
union-preserves-multiset diff-preserves-multiset MCollect-preserves-multiset

38.2 Algebraic properties

38.2.1 Union

lemma *union-empty* [*simp*]: $M + \{\#\} = M \wedge \{\#\} + M = M$
by (*simp add*: *union-def Mempty-def in-multiset*)

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$
by (*simp add*: *union-def add-ac in-multiset*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
by (*simp add*: *union-def add-ac in-multiset*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
proof −
have $M + (N + K) = (N + K) + M$ **by** (*rule union-commute*)

also have $\ldots = N + (K + M)$ **by** (*rule union-assoc*)
also have $K + M = M + K$ **by** (*rule union-commute*)
finally show *?thesis* **.**
qed

lemmas *union-ac* = *union-assoc union-commute union-lcomm*

instance *multiset* :: (*type*) *comm-monoid-add*
proof
fix *a b c* :: $'a$ *multiset*
show $(a + b) + c = a + (b + c)$ **by** (*rule union-assoc*)
show $a + b = b + a$ **by** (*rule union-commute*)
show $0 + a = a$ **by** *simp*
qed

38.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
by (*simp add*: *Mempty-def diff-def in-multiset*)

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
by (*simp add*: *union-def diff-def in-multiset*)

lemma *diff-cancel*: $A - A = \{\#\}$
by (*simp add*: *diff-def Mempty-def*)

38.2.3 Count of elements

lemma *count-empty* [*simp*]: *count* $\{\#\}$ $a = 0$
by (*simp add*: *count-def Mempty-def in-multiset*)

lemma *count-single* [*simp*]: *count* $\{\#b\#\}$ $a = (\textit{if } b = a \textit{ then } 1 \textit{ else } 0)$
by (*simp add*: *count-def single-def in-multiset*)

lemma *count-union* [*simp*]: *count* $(M + N)$ $a =$ *count* M $a +$ *count* N a
by (*simp add*: *count-def union-def in-multiset*)

lemma *count-diff* [*simp*]: *count* $(M - N)$ $a =$ *count* M $a -$ *count* N a
by (*simp add*: *count-def diff-def in-multiset*)

lemma *count-MCollect* [*simp*]:
count $\{\#\ x{:}\#M.\ P\ x\ \#\}$ $a = (\textit{if } P\ a \textit{ then count } M\ a \textit{ else } 0)$
by (*simp add*: *count-def MCollect-def in-multiset*)

38.2.4 Set of elements

lemma *set-of-empty* [*simp*]: *set-of* $\{\#\} = \{\}$
by (*simp add*: *set-of-def*)

lemma *set-of-single* [*simp*]: *set-of* $\{\#b\#\} = \{b\}$

by (*simp add*: *set-of-def*)

lemma *set-of-union* [*simp*]: *set-of* (*M* + *N*) = *set-of M* ∪ *set-of N*
by (*auto simp add*: *set-of-def*)

lemma *set-of-eq-empty-iff* [*simp*]: (*set-of M* = {}) = (*M* = {#})
by (*auto simp*: *set-of-def Mempty-def in-multiset count-def expand-fun-eq* [**where** *f*=*Rep-multiset M*])

lemma *mem-set-of-iff* [*simp*]: (*x* ∈ *set-of M*) = (*x* :# *M*)
by (*auto simp add*: *set-of-def*)

lemma *set-of-MCollect* [*simp*]: *set-of* {# *x*:#*M*. *P x* #} = *set-of M* ∩ {*x*. *P x*}
by (*auto simp add*: *set-of-def*)

38.2.5 Size

lemma *size-empty* [*simp*,*code func*]: *size* {#} = *0*
by (*simp add*: *size-def*)

lemma *size-single* [*simp*,*code func*]: *size* {#*b*#} = *1*
by (*simp add*: *size-def*)

lemma *finite-set-of* [*iff*]: *finite* (*set-of M*)
using *Rep-multiset* [*of M*] **by** (*simp add*: *multiset-def set-of-def count-def*)

lemma *setsum-count-Int*:
finite A ==> *setsum* (*count N*) (*A* ∩ *set-of N*) = *setsum* (*count N*) *A*
apply (*induct rule*: *finite-induct*)
apply *simp*
apply (*simp add*: *Int-insert-left set-of-def*)
done

lemma *size-union*[*simp*,*code func*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
apply (*unfold size-def*)
apply (*subgoal-tac count* (*M* + *N*) = (λ*a*. *count M a* + *count N a*))
prefer *2*
apply (*rule ext*, *simp*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-Un-nat setsum-addf setsum-count-Int*)
apply (*subst Int-commute*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-count-Int*)
done

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
apply (*unfold size-def Mempty-def count-def*, *auto simp*: *in-multiset*)
apply (*simp add*: *set-of-def count-def in-multiset expand-fun-eq*)
done

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)

by (*metis gr0I gr-implies-not0 size-empty size-eq-0-iff-empty*)

lemma *size-eq-Suc-imp-elem*: *size M = Suc n ==> ∃ a. a :# M*
apply (*unfold size-def*)
apply (*drule setsum-SucD*)
apply *auto*
done

38.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: *(M = N) = (∀ a. count M a = count N a)*
by (*simp add*: *count-def expand-fun-eq*)

lemma *single-not-empty* [*simp*]: *{#a#} ≠ {#} ∧ {#} ≠ {#a#}*
by (*simp add*: *single-def Mempty-def in-multiset expand-fun-eq*)

lemma *single-eq-single* [*simp*]: *({#a#} = {#b#}) = (a = b)*
by (*auto simp add*: *single-def in-multiset expand-fun-eq*)

lemma *union-eq-empty* [*iff*]: *(M + N = {#}) = (M = {#} ∧ N = {#})*
by (*auto simp add*: *union-def Mempty-def in-multiset expand-fun-eq*)

lemma *empty-eq-union* [*iff*]: *({#} = M + N) = (M = {#} ∧ N = {#})*
by (*auto simp add*: *union-def Mempty-def in-multiset expand-fun-eq*)

lemma *union-right-cancel* [*simp*]: *(M + K = N + K) = (M = (N::'a multiset))*
by (*simp add*: *union-def in-multiset expand-fun-eq*)

lemma *union-left-cancel* [*simp*]: *(K + M = K + N) = (M = (N::'a multiset))*
by (*simp add*: *union-def in-multiset expand-fun-eq*)

lemma *union-is-single*:
(M + N = {#a#}) = (M = {#a#} ∧ N={#} ∨ M = {#} ∧ N = {#a#})
apply (*simp add*: *Mempty-def single-def union-def in-multiset add-is-1 expand-fun-eq*)
apply *blast*
done

lemma *single-is-union*:
({#a#} = M + N) ⟷ ({#a#} = M ∧ N = {#} ∨ M = {#} ∧ {#a#} = N)
apply (*unfold Mempty-def single-def union-def*)
apply (*simp add*: *add-is-1 one-is-add in-multiset expand-fun-eq*)
apply (*blast dest*: *sym*)
done

lemma *add-eq-conv-diff*:
(M + {#a#} = N + {#b#}) =
(M = N ∧ a = b ∨ M = N − {#a#} + {#b#} ∧ N = M − {#b#} + {#a#})

using [[*simproc del*: *neq*]]
apply (*unfold single-def union-def diff-def*)
apply (*simp* (*no-asm*) *add*: *in-multiset expand-fun-eq*)
apply (*rule conjI*, *force*, *safe*, *simp-all*)
apply (*simp add*: *eq-sym-conv*)
done

declare *Rep-multiset-inject* [*symmetric*, *simp del*]

instance *multiset* :: (*type*) *cancel-ab-semigroup-add*
proof
  fix *a b c* :: *'a multiset*
  show $a + b = a + c \Longrightarrow b = c$ **by** *simp*
qed

lemma *insert-DiffM*:
  $x \in\# M \Longrightarrow \{\#x\#\} + (M - \{\#x\#\}) = M$
by (*clarsimp simp*: *multiset-eq-conv-count-eq*)

lemma *insert-DiffM2*[*simp*]:
  $x \in\# M \Longrightarrow M - \{\#x\#\} + \{\#x\#\} = M$
by (*clarsimp simp*: *multiset-eq-conv-count-eq*)

lemma *multi-union-self-other-eq*:
  (*A*::*'a multiset*) + *X* = *A* + *Y* $\Longrightarrow$ *X* = *Y*
by (*induct A arbitrary*: *X Y*) *auto*

lemma *multi-self-add-other-not-self*[*simp*]: (*A* = *A* + {#*x*#}) = *False*
by (*metis single-not-empty union-empty union-left-cancel*)

lemma *insert-noteq-member*:
  assumes *BC*: *B* + {#*b*#} = *C* + {#*c*#}
   and *bnotc*: *b* ≠ *c*
  shows *c* ∈# *B*
proof −
  have *c* ∈# *C* + {#*c*#} **by** *simp*
  have *nc*: ¬ *c* ∈# {#*b*#} **using** *bnotc* **by** *simp*
  then have *c* ∈# *B* + {#*b*#} **using** *BC* **by** *simp*
  then show *c* ∈# *B* **using** *nc* **by** *simp*
qed

lemma *add-eq-conv-ex*:
  (*M* + {#*a*#} = *N* + {#*b*#}) =
    (*M* = *N* ∧ *a* = *b* ∨ (∃*K*. *M* = *K* + {#*b*#} ∧ *N* = *K* + {#*a*#}))
by (*auto simp add*: *add-eq-conv-diff*)

lemma *empty-multiset-count*:

$(\forall x.\ count\ A\ x = 0) = (A = \{\#\})$
by (*metis count-empty multiset-eq-conv-count-eq*)

38.2.7 Intersection

lemma *multiset-inter-count*:
$count\ (A\ \#\cap\ B)\ x = min\ (count\ A\ x)\ (count\ B\ x)$
by (*simp add*: *multiset-inter-def min-def*)

lemma *multiset-inter-commute*: $A\ \#\cap\ B = B\ \#\cap\ A$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count*
min-max.inf-commute)

lemma *multiset-inter-assoc*: $A\ \#\cap\ (B\ \#\cap\ C) = A\ \#\cap\ B\ \#\cap\ C$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count*
min-max.inf-assoc)

lemma *multiset-inter-left-commute*: $A\ \#\cap\ (B\ \#\cap\ C) = B\ \#\cap\ (A\ \#\cap\ C)$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def*)

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *multiset-inter-single*: $a \neq b \Longrightarrow \{\#a\#\}\ \#\cap\ \{\#b\#\} = \{\#\}$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count*)

lemma *multiset-union-diff-commute*: $B\ \#\cap\ C = \{\#\} \Longrightarrow A + B - C = A - C + B$
apply (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def*
split: *split-if-asm*)
apply *clarsimp*
apply (*erule-tac* $x = a$ **in** *allE*)
apply *auto*
done

38.2.8 Comprehension (filter)

lemma *MCollect-empty*[*simp*, *code func*]: $MCollect\ \{\#\}\ P = \{\#\}$
by (*simp add*: *MCollect-def Mempty-def Abs-multiset-inject*
in-multiset expand-fun-eq)

lemma *MCollect-single*[*simp*, *code func*]:
$MCollect\ \{\#x\#\}\ P = (if\ P\ x\ then\ \{\#x\#\}\ else\ \{\#\})$
by (*simp add*: *MCollect-def Mempty-def single-def Abs-multiset-inject*
in-multiset expand-fun-eq)

lemma *MCollect-union*[*simp*, *code func*]:
$MCollect\ (M+N)\ f = MCollect\ M\ f\ +\ MCollect\ N\ f$

by (*simp add*: *MCollect-def union-def Abs-multiset-inject*
in-multiset expand-fun-eq)

38.3 Induction and case splits

lemma *setsum-decr*:
finite F ==> (0::nat) < f a ==>
setsum (f (a := f a − 1)) F = (if a∈F then setsum f F − 1 else setsum f F)
apply (*induct rule*: *finite-induct*)
apply *auto*
apply (*drule-tac a = a* **in** *mk-disjoint-insert*, *auto*)
done

lemma *rep-multiset-induct-aux*:
assumes *1*: *P* (λ*a*. (*0*::*nat*))
and *2*: !!*f b*. *f* ∈ *multiset ==> P f ==> P* (*f* (*b* := *f b* + *1*))
shows ∀*f*. *f* ∈ *multiset −−> setsum f* {*x*. *f x* ≠ *0*} = *n −−> P f*
apply (*unfold multiset-def*)
apply (*induct-tac n*, *simp*, *clarify*)
apply (*subgoal-tac f* = (λ*a.0*))
apply *simp*
apply (*rule 1*)
apply (*rule ext*, *force*, *clarify*)
apply (*frule setsum-SucD*, *clarify*)
apply (*rename-tac a*)
apply (*subgoal-tac finite* {*x*. (*f* (*a* := *f a* − *1*)) *x* > *0*})
prefer *2*
apply (*rule finite-subset*)
prefer *2*
apply *assumption*
apply *simp*
apply *blast*
apply (*subgoal-tac f* = (*f* (*a* := *f a* − *1*))(*a* := (*f* (*a* := *f a* − *1*)) *a* + *1*))
prefer *2*
apply (*rule ext*)
apply (*simp* (*no-asm-simp*))
apply (*erule ssubst*, *rule 2* [*unfolded multiset-def*], *blast*)
apply (*erule allE*, *erule impE*, *erule-tac* [*2*] *mp*, *blast*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-decr del*: *fun-upd-apply One-nat-def*)
apply (*subgoal-tac* {*x*. *x* ≠ *a −−> f x* ≠ *0*} = {*x*. *f x* ≠ *0*})
prefer *2*
apply *blast*
apply (*subgoal-tac* {*x*. *x* ≠ *a* ∧ *f x* ≠ *0*} = {*x*. *f x* ≠ *0*} − {*a*})
prefer *2*
apply *blast*
apply (*simp add*: *le-imp-diff-is-add setsum-diff1-nat cong*: *conj-cong*)
done

theorem *rep-multiset-induct*:

```
  f ∈ multiset ==> P (λa. 0) ==>
    (!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))) ==> P f
using rep-multiset-induct-aux by blast

theorem multiset-induct [case-names empty add, induct type: multiset]:
assumes empty: P {#}
  and add: !!M x. P M ==> P (M + {#x#})
shows P M
proof −
  note defns = union-def single-def Mempty-def
  show ?thesis
    apply (rule Rep-multiset-inverse [THEN subst])
    apply (rule Rep-multiset [THEN rep-multiset-induct])
     apply (rule empty [unfolded defns])
    apply (subgoal-tac f(b := f b + 1) = (λa. f a + (if a=b then 1 else 0)))
     prefer 2
     apply (simp add: expand-fun-eq)
    apply (erule ssubst)
    apply (erule Abs-multiset-inverse [THEN subst])
    apply (drule add [unfolded defns, simplified])
    apply(simp add:in-multiset)
    done
qed

lemma multi-nonempty-split: M ≠ {#} ⟹ ∃A a. M = A + {#a#}
by (induct M) auto

lemma multiset-cases [cases type, case-names empty add]:
assumes em:  M = {#} ⟹ P
assumes add: ⋀N x. M = N + {#x#} ⟹ P
shows P
proof (cases M = {#})
  assume M = {#} then show ?thesis using em by simp
next
  assume M ≠ {#}
  then obtain M' m where M = M' + {#m#}
    by (blast dest: multi-nonempty-split)
  then show ?thesis using add by simp
qed

lemma multi-member-split: x ∈# M ⟹ ∃A. M = A + {#x#}
apply (cases M)
 apply simp
apply (rule-tac x=M − {#x#} in exI, simp)
done

lemma multiset-partition: M = {# x:#M. P x #} + {# x:#M. ¬ P x #}
apply (subst multiset-eq-conv-count-eq)
apply auto
```

done

declare *multiset-typedef* [*simp del*]

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
by (*cases* $B = \{\#\}$) (*auto dest*: *multi-member-split*)

38.4 Orderings

38.4.1 Well-foundedness

definition
mult1 :: $('a \times 'a)$ *set* => $('a$ *multiset* $\times$ $'a$ *multiset*$)$ *set* **where**
mult1 r =
$\{(N, M).\ \exists a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge$
$(\forall b.\ b :\# K \longrightarrow (b, a) \in r)\}$

definition
mult :: $('a \times 'a)$ *set* => $('a$ *multiset* $\times$ $'a$ *multiset*$)$ *set* **where**
mult $r = (mult1\ r)^+$

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin mult1\ r$
by (*simp add*: *mult1-def*)

lemma *less-add*: $(N, M0 + \{\#a\#\}) \in mult1\ r$ ==>
$(\exists M.\ (M, M0) \in mult1\ r \wedge N = M + \{\#a\#\}) \vee$
$(\exists K.\ (\forall b.\ b :\# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$
(**is** - $\Longrightarrow$ *?case1* $(mult1\ r) \vee$ *?case2*)
proof (*unfold mult1-def*)
let *?r* $= \lambda K\ a.\ \forall b.\ b :\# K \longrightarrow (b, a) \in r$
let *?R* $= \lambda N\ M.\ \exists a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge$ *?r* $K\ a$
let *?case1* = *?case1* $\{(N, M).$ *?R* $N\ M\}$

assume $(N, M0 + \{\#a\#\}) \in \{(N, M).$ *?R* $N\ M\}$
then have $\exists a'\ M0'\ K.$
$M0 + \{\#a\#\} = M0' + \{\#a'\#\} \wedge N = M0' + K \wedge$ *?r* $K\ a'$ **by** *simp*
then show *?case1* $\vee$ *?case2*
proof (*elim exE conjE*)
fix $a'\ M0'\ K$
assume *N*: $N = M0' + K$ **and** *r*: *?r* $K\ a'$
assume $M0 + \{\#a\#\} = M0' + \{\#a'\#\}$
then have $M0 = M0' \wedge a = a' \vee$
$(\exists K'.\ M0 = K' + \{\#a'\#\} \wedge M0' = K' + \{\#a\#\})$
by (*simp only*: *add-eq-conv-ex*)
then show *?thesis*
proof (*elim disjE conjE exE*)
assume $M0 = M0'\ a = a'$
with $N\ r$ **have** *?r* $K\ a \wedge N = M0 + K$ **by** *simp*
then have *?case2* **.. then show** *?thesis* **..**
next

```
      fix K'
      assume M0' = K' + {#a#}
      with N have n: N = K' + K + {#a#} by (simp add: union-ac)

      assume M0 = K' + {#a'#}
      with r have ?R (K' + K) M0 by blast
      with n have ?case1 by simp then show ?thesis ..
    qed
  qed
qed

lemma all-accessible: wf r ==> ∀ M. M ∈ acc (mult1 r)
proof
  let ?R = mult1 r
  let ?W = acc ?R
  {
    fix M M0 a
    assume M0: M0 ∈ ?W
      and wf-hyp: !!b. (b, a) ∈ r ==> (∀ M ∈ ?W. M + {#b#} ∈ ?W)
      and acc-hyp: ∀ M. (M, M0) ∈ ?R --> M + {#a#} ∈ ?W
    have M0 + {#a#} ∈ ?W
    proof (rule accI [of M0 + {#a#}])
      fix N
      assume (N, M0 + {#a#}) ∈ ?R
      then have ((∃ M. (M, M0) ∈ ?R ∧ N = M + {#a#}) ∨
          (∃ K. (∀ b. b :# K --> (b, a) ∈ r) ∧ N = M0 + K))
        by (rule less-add)
      then show N ∈ ?W
      proof (elim exE disjE conjE)
        fix M assume (M, M0) ∈ ?R and N: N = M + {#a#}
        from acc-hyp have (M, M0) ∈ ?R --> M + {#a#} ∈ ?W ..
        from this and ‹(M, M0) ∈ ?R› have M + {#a#} ∈ ?W ..
        then show N ∈ ?W by (simp only: N)
      next
        fix K
        assume N: N = M0 + K
        assume ∀ b. b :# K --> (b, a) ∈ r
        then have M0 + K ∈ ?W
        proof (induct K)
          case empty
          from M0 show M0 + {#} ∈ ?W by simp
        next
          case (add K x)
          from add.prems have (x, a) ∈ r by simp
          with wf-hyp have ∀ M ∈ ?W. M + {#x#} ∈ ?W by blast
          moreover from add have M0 + K ∈ ?W by simp
          ultimately have (M0 + K) + {#x#} ∈ ?W ..
          then show M0 + (K + {#x#}) ∈ ?W by (simp only: union-assoc)
        qed
```

```
      then show N ∈ ?W by (simp only: N)
    qed
  qed
 } note tedious-reasoning = this

 assume wf: wf r
 fix M
 show M ∈ ?W
 proof (induct M)
  show {#} ∈ ?W
  proof (rule accI)
   fix b assume (b, {#}) ∈ ?R
   with not-less-empty show b ∈ ?W by contradiction
  qed

  fix M a assume M ∈ ?W
  from wf have ∀ M ∈ ?W. M + {#a#} ∈ ?W
  proof induct
   fix a
   assume r: !!b. (b, a) ∈ r ==> (∀ M ∈ ?W. M + {#b#} ∈ ?W)
   show ∀ M ∈ ?W. M + {#a#} ∈ ?W
   proof
    fix M assume M ∈ ?W
    then show M + {#a#} ∈ ?W
     by (rule acc-induct) (rule tedious-reasoning [OF - r])
   qed
  qed
  from this and ‹M ∈ ?W› show M + {#a#} ∈ ?W ..
 qed
qed

theorem wf-mult1: wf r ==> wf (mult1 r)
by (rule acc-wfI) (rule all-accessible)

theorem wf-mult: wf r ==> wf (mult r)
unfolding mult-def by (rule wf-trancl) (rule wf-mult1)
```

38.4.2 Closure-free presentation

```
lemma diff-union-single-conv: a :# J ==> I + J - {#a#} = I + (J - {#a#})
by (simp add: multiset-eq-conv-count-eq)
```

One direction.

```
lemma mult-implies-one-step:
  trans r ==> (M, N) ∈ mult r ==>
   ∃ I J K. N = I + J ∧ M = I + K ∧ J ≠ {#} ∧
   (∀ k ∈ set-of K. ∃ j ∈ set-of J. (k, j) ∈ r)
apply (unfold mult-def mult1-def set-of-def)
apply (erule converse-trancl-induct, clarify)
 apply (rule-tac x = M0 in exI, simp, clarify)
```

apply (*case-tac a* :# *K*)
 apply (*rule-tac x* = *I* **in** *exI*)
 apply (*simp* (*no-asm*))
 apply (*rule-tac x* = (*K* − {#*a*#}) + *Ka* **in** *exI*)
 apply (*simp* (*no-asm-simp*) *add*: *union-assoc* [*symmetric*])
 apply (*drule-tac f* = λ*M*. *M* − {#*a*#} **in** *arg-cong*)
 apply (*simp add*: *diff-union-single-conv*)
 apply (*simp* (*no-asm-use*) *add*: *trans-def*)
 apply *blast*
apply (*subgoal-tac a* :# *I*)
 apply (*rule-tac x* = *I* − {#*a*#} **in** *exI*)
 apply (*rule-tac x* = *J* + {#*a*#} **in** *exI*)
 apply (*rule-tac x* = *K* + *Ka* **in** *exI*)
 apply (*rule conjI*)
  apply (*simp add*: *multiset-eq-conv-count-eq split*: *nat-diff-split*)
 apply (*rule conjI*)
  apply (*drule-tac f* = λ*M*. *M* − {#*a*#} **in** *arg-cong*, *simp*)
  apply (*simp add*: *multiset-eq-conv-count-eq split*: *nat-diff-split*)
 apply (*simp* (*no-asm-use*) *add*: *trans-def*)
 apply *blast*
apply (*subgoal-tac a* :# (*M0* + {#*a*#}))
 apply *simp*
apply (*simp* (*no-asm*))
done

lemma *elem-imp-eq-diff-union*: *a* :# *M* ==> *M* = *M* − {#*a*#} + {#*a*#}
by (*simp add*: *multiset-eq-conv-count-eq*)

lemma *size-eq-Suc-imp-eq-union*: *size M* = *Suc n* ==> ∃ *a N*. *M* = *N* + {#*a*#}
apply (*erule size-eq-Suc-imp-elem* [*THEN exE*])
apply (*drule elem-imp-eq-diff-union*, *auto*)
done

lemma *one-step-implies-mult-aux*:
  trans r ==>
    ∀ *I J K*. (*size J* = *n* ∧ *J* ≠ {#} ∧ (∀ *k* ∈ *set-of K*. ∃ *j* ∈ *set-of J*. (*k*, *j*) ∈ *r*))
      −−> (*I* + *K*, *I* + *J*) ∈ *mult r*
apply (*induct-tac n*, *auto*)
apply (*frule size-eq-Suc-imp-eq-union*, *clarify*)
apply (*rename-tac J'*, *simp*)
apply (*erule notE*, *auto*)
apply (*case-tac J'* = {#})
 apply (*simp add*: *mult-def*)
 apply (*rule r-into-trancl*)
 apply (*simp add*: *mult1-def set-of-def*, *blast*)

    Now we know *J'* ≠ {#}.

apply (*cut-tac M* = *K* **and** *P* = λ*x*. (*x*, *a*) ∈ *r* **in** *multiset-partition*)
apply (*erule-tac P* = ∀ *k* ∈ *set-of K*. *?P k* **in** *rev-mp*)
apply (*erule ssubst*)

apply (*simp add*: *Ball-def*, *auto*)
apply (*subgoal-tac*
  ((*I* + {# *x* :# *K*. (*x*, *a*) ∈ *r* #}) + {# *x* :# *K*. (*x*, *a*) ∉ *r* #},
    (*I* + {# *x* :# *K*. (*x*, *a*) ∈ *r* #}) + *J'*) ∈ *mult r*)
 prefer *2*
 apply *force*
apply (*simp* (*no-asm-use*) *add*: *union-assoc* [*symmetric*] *mult-def*)
apply (*erule trancl-trans*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def set-of-def*)
apply (*rule-tac x* = *a* **in** *exI*)
apply (*rule-tac x* = *I* + *J'* **in** *exI*)
apply (*simp add*: *union-ac*)
done

lemma *one-step-implies-mult*:
  trans r ==> *J* ≠ {#} ==> ∀ *k* ∈ *set-of K*. ∃ *j* ∈ *set-of J*. (*k*, *j*) ∈ *r*
    ==> (*I* + *K*, *I* + *J*) ∈ *mult r*
using *one-step-implies-mult-aux* **by** *blast*

38.4.3 Partial-order properties

instantiation *multiset* :: (*order*) *order*
begin

definition
  less-multiset-def: *M'* < *M* ⟷ (*M'*, *M*) ∈ *mult* {(*x'*, *x*). *x'* < *x*}

definition
  le-multiset-def: *M'* <= *M* ⟷ *M'* = *M* ∨ *M'* < (*M*::'*a multiset*)

lemma *trans-base-order*: *trans* {(*x'*, *x*). *x'* < (*x*::'*a*::*order*)}
unfolding *trans-def* **by** (*blast intro*: *order-less-trans*)

Irreflexivity.

lemma *mult-irrefl-aux*:
  finite A ==> (∀ *x* ∈ *A*. ∃ *y* ∈ *A*. *x* < (*y*::'*a*::*order*)) ⟹ *A* = {}
by (*induct rule*: *finite-induct*) (*auto intro*: *order-less-trans*)

lemma *mult-less-not-refl*: ¬ *M* < (*M*::'*a*::*order multiset*)
apply (*unfold less-multiset-def*, *auto*)
apply (*drule trans-base-order* [*THEN mult-implies-one-step*], *auto*)
apply (*drule finite-set-of* [*THEN mult-irrefl-aux* [*rule-format* (*no-asm*)]])
apply (*simp add*: *set-of-eq-empty-iff*)
done

lemma *mult-less-irrefl* [*elim*!]: *M* < (*M*::'*a*::*order multiset*) ==> *R*
using *insert mult-less-not-refl* **by** *fast*

Transitivity.

theorem *mult-less-trans*: $K < M \Longrightarrow M < N \Longrightarrow K < (N::'a::order\ multiset)$
unfolding *less-multiset-def mult-def* **by** (*blast intro*: *trancl-trans*)

Asymmetry.

theorem *mult-less-not-sym*: $M < N \Longrightarrow \neg\ N < (M::'a::order\ multiset)$
apply *auto*
apply (*rule mult-less-not-refl* [*THEN notE*])
apply (*erule mult-less-trans*, *assumption*)
done

theorem *mult-less-asym*:
$M < N \Longrightarrow (\neg\ P \Longrightarrow N < (M::'a::order\ multiset)) \Longrightarrow P$
using *mult-less-not-sym* **by** *blast*

theorem *mult-le-refl* [*iff*]: $M <= (M::'a::order\ multiset)$
unfolding *le-multiset-def* **by** *auto*

Anti-symmetry.

theorem *mult-le-antisym*:
$M <= N \Longrightarrow N <= M \Longrightarrow M = (N::'a::order\ multiset)$
unfolding *le-multiset-def* **by** (*blast dest*: *mult-less-not-sym*)

Transitivity.

theorem *mult-le-trans*:
$K <= M \Longrightarrow M <= N \Longrightarrow K <= (N::'a::order\ multiset)$
unfolding *le-multiset-def* **by** (*blast intro*: *mult-less-trans*)

theorem *mult-less-le*: $(M < N) = (M <= N \wedge M \neq (N::'a::order\ multiset))$
unfolding *le-multiset-def* **by** *auto*

instance
apply *intro-classes*
apply (*rule mult-less-le*)
apply (*rule mult-le-refl*)
apply (*erule mult-le-trans*, *assumption*)
apply (*erule mult-le-antisym*, *assumption*)
done

end

38.4.4 Monotonicity of multiset union

lemma *mult1-union*:
$(B, D) \in mult1\ r \Longrightarrow trans\ r \Longrightarrow (C + B, C + D) \in mult1\ r$
apply (*unfold mult1-def*)
apply *auto*
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = C + M0$ **in** *exI*)
apply (*simp add*: *union-assoc*)

done

lemma *union-less-mono2*: $B < D ==> C + B < C + (D::'a::order\ multiset)$
apply (*unfold less-multiset-def mult-def*)
apply (*erule trancl-induct*)
apply (*blast intro*: *mult1-union transI order-less-trans r-into-trancl*)
apply (*blast intro*: *mult1-union transI order-less-trans r-into-trancl trancl-trans*)
done

lemma *union-less-mono1*: $B < D ==> B + C < D + (C::'a::order\ multiset)$
apply (*subst union-commute* [*of B C*])
apply (*subst union-commute* [*of D C*])
apply (*erule union-less-mono2*)
done

lemma *union-less-mono*:
$A < C ==> B < D ==> A + B < C + (D::'a::order\ multiset)$
by (*blast intro*!: *union-less-mono1 union-less-mono2 mult-less-trans*)

lemma *union-le-mono*:
$A <= C ==> B <= D ==> A + B <= C + (D::'a::order\ multiset)$
unfolding *le-multiset-def*
by (*blast intro*: *union-less-mono union-less-mono1 union-less-mono2*)

lemma *empty-leI* [*iff*]: $\{\#\} <= (M::'a::order\ multiset)$
apply (*unfold le-multiset-def less-multiset-def*)
apply (*case-tac* $M = \{\#\}$)
prefer *2*
apply (*subgoal-tac* $(\{\#\} + \{\#\}, \{\#\} + M) \in mult\ (Collect\ (split\ op <)))$
prefer *2*
apply (*rule one-step-implies-mult*)
apply (*simp only*: *trans-def*)
apply *auto*
done

lemma *union-upper1*: $A <= A + (B::'a::order\ multiset)$
proof −
have $A + \{\#\} <= A + B$ **by** (*blast intro*: *union-le-mono*)
then show *?thesis* **by** *simp*
qed

lemma *union-upper2*: $B <= A + (B::'a::order\ multiset)$
by (*subst union-commute*) (*rule union-upper1*)

instance *multiset* :: (*order*) *pordered-ab-semigroup-add*
apply *intro-classes*
apply (*erule union-le-mono*[*OF mult-le-refl*])
done

38.5 Link with lists

primrec *multiset-of* :: *'a list* ⇒ *'a multiset* **where**
multiset-of [] = {#} |
multiset-of (*a* # *x*) = *multiset-of x* + {# *a* #}

lemma *multiset-of-zero-iff*[*simp*]: (*multiset-of x* = {#}) = (*x* = [])
by (*induct x*) *auto*

lemma *multiset-of-zero-iff-right*[*simp*]: ({#} = *multiset-of x*) = (*x* = [])
by (*induct x*) *auto*

lemma *set-of-multiset-of*[*simp*]: *set-of*(*multiset-of x*) = *set x*
by (*induct x*) *auto*

lemma *mem-set-multiset-eq*: *x* ∈ *set xs* = (*x* :# *multiset-of xs*)
by (*induct xs*) *auto*

lemma *multiset-of-append* [*simp*]:
multiset-of (*xs* @ *ys*) = *multiset-of xs* + *multiset-of ys*
by (*induct xs arbitrary*: *ys*) (*auto simp*: *union-ac*)

lemma *surj-multiset-of*: *surj multiset-of*
apply (*unfold surj-def*)
apply (*rule allI*)
apply (*rule-tac M* = *y* **in** *multiset-induct*)
apply *auto*
apply (*rule-tac x* = *x* # *xa* **in** *exI*)
apply *auto*
done

lemma *set-count-greater-0*: *set x* = {*a*. *count* (*multiset-of x*) *a* > *0*}
by (*induct x*) *auto*

lemma *distinct-count-atmost-1*:
distinct x = (! *a*. *count* (*multiset-of x*) *a* = (*if a* ∈ *set x then 1 else 0*))
apply (*induct x*, *simp*, *rule iffI*, *simp-all*)
apply (*rule conjI*)
apply (*simp-all add*: *set-of-multiset-of* [*THEN sym*] *del*: *set-of-multiset-of*)
apply (*erule-tac x* = *a* **in** *allE*, *simp*, *clarify*)
apply (*erule-tac x* = *aa* **in** *allE*, *simp*)
done

lemma *multiset-of-eq-setD*:
multiset-of xs = *multiset-of ys* ⟹ *set xs* = *set ys*
by (*rule*) (*auto simp add*:*multiset-eq-conv-count-eq set-count-greater-0*)

lemma *set-eq-iff-multiset-of-eq-distinct*:
distinct x ⟹ *distinct y* ⟹
(*set x* = *set y*) = (*multiset-of x* = *multiset-of y*)

by (*auto simp*: *multiset-eq-conv-count-eq distinct-count-atmost-1*)

lemma *set-eq-iff-multiset-of-remdups-eq*:
  (*set x* = *set y*) = (*multiset-of* (*remdups x*) = *multiset-of* (*remdups y*))
apply (*rule iffI*)
apply (*simp add*: *set-eq-iff-multiset-of-eq-distinct*[*THEN iffD1*])
apply (*drule distinct-remdups* [*THEN distinct-remdups*
    [*THEN set-eq-iff-multiset-of-eq-distinct* [*THEN iffD2*]]])
apply *simp*
done

lemma *multiset-of-compl-union* [*simp*]:
  multiset-of [*x*←*xs*. *P x*] + *multiset-of* [*x*←*xs*. ¬*P x*] = *multiset-of xs*
by (*induct xs*) (*auto simp*: *union-ac*)

lemma *count-filter*:
  count (*multiset-of xs*) *x* = *length* [*y* ← *xs*. *y* = *x*]
by (*induct xs*) *auto*

lemma *nth-mem-multiset-of*: *i* < *length ls* ⟹ (*ls* ! *i*) :# *multiset-of ls*
apply (*induct ls arbitrary*: *i*)
 apply *simp*
apply (*case-tac i*)
 apply *auto*
done

lemma *multiset-of-remove1*: *multiset-of* (*remove1 a xs*) = *multiset-of xs* − {#*a*#}
by (*induct xs*) (*auto simp add*: *multiset-eq-conv-count-eq*)

lemma *multiset-of-eq-length*:
assumes *multiset-of xs* = *multiset-of ys*
shows *length xs* = *length ys*
using *assms*
proof (*induct arbitrary*: *ys rule*: *length-induct*)
  case (*1 xs ys*)
  show *?case*
  proof (*cases xs*)
    case *Nil* **with** *1.prems* **show** *?thesis* **by** *simp*
  next
    case (*Cons x xs'*)
    note *xCons* = *Cons*
    show *?thesis*
    proof (*cases ys*)
      case *Nil*
      with *1.prems Cons* **show** *?thesis* **by** *simp*
    next
      case (*Cons y ys'*)
      have *x-in-ys*: *x* = *y* ∨ *x* ∈ *set ys'*
      proof (*cases x* = *y*)

```
      case True then show ?thesis ..
    next
      case False
     from 1.prems [symmetric] xCons Cons have x :# multiset-of ys' + {#y#}
by simp
      with False show ?thesis by (simp add: mem-set-multiset-eq)
    qed
    from 1.hyps have IH: length xs' < length xs ⟶
      (∀ x. multiset-of xs' = multiset-of x ⟶ length xs' = length x) by blast
   from 1.prems x-in-ys Cons xCons have multiset-of xs' = multiset-of (remove1
x (y#ys'))
      apply −
      apply (simp add: multiset-of-remove1, simp only: add-eq-conv-diff)
      apply fastsimp
      done
   with IH xCons have IH': length xs' = length (remove1 x (y#ys')) by fastsimp
    from x-in-ys have x ≠ y ⟹ length ys' > 0 by auto
   with Cons xCons x-in-ys IH' show ?thesis by (auto simp add: length-remove1)
   qed
  qed
qed
```

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

```
lemma properties-for-sort:
  multiset-of ys = multiset-of xs ⟹ sorted ys ⟹ sort xs = ys
proof (induct xs arbitrary: ys)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  then have x ∈ set ys
    by (auto simp add:  mem-set-multiset-eq intro!: ccontr)
  with Cons.prems Cons.hyps [of remove1 x ys] show ?case
    by (simp add: sorted-remove1 multiset-of-remove1 insort-remove1)
qed
```

38.6 Pointwise ordering induced by count

```
definition
  mset-le :: 'a multiset ⇒ 'a multiset ⇒ bool  (infix ≤# 50) where
  (A ≤# B) = (∀ a. count A a ≤ count B a)

definition
  mset-less :: 'a multiset ⇒ 'a multiset ⇒ bool  (infix <# 50) where
  (A <# B) = (A ≤# B ∧ A ≠ B)

notation mset-le  (infix ⊆# 50)
notation mset-less  (infix ⊂# 50)
```

lemma *mset-le-refl*[*simp*]: $A \leq\# A$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-trans*: $A \leq\# B \Longrightarrow B \leq\# C \Longrightarrow A \leq\# C$
unfolding *mset-le-def* **by** (*fast intro*: *order-trans*)

lemma *mset-le-antisym*: $A \leq\# B \Longrightarrow B \leq\# A \Longrightarrow A = B$
apply (*unfold mset-le-def*)
apply (*rule multiset-eq-conv-count-eq* [*THEN iffD2*])
apply (*blast intro*: *order-antisym*)
done

lemma *mset-le-exists-conv*: $(A \leq\# B) = (\exists C.\ B = A + C)$
apply (*unfold mset-le-def*, *rule iffI*, *rule-tac* $x = B - A$ **in** *exI*)
apply (*auto intro*: *multiset-eq-conv-count-eq* [*THEN iffD2*])
done

lemma *mset-le-mono-add-right-cancel*[*simp*]: $(A + C \leq\# B + C) = (A \leq\# B)$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-mono-add-left-cancel*[*simp*]: $(C + A \leq\# C + B) = (A \leq\# B)$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-mono-add*: $[\![A \leq\# B;\ C \leq\# D]\!] \Longrightarrow A + C \leq\# B + D$
apply (*unfold mset-le-def*)
apply *auto*
apply (*erule-tac* $x = a$ **in** *allE*)+
apply *auto*
done

lemma *mset-le-add-left*[*simp*]: $A \leq\# A + B$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-add-right*[*simp*]: $B \leq\# A + B$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-single*: $a :\# B \Longrightarrow \{\#a\#\} \leq\# B$
by (*simp add*: *mset-le-def*)

lemma *multiset-diff-union-assoc*: $C \leq\# B \Longrightarrow A + B - C = A + (B - C)$
by (*simp add*: *multiset-eq-conv-count-eq mset-le-def*)

lemma *mset-le-multiset-union-diff-commute*:
assumes $B \leq\# A$
shows $A - B + C = A + C - B$
proof −
  from *mset-le-exists-conv* [*of B A*] *assms* **have** $\exists D.\ A = B + D$ **..**
  from *this* **obtain** D **where** $A = B + D$ **..**
  then show *?thesis*

```
    apply simp
    apply (subst union-commute)
    apply (subst multiset-diff-union-assoc)
    apply simp
    apply (simp add: diff-cancel)
    apply (subst union-assoc)
    apply (subst union-commute[of B -])
    apply (subst multiset-diff-union-assoc)
    apply simp
    apply (simp add: diff-cancel)
    done
qed

lemma multiset-of-remdups-le: multiset-of (remdups xs) ≤# multiset-of xs
apply (induct xs)
 apply auto
apply (rule mset-le-trans)
 apply auto
done

lemma multiset-of-update:
  i < length ls ⟹ multiset-of (ls[i := v]) = multiset-of ls − {#ls ! i#} + {#v#}
proof (induct ls arbitrary: i)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases i)
    case 0 then show ?thesis by simp
  next
    case (Suc i')
    with Cons show ?thesis
      apply simp
      apply (subst union-assoc)
      apply (subst union-commute [where M = {#v#} and N = {#x#}])
      apply (subst union-assoc [symmetric])
      apply simp
      apply (rule mset-le-multiset-union-diff-commute)
      apply (simp add: mset-le-single nth-mem-multiset-of)
      done
  qed
qed

lemma multiset-of-swap:
  i < length ls ⟹ j < length ls ⟹
    multiset-of (ls[j := ls ! i, i := ls ! j]) = multiset-of ls
apply (case-tac i = j)
 apply simp
apply (simp add: multiset-of-update)
```

apply (*subst elem-imp-eq-diff-union*[*symmetric*])
apply (*simp add*: *nth-mem-multiset-of*)
apply *simp*
done

interpretation *mset-order*: *order* [*op* ≤# *op* <#]
by (*auto intro*: *order.intro mset-le-refl mset-le-antisym*
mset-le-trans simp: *mset-less-def*)

interpretation *mset-order-cancel-semigroup*:
pordered-cancel-ab-semigroup-add [*op* + *op* ≤# *op* <#]
by *unfold-locales* (*erule mset-le-mono-add* [*OF mset-le-refl*])

interpretation *mset-order-semigroup-cancel*:
pordered-ab-semigroup-add-imp-le [*op* + *op* ≤# *op* <#]
by (*unfold-locales*) *simp*

lemma *mset-lessD*: *A* ⊂# *B* ⟹ *x* ∈# *A* ⟹ *x* ∈# *B*
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply (*erule-tac x*=*x* **in** *allE*)
apply *auto*
done

lemma *mset-leD*: *A* ⊆# *B* ⟹ *x* ∈# *A* ⟹ *x* ∈# *B*
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply (*erule-tac x* = *x* **in** *allE*)
apply *auto*
done

lemma *mset-less-insertD*: (*A* + {#*x*#} ⊂# *B*) ⟹ (*x* ∈# *B* ∧ *A* ⊂# *B*)
apply (*rule conjI*)
apply (*simp add*: *mset-lessD*)
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply *safe*
apply (*erule-tac x* = *a* **in** *allE*)
apply (*auto split*: *split-if-asm*)
done

lemma *mset-le-insertD*: (*A* + {#*x*#} ⊆# *B*) ⟹ (*x* ∈# *B* ∧ *A* ⊆# *B*)
apply (*rule conjI*)
apply (*simp add*: *mset-leD*)
apply (*force simp*: *mset-le-def mset-less-def split*: *split-if-asm*)
done

lemma *mset-less-of-empty*[*simp*]: *A* ⊂# {#} = *False*
by (*induct A*) (*auto simp*: *mset-le-def mset-less-def*)

lemma *multi-psub-of-add-self*[*simp*]: *A* ⊂# *A* + {#*x*#}

by (*auto simp*: *mset-le-def mset-less-def*)

lemma *multi-psub-self*[*simp*]: $A \subset\# A = False$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-add-bothsides*:
$T + \{\#x\#\} \subset\# S + \{\#x\#\} \Longrightarrow T \subset\# S$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-empty-nonempty*: $(\{\#\} \subset\# S) = (S \neq \{\#\})$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-size*: $A \subset\# B \Longrightarrow size\ A < size\ B$
proof (*induct A arbitrary*: *B*)
case (*empty M*)
then have $M \neq \{\#\}$ **by** (*simp add*: *mset-less-empty-nonempty*)
then obtain $M'\ x$ **where** $M = M' + \{\#x\#\}$
by (*blast dest*: *multi-nonempty-split*)
then show *?case* **by** *simp*
next
case (*add S x T*)
have *IH*: $\bigwedge B.\ S \subset\# B \Longrightarrow size\ S < size\ B$ **by** *fact*
have *SxsubT*: $S + \{\#x\#\} \subset\# T$ **by** *fact*
then have $x \in\# T$ **and** $S \subset\# T$ **by** (*auto dest*: *mset-less-insertD*)
then obtain T' **where** *T*: $T = T' + \{\#x\#\}$
by (*blast dest*: *multi-member-split*)
then have $S \subset\# T'$ **using** *SxsubT*
by (*blast intro*: *mset-less-add-bothsides*)
then have $size\ S < size\ T'$ **using** *IH* **by** *simp*
then show *?case* **using** *T* **by** *simp*
qed

lemmas *mset-less-trans* = *mset-order.less-eq-less.less-trans*

lemma *mset-less-diff-self*: $c \in\# B \Longrightarrow B - \{\#c\#\} \subset\# B$
by (*auto simp*: *mset-le-def mset-less-def multi-drop-mem-not-eq*)

38.7 Strong induction and subset induction for multisets

Well-foundedness of proper subset operator:

proper multiset subset

definition
mset-less-rel :: ('*a multiset* × '*a multiset*) *set* **where**
mset-less-rel = {(*A*,*B*). $A \subset\# B$}

lemma *multiset-add-sub-el-shuffle*:
assumes $c \in\# B$ **and** $b \neq c$
shows $B - \{\#c\#\} + \{\#b\#\} = B + \{\#b\#\} - \{\#c\#\}$

proof −
from ‹$c \in\# B$› **obtain** A **where** B: $B = A + \{\#c\#\}$
by (*blast dest*: *multi-member-split*)
have $A + \{\#b\#\} = A + \{\#b\#\} + \{\#c\#\} - \{\#c\#\}$ **by** *simp*
then have $A + \{\#b\#\} = A + \{\#c\#\} + \{\#b\#\} - \{\#c\#\}$
by (*simp add*: *union-ac*)
then show *?thesis* **using** B **by** *simp*
qed

lemma *wf-mset-less-rel*: *wf mset-less-rel*
apply (*unfold mset-less-rel-def*)
apply (*rule wf-measure* [*THEN wf-subset*, **where** *f1*=*size*])
apply (*clarsimp simp*: *measure-def inv-image-def mset-less-size*)
done

The induction rules:

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall A.\ A \subset\# B \longrightarrow P\ A \Longrightarrow P\ B$
shows $P\ B$
apply (*rule wf-mset-less-rel* [*THEN wf-induct*])
apply (*rule ih*, *auto simp*: *mset-less-rel-def*)
done

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \subseteq\# A$
and *empty*: $P\ \{\#\}$
and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (F + \{\#a\#\})$
shows $P\ F$
proof −
from ‹$F \subseteq\# A$›
show *?thesis*
proof (*induct F*)
show $P\ \{\#\}$ **by** *fact*
next
fix $x\ F$
assume P: $F \subseteq\# A \Longrightarrow P\ F$ **and** i: $F + \{\#x\#\} \subseteq\# A$
show $P\ (F + \{\#x\#\})$
proof (*rule insert*)
from i **show** $x \in\# A$ **by** (*auto dest*: *mset-le-insertD*)
from i **have** $F \subseteq\# A$ **by** (*auto dest*: *mset-le-insertD*)
with P **show** $P\ F$ **.**
qed
qed
qed

A consequence: Extensionality.

lemma *multi-count-eq*: $(\forall x.\ count\ A\ x = count\ B\ x) = (A = B)$
apply (*rule iffI*)
prefer *2*

apply *clarsimp*
apply (*induct A arbitrary*: *B rule*: *full-multiset-induct*)
apply (*rename-tac C*)
apply (*case-tac B rule*: *multiset-cases*)
apply (*simp add*: *empty-multiset-count*)
apply *simp*
apply (*case-tac x* $\in\#$ *C*)
apply (*force dest*: *multi-member-split*)
apply (*erule-tac x* = *x* **in** *allE*)
apply *simp*
done

lemmas *multi-count-ext* = *multi-count-eq* [*THEN iffD1*, *rule-format*]

38.8 The fold combinator

The intended behaviour is *fold-mset f z* $\{\#x_1, ..., x_n\#\} = f\ x_1\ (\ldots\ (f\ x_n\ z)\ldots)$ if f is associative-commutative.

The graph of *fold-mset*, z: the start element, f: folding function, A: the multiset, y: the result.

inductive
fold-msetG :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a$ *multiset* $\Rightarrow 'b \Rightarrow$ *bool*
for $f :: 'a \Rightarrow 'b \Rightarrow 'b$
and $z :: 'b$
where
emptyI [*intro*]: *fold-msetG f z* $\{\#\}$ *z*
| *insertI* [*intro*]: *fold-msetG f z A y* $\Longrightarrow$ *fold-msetG f z* $(A + \{\#x\#\})$ $(f\ x\ y)$

inductive-cases *empty-fold-msetGE* [*elim*!]: *fold-msetG f z* $\{\#\}$ *x*
inductive-cases *insert-fold-msetGE*: *fold-msetG f z* $(A + \{\#\})$ *y*

definition
fold-mset :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a$ *multiset* $\Rightarrow 'b$ **where**
fold-mset f z A = (*THE x*. *fold-msetG f z A x*)

lemma *Diff1-fold-msetG*:
fold-msetG f z $(A - \{\#x\#\})$ *y* $\Longrightarrow$ *x* $\in\#$ *A* $\Longrightarrow$ *fold-msetG f z A* $(f\ x\ y)$
apply (*frule-tac x* = *x* **in** *fold-msetG.insertI*)
apply *auto*
done

lemma *fold-msetG-nonempty*: $\exists x$. *fold-msetG f z A x*
apply (*induct A*)
apply *blast*
apply *clarsimp*
apply (*drule-tac x* = *x* **in** *fold-msetG.insertI*)
apply *auto*
done

lemma *fold-mset-empty*[*simp*]: *fold-mset f z* {#} = *z*
unfolding *fold-mset-def* **by** *blast*

locale *left-commutative* =
fixes *f* :: *'a* => *'b* => *'b*
assumes *left-commute*: *f x* (*f y z*) = *f y* (*f x z*)
begin

lemma *fold-msetG-determ*:
fold-msetG f z A x ⟹ *fold-msetG f z A y* ⟹ *y* = *x*
proof (*induct arbitrary*: *x y z rule*: *full-multiset-induct*)
case (*less M* x_1 x_2 *Z*)
have *IH*: ∀ *A*. *A* ⊂# *M* ⟶
(∀ *x x' x''*. *fold-msetG f x'' A x* ⟶ *fold-msetG f x'' A x'*
⟶ *x'* = *x*) **by** *fact*
have $Mfoldx_1$: *fold-msetG f Z M* x_1 **and** $Mfoldx_2$: *fold-msetG f Z M* x_2 **by** *fact*+
show *?case*
proof (*rule fold-msetG.cases* [*OF* $Mfoldx_1$])
assume *M* = {#} **and** x_1 = *Z*
then show *?case* **using** $Mfoldx_2$ **by** *auto*
next
fix *B b u*
assume *M* = *B* + {#*b*#} **and** x_1 = *f b u* **and** *Bu*: *fold-msetG f Z B u*
then have *MBb*: *M* = *B* + {#*b*#} **and** x_1: x_1 = *f b u* **by** *auto*
show *?case*
proof (*rule fold-msetG.cases* [*OF* $Mfoldx_2$])
assume *M* = {#} x_2 = *Z*
then show *?case* **using** $Mfoldx_1$ **by** *auto*
next
fix *C c v*
assume *M* = *C* + {#*c*#} **and** x_2 = *f c v* **and** *Cv*: *fold-msetG f Z C v*
then have *MCc*: *M* = *C* + {#*c*#} **and** x_2: x_2 = *f c v* **by** *auto*
then have *CsubM*: *C* ⊂# *M* **by** *simp*
from *MBb* **have** *BsubM*: *B* ⊂# *M* **by** *simp*
show *?case*
proof *cases*
assume *b*=*c*
then moreover have *B* = *C* **using** *MBb MCc* **by** *auto*
ultimately show *?thesis* **using** *Bu Cv* x_1 x_2 *CsubM IH* **by** *auto*
next
assume *diff*: *b* ≠ *c*
let *?D* = *B* − {#*c*#}
have *cinB*: *c* ∈# *B* **and** *binC*: *b* ∈# *C* **using** *MBb MCc diff*
by (*auto intro*: *insert-noteq-member dest*: *sym*)
have *B* − {#*c*#} ⊂# *B* **using** *cinB* **by** (*rule mset-less-diff-self*)
then have *DsubM*: *?D* ⊂# *M* **using** *BsubM* **by** (*blast intro*: *mset-less-trans*)
from *MBb MCc* **have** *B* + {#*b*#} = *C* + {#*c*#} **by** *blast*
then have [*simp*]: *B* + {#*b*#} − {#*c*#} = *C*

```
        using MBb MCc binC cinB by auto
      have B: B = ?D + {#c#} and C: C = ?D + {#b#}
        using MBb MCc diff binC cinB
        by (auto simp: multiset-add-sub-el-shuffle)
      then obtain d where Dfoldd: fold-msetG f Z ?D d
        using fold-msetG-nonempty by iprover
      then have fold-msetG f Z B (f c d) using cinB
        by (rule Diff1-fold-msetG)
      then have f c d = u using IH BsubM Bu by blast
      moreover
      have fold-msetG f Z C (f b d) using binC cinB diff Dfoldd
        by (auto simp: multiset-add-sub-el-shuffle
          dest: fold-msetG.insertI [where x=b])
      then have f b d = v using IH CsubM Cv by blast
      ultimately show ?thesis using x1 x2
        by (auto simp: left-commute)
    qed
   qed
  qed
qed

lemma fold-mset-insert-aux:
  (fold-msetG f z (A + {#x#}) v) =
    (∃ y. fold-msetG f z A y ∧ v = f x y)
apply (rule iffI)
 prefer 2
 apply blast
apply (rule-tac A=A and f=f in fold-msetG-nonempty [THEN exE, standard])
apply (blast intro: fold-msetG-determ)
done

lemma fold-mset-equality: fold-msetG f z A y ⟹ fold-mset f z A = y
unfolding fold-mset-def by (blast intro: fold-msetG-determ)

lemma fold-mset-insert:
  fold-mset f z (A + {#x#}) = f x (fold-mset f z A)
apply (simp add: fold-mset-def fold-mset-insert-aux union-commute)
apply (rule the-equality)
 apply (auto cong add: conj-cong
    simp add: fold-mset-def [symmetric] fold-mset-equality fold-msetG-nonempty)
done

lemma fold-mset-insert-idem:
  fold-mset f z (A + {#a#}) = f a (fold-mset f z A)
apply (simp add: fold-mset-def fold-mset-insert-aux)
apply (rule the-equality)
 apply (auto cong add: conj-cong
    simp add: fold-mset-def [symmetric] fold-mset-equality fold-msetG-nonempty)
done
```

lemma *fold-mset-commute*: *f x* (*fold-mset f z A*) = *fold-mset f* (*f x z*) *A*
by (*induct A*) (*auto simp*: *fold-mset-insert left-commute* [*of x*])

lemma *fold-mset-single* [*simp*]: *fold-mset f z* {#*x*#} = *f x z*
using *fold-mset-insert* [*of z* {#}] **by** *simp*

lemma *fold-mset-union* [*simp*]:
fold-mset f z (*A*+*B*) = *fold-mset f* (*fold-mset f z A*) *B*
proof (*induct A*)
case *empty* **then show** *?case* **by** *simp*
next
case (*add A x*)
have *A* + {#*x*#} + *B* = (*A*+*B*) + {#*x*#} **by**(*simp add*:*union-ac*)
then have *fold-mset f z* (*A* + {#*x*#} + *B*) = *f x* (*fold-mset f z* (*A* + *B*))
by (*simp add*: *fold-mset-insert*)
also have ... = *fold-mset f* (*fold-mset f z* (*A* + {#*x*#})) *B*
by (*simp add*: *fold-mset-commute*[*of x*,*symmetric*] *add fold-mset-insert*)
finally show *?case* **.**
qed

lemma *fold-mset-fusion*:
includes *left-commutative g*
shows (⋀*x y*. *h* (*g x y*) = *f x* (*h y*)) ⟹ *h* (*fold-mset g w A*) = *fold-mset f* (*h w*) *A*
by (*induct A*) *auto*

lemma *fold-mset-rec*:
assumes *a* ∈# *A*
shows *fold-mset f z A* = *f a* (*fold-mset f z* (*A* − {#*a*#}))
proof −
from *assms* **obtain** *A*′ **where** *A* = *A*′ + {#*a*#}
by (*blast dest*: *multi-member-split*)
then show *?thesis* **by** *simp*
qed

end

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 *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.

38.9 Image

definition [*code func del*]: *image-mset f* == *fold-mset* (*op* + *o single o f*) {#}

interpretation *image-left-comm*: *left-commutative* [*op* + *o single o f*]
by (*unfold-locales*) (*simp add*:*union-ac*)

lemma *image-mset-empty* [*simp*, *code func*]: *image-mset f* {#} = {#}
by (*simp add*: *image-mset-def*)

lemma *image-mset-single* [*simp*, *code func*]: *image-mset f* {#*x*#} = {#*f x*#}
by (*simp add*: *image-mset-def*)

lemma *image-mset-insert*:
image-mset f (*M* + {#*a*#}) = *image-mset f M* + {#*f a*#}
by (*simp add*: *image-mset-def add-ac*)

lemma *image-mset-union*[*simp*, *code func*]:
image-mset f (*M*+*N*) = *image-mset f M* + *image-mset f N*
apply (*induct N*)
apply *simp*
apply (*simp add*: *union-assoc* [*symmetric*] *image-mset-insert*)
done

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*
by (*induct M*) *simp-all*

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} ⟷ *M* = {#}
by (*cases M*) *auto*

syntax
comprehension1-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset*
(({#-/. - :# -#}))
translations
{#*e*. *x*:#*M*#} == *CONST image-mset* (%*x*. *e*) *M*

syntax
comprehension2-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset*
(({#-/ | - :# -./ -#}))
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*#}#}.

end

39 NatPair: Pairs of Natural Numbers

theory *NatPair*
imports *ATP-Linkup*

begin

An injective function from $\mathbb{N}^2$ to $\mathbb{N}$. Definition and proofs are from [4, page 85].

definition
nat2-to-nat:: (*nat* ∗ *nat*) ⇒ *nat* **where**
nat2-to-nat pair = (*let* (*n*,*m*) = *pair in* (*n*+*m*) ∗ *Suc* (*n*+*m*) *div* 2 + *n*)

lemma *dvd2-a-x-suc-a*: *2 dvd a* ∗ (*Suc a*)
proof (*cases 2 dvd a*)
case *True*
then show *?thesis* **by** (*rule dvd-mult2*)
next
case *False*
then have *Suc* (*a mod 2*) = *2* **by** (*simp add*: *dvd-eq-mod-eq-0*)
then have *Suc a mod 2* = *0* **by** (*simp add*: *mod-Suc*)
then have *2 dvd Suc a* **by** (*simp only*:*dvd-eq-mod-eq-0*)
then show *?thesis* **by** (*rule dvd-mult*)
qed

lemma
assumes *eq*: *nat2-to-nat* (*u*,*v*) = *nat2-to-nat* (*x*,*y*)
shows *nat2-to-nat-help*: *u*+*v* ≤ *x*+*y*
proof (*rule classical*)
assume ¬ *?thesis*
then have *contrapos*: *x*+*y* < *u*+*v*
by *simp*
have *nat2-to-nat* (*x*,*y*) < (*x*+*y*) ∗ *Suc* (*x*+*y*) *div 2* + *Suc* (*x* + *y*)
by (*unfold nat2-to-nat-def*) (*simp add*: *Let-def*)
also have ... = (*x*+*y*)∗*Suc*(*x*+*y*) *div 2* + *2* ∗ *Suc*(*x*+*y*) *div 2*
by (*simp only*: *div-mult-self1-is-m*)
also have ... = (*x*+*y*)∗*Suc*(*x*+*y*) *div 2* + *2* ∗ *Suc*(*x*+*y*) *div 2*
+ ((*x*+*y*)∗*Suc*(*x*+*y*) *mod 2* + *2* ∗ *Suc*(*x*+*y*) *mod 2*) *div 2*
proof −
have *2 dvd* (*x*+*y*)∗*Suc*(*x*+*y*)
by (*rule dvd2-a-x-suc-a*)
then have (*x*+*y*)∗*Suc*(*x*+*y*) *mod 2* = *0*
by (*simp only*: *dvd-eq-mod-eq-0*)
also
have *2* ∗ *Suc*(*x*+*y*) *mod 2* = *0*
by (*rule mod-mult-self1-is-0*)
ultimately have
((*x*+*y*)∗*Suc*(*x*+*y*) *mod 2* + *2* ∗ *Suc*(*x*+*y*) *mod 2*) *div 2* = *0*
by *simp*
then show *?thesis*
by *simp*
qed
also have ... = ((*x*+*y*)∗*Suc*(*x*+*y*) + *2*∗*Suc*(*x*+*y*)) *div 2*
by (*rule div-add1-eq* [*symmetric*])

```
  also have ... = ((x+y+2)*Suc(x+y)) div 2
    by (simp only: add-mult-distrib [symmetric])
  also from contrapos have ... ≤ ((Suc(u+v))*(u+v)) div 2
    by (simp only: mult-le-mono div-le-mono)
  also have ... ≤ nat2-to-nat (u,v)
    by (unfold nat2-to-nat-def) (simp add: Let-def)
  finally show ?thesis
    by (simp only: eq)
qed

theorem nat2-to-nat-inj: inj nat2-to-nat
proof −
  {
    fix u v x y
    assume eq1: nat2-to-nat (u,v) = nat2-to-nat (x,y)
    then have u+v ≤ x+y by (rule nat2-to-nat-help)
    also from eq1 [symmetric] have x+y ≤ u+v
      by (rule nat2-to-nat-help)
    finally have eq2: u+v = x+y .
    with eq1 have ux: u=x
      by (simp add: nat2-to-nat-def Let-def)
    with eq2 have vy: v=y by simp
    with ux have (u,v) = (x,y) by simp
  }
  then have ⋀x y. nat2-to-nat x = nat2-to-nat y ⟹ x=y by fast
  then show ?thesis unfolding inj-on-def by simp
qed

end
```

40 Nat-Infinity: Natural numbers with infinity

```
theory Nat-Infinity
imports ATP-Linkup
begin
```

40.1 Definitions

We extend the standard natural numbers by a special value indicating infinity. This includes extending the ordering relations *op* < and *op* ≤.

```
datatype inat = Fin nat | Infty

notation (xsymbols)
  Infty  (∞)

notation (HTML output)
  Infty  (∞)
```

definition
iSuc :: *inat* => *inat* **where**
iSuc i = (*case i of Fin n* => *Fin* (*Suc n*) | ∞ => ∞)

instantiation *inat* :: {*ord*, *zero*}
begin

definition
Zero-inat-def: *0* == *Fin 0*

definition
iless-def: *m* < *n* ==
case m of Fin m1 => (*case n of Fin n1* => *m1* < *n1* | ∞ => *True*)
| ∞ => *False*

definition
ile-def: *m* ≤ *n* ==
case n of Fin n1 => (*case m of Fin m1* => *m1* ≤ *n1* | ∞ => *False*)
| ∞ => *True*

instance ..

end

lemmas *inat-defs* = *Zero-inat-def iSuc-def iless-def ile-def*
lemmas *inat-splits* = *inat.split inat.split-asm*

Below is a not quite complete set of theorems. Use the method (*simp add*: *inat-defs split*:*inat-splits*, *arith?*) to prove new theorems or solve arithmetic subgoals involving *inat* on the fly.

40.2 Constructors

lemma *Fin-0*: *Fin 0* = *0*
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Infty-ne-i0* [*simp*]: ∞ ≠ *0*
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *i0-ne-Infty* [*simp*]: *0* ≠ ∞
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iSuc-Fin* [*simp*]: *iSuc* (*Fin n*) = *Fin* (*Suc n*)
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iSuc-Infty* [*simp*]: *iSuc* ∞ = ∞
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iSuc-ne-0* [*simp*]: $iSuc\ n \neq 0$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iSuc-inject* [*simp*]: $(iSuc\ x = iSuc\ y) = (x = y)$
by (*simp add*: *inat-defs split*:*inat-splits*)

40.3 Ordering relations

instance *inat* :: *linorder*
proof
fix x :: *inat*
show $x \leq x$
by (*simp add*: *inat-defs split*: *inat-splits*)
next
fix $x\ y$ :: *inat*
assume $x \leq y$ **and** $y \leq x$ **thus** $x = y$
by (*simp add*: *inat-defs split*: *inat-splits*)
next
fix $x\ y\ z$ :: *inat*
assume $x \leq y$ **and** $y \leq z$ **thus** $x \leq z$
by (*simp add*: *inat-defs split*: *inat-splits*)
next
fix $x\ y$ :: *inat*
show $(x < y) = (x \leq y \wedge x \neq y)$
by (*simp add*: *inat-defs order-less-le split*: *inat-splits*)
next
fix $x\ y$ :: *inat*
show $x \leq y \vee y \leq x$
by (*simp add*: *inat-defs linorder-linear split*: *inat-splits*)
qed

lemma *Infty-ilessE* [*elim*!]: $\infty < Fin\ m \Longrightarrow R$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iless-linear*: $m < n \vee m = n \vee n < (m::inat)$
by (*rule linorder-less-linear*)

lemma *iless-not-refl*: $\neg\ n < (n::inat)$
by (*rule order-less-irrefl*)

lemma *iless-trans*: $i < j \Longrightarrow j < k \Longrightarrow i < (k::inat)$
by (*rule order-less-trans*)

lemma *iless-not-sym*: $n < m \Longrightarrow \neg\ m < (n::inat)$
by (*rule order-less-not-sym*)

lemma *Fin-iless-mono* [*simp*]: $(Fin\ n < Fin\ m) = (n < m)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Fin-iless-Infty* [*simp*]: $Fin\ n < \infty$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Infty-eq* [*simp*]: $(n < \infty) = (n \neq \infty)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *i0-eq* [*simp*]: $((0::inat) < n) = (n \neq 0)$
by (*fastsimp simp*: *inat-defs split*:*inat-splits*)

lemma *i0-iless-iSuc* [*simp*]: $0 < iSuc\ n$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *not-ilessi0* [*simp*]: $\neg\ n < (0::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Fin-iless*: $n < Fin\ m \Longrightarrow \exists\, k.\ n = Fin\ k$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iSuc-mono* [*simp*]: $(iSuc\ n < iSuc\ m) = (n < m)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *ile-def2*: $(m \leq n) = (m < n \vee m = (n::inat))$
by (*rule order-le-less*)

lemma *ile-refl* [*simp*]: $n \leq (n::inat)$
by (*rule order-refl*)

lemma *ile-trans*: $i \leq j \Longrightarrow j \leq k \Longrightarrow i \leq (k::inat)$
by (*rule order-trans*)

lemma *ile-iless-trans*: $i \leq j \Longrightarrow j < k \Longrightarrow i < (k::inat)$
by (*rule order-le-less-trans*)

lemma *iless-ile-trans*: $i < j \Longrightarrow j \leq k \Longrightarrow i < (k::inat)$
by (*rule order-less-le-trans*)

lemma *Infty-ub* [*simp*]: $n \leq \infty$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *i0-lb* [*simp*]: $(0::inat) \leq n$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Infty-ileE* [*elim*!]: $\infty \leq Fin\ m \Longrightarrow R$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Fin-ile-mono* [*simp*]: $(Fin\ n \leq Fin\ m) = (n \leq m)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *ilessI1*: $n \leq m \Longrightarrow n \neq m \Longrightarrow n < (m::inat)$
by (*rule order-le-neq-trans*)

lemma *ileI1*: $m < n \Longrightarrow iSuc\ m \leq n$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Suc-ile-eq*: $(Fin\ (Suc\ m) \leq n) = (Fin\ m < n)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith*)

lemma *iSuc-ile-mono* [*simp*]: $(iSuc\ n \leq iSuc\ m) = (n \leq m)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iless-Suc-eq* [*simp*]: $(Fin\ m < iSuc\ n) = (Fin\ m \leq n)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith*)

lemma *not-iSuc-ilei0* [*simp*]: $\neg\ iSuc\ n \leq 0$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *ile-iSuc* [*simp*]: $n \leq iSuc\ n$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Fin-ile*: $n \leq Fin\ m \Longrightarrow \exists k.\ n = Fin\ k$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *chain-incr*: $\forall i.\ \exists j.\ Y\ i < Y\ j \Longrightarrow \exists j.\ Fin\ k < Y\ j$
apply (*induct-tac k*)
apply (*simp* (*no-asm*) *only*: *Fin-0*)
apply (*fast intro*: *ile-iless-trans* [*OF i0-lb*])
apply (*erule exE*)
apply (*drule spec*)
apply (*erule exE*)
apply (*drule ileI1*)
apply (*rule iSuc-Fin* [*THEN subst*])
apply (*rule exI*)
apply (*erule* (*1*) *ile-iless-trans*)
done

40.4 Well-ordering

lemma *less-FinE*:
$[\!|\ n < Fin\ m;\ !!k.\ n = Fin\ k \Longrightarrow k < m \Longrightarrow P\ |\!] \Longrightarrow P$
by (*induct n*) *auto*

lemma *less-InftyE*:
$[\!|\ n < Infty;\ !!k.\ n = Fin\ k \Longrightarrow P\ |\!] \Longrightarrow P$
by (*induct n*) *auto*

lemma *inat-less-induct*:

```
  assumes prem: !!n. ∀ m::inat. m < n --> P m ==> P n shows P n
proof −
  have P-Fin: !!k. P (Fin k)
    apply (rule nat-less-induct)
    apply (rule prem, clarify)
    apply (erule less-FinE, simp)
    done
  show ?thesis
  proof (induct n)
    fix nat
    show P (Fin nat) by (rule P-Fin)
  next
    show P Infty
      apply (rule prem, clarify)
      apply (erule less-InftyE)
      apply (simp add: P-Fin)
      done
  qed
qed

instance inat :: wellorder
proof
  show wf {(x::inat, y::inat). x < y}
  proof (rule wfUNIVI)
    fix P and x :: inat
    assume ∀ x::inat. (∀ y. (y, x) ∈ {(x, y). x < y} ⟶ P y) ⟶ P x
    hence 1: !!x::inat. ALL y. y < x --> P y ==> P x by fast
    thus P x by (rule inat-less-induct)
  qed
qed

end
```

41 Nested-Environment: Nested environments

```
theory Nested-Environment
imports List
begin
```

Consider a partial function $e :: 'a => 'b\ option$; this may be understood as an *environment* mapping indexes $'a$ to optional entry values $'b$ (cf. the basic theory *Map* of Isabelle/HOL). This basic idea is easily generalized to that of a *nested environment*, where entries may be either basic values or again proper environments. Then each entry is accessed by a *path*, i.e. a list of indexes leading to its position within the structure.

```
datatype ('a, 'b, 'c) env =
    Val 'a
```

| *Env 'b 'c => ('a, 'b, 'c) env option*

In the type *('a, 'b, 'c) env* the parameter *'a* refers to basic values (occurring in terminal positions), type *'b* to values associated with proper (inner) environments, and type *'c* with the index type for branching. Note that there is no restriction on any of these types. In particular, arbitrary branching may yield rather large (transfinite) tree structures.

41.1 The lookup operation

Lookup in nested environments works by following a given path of index elements, leading to an optional result (a terminal value or nested environment). A *defined position* within a nested environment is one where *lookup* at its path does not yield *None*.

consts
lookup :: ('a, 'b, 'c) env => 'c list => ('a, 'b, 'c) env option
lookup-option :: ('a, 'b, 'c) env option => 'c list => ('a, 'b, 'c) env option

primrec (*lookup*)
lookup (Val a) xs = (if xs = [] then Some (Val a) else None)
lookup (Env b es) xs =
(case xs of
[] => Some (Env b es)
| y # ys => lookup-option (es y) ys)
lookup-option None xs = None
lookup-option (Some e) xs = lookup e xs

hide *const lookup-option*

The characteristic cases of *lookup* are expressed by the following equalities.

theorem *lookup-nil*: *lookup e [] = Some e*
by (*cases e*) *simp-all*

theorem *lookup-val-cons*: *lookup (Val a) (x # xs) = None*
by *simp*

theorem *lookup-env-cons*:
lookup (Env b es) (x # xs) =
(case es x of
None => None
| Some e => lookup e xs)
by (*cases es x*) *simp-all*

lemmas *lookup.simps* [*simp del*]
and *lookup-simps* [*simp*] = *lookup-nil lookup-val-cons lookup-env-cons*

```
theorem lookup-eq:
  lookup env xs =
    (case xs of
      [] => Some env
    | x # xs =>
      (case env of
        Val a => None
      | Env b es =>
          (case es x of
            None => None
          | Some e => lookup e xs)))
  by (simp split: list.split env.split)
```

Displaced *lookup* operations, relative to a certain base path prefix, may be reduced as follows. There are two cases, depending whether the environment actually extends far enough to follow the base path.

```
theorem lookup-append-none:
  assumes lookup env xs = None
  shows lookup env (xs @ ys) = None
  using assms
proof (induct xs arbitrary: env)
  case Nil
  then have False by simp
  then show ?case ..
next
  case (Cons x xs)
  show ?case
  proof (cases env)
    case Val
    then show ?thesis by simp
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      with Env show ?thesis by simp
    next
      case (Some e)
      note es = ‹es x = Some e›
      show ?thesis
      proof (cases lookup e xs)
        case None
        then have lookup e (xs @ ys) = None by (rule Cons.hyps)
        with Env Some show ?thesis by simp
      next
        case Some
        with Env es have False using Cons.prems by simp
        then show ?thesis ..
      qed
```

```
    qed
  qed
qed

theorem lookup-append-some:
  assumes lookup env xs = Some e
  shows lookup env (xs @ ys) = lookup e ys
  using assms
proof (induct xs arbitrary: env e)
  case Nil
  then have env = e by simp
  then show lookup env ([] @ ys) = lookup e ys by simp
next
  case (Cons x xs)
  note asm = ‹lookup env (x # xs) = Some e›
  show lookup env ((x # xs) @ ys) = lookup e ys
  proof (cases env)
    case (Val a)
    with asm have False by simp
    then show ?thesis ..
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      with asm Env have False by simp
      then show ?thesis ..
    next
      case (Some e')
      note es = ‹es x = Some e'›
      show ?thesis
      proof (cases lookup e' xs)
        case None
        with asm Env es have False by simp
        then show ?thesis ..
      next
        case Some
        with asm Env es have lookup e' xs = Some e
          by simp
        then have lookup e' (xs @ ys) = lookup e ys by (rule Cons.hyps)
        with Env es show ?thesis by simp
      qed
    qed
  qed
qed
```

Successful *lookup* deeper down an environment structure means we are able to peek further up as well. Note that this is basically just the contrapositive statement of *lookup-append-none* above.

theorem *lookup-some-append*:
assumes *lookup env* (*xs* @ *ys*) = *Some e*
shows ∃ *e*. *lookup env xs* = *Some e*
proof −
from *assms* **have** *lookup env* (*xs* @ *ys*) ≠ *None* **by** *simp*
then have *lookup env xs* ≠ *None*
by (*rule contrapos-nn*) (*simp only*: *lookup-append-none*)
then show *?thesis* **by** (*simp*)
qed

The subsequent statement describes in more detail how a successful *lookup* with a non-empty path results in a certain situation at any upper position.

theorem *lookup-some-upper*:
assumes *lookup env* (*xs* @ *y* # *ys*) = *Some e*
shows ∃ *b′ es′ env′*.
lookup env xs = *Some* (*Env b′ es′*) ∧
es′ y = *Some env′* ∧
lookup env′ ys = *Some e*
using *assms*
proof (*induct xs arbitrary*: *env e*)
case *Nil*
from *Nil.prems* **have** *lookup env* (*y* # *ys*) = *Some e*
by *simp*
then obtain *b′ es′ env′* **where**
env: *env* = *Env b′ es′* **and**
es′: *es′ y* = *Some env′* **and**
look′: *lookup env′ ys* = *Some e*
by (*auto simp add*: *lookup-eq split*: *option.splits env.splits*)
from *env* **have** *lookup env* [] = *Some* (*Env b′ es′*) **by** *simp*
with *es′ look′* **show** *?case* **by** *blast*
next
case (*Cons x xs*)
from *Cons.prems*
obtain *b′ es′ env′* **where**
env: *env* = *Env b′ es′* **and**
es′: *es′ x* = *Some env′* **and**
look′: *lookup env′* (*xs* @ *y* # *ys*) = *Some e*
by (*auto simp add*: *lookup-eq split*: *option.splits env.splits*)
from *Cons.hyps* [*OF look′*] **obtain** *b″ es″ env″* **where**
upper′: *lookup env′ xs* = *Some* (*Env b″ es″*) **and**
es″: *es″ y* = *Some env″* **and**
look″: *lookup env″ ys* = *Some e*
by *blast*
from *env es′ upper′* **have** *lookup env* (*x* # *xs*) = *Some* (*Env b″ es″*)
by *simp*
with *es″ look″* **show** *?case* **by** *blast*
qed

41.2 The update operation

Update at a certain position in a nested environment may either delete an existing entry, or overwrite an existing one. Note that update at undefined positions is simple absorbed, i.e. the environment is left unchanged.

```
consts
  update :: 'c list => ('a, 'b, 'c) env option
    => ('a, 'b, 'c) env => ('a, 'b, 'c) env
  update-option :: 'c list => ('a, 'b, 'c) env option
    => ('a, 'b, 'c) env option => ('a, 'b, 'c) env option

primrec (update)
  update xs opt (Val a) =
    (if xs = [] then (case opt of None => Val a | Some e => e)
    else Val a)
  update xs opt (Env b es) =
    (case xs of
      [] => (case opt of None => Env b es | Some e => e)
    | y # ys => Env b (es (y := update-option ys opt (es y))))
  update-option xs opt None =
    (if xs = [] then opt else None)
  update-option xs opt (Some e) =
    (if xs = [] then opt else Some (update xs opt e))

hide const update-option
```

The characteristic cases of *update* are expressed by the following equalities.

```
theorem update-nil-none: update [] None env = env
  by (cases env) simp-all

theorem update-nil-some: update [] (Some e) env = e
  by (cases env) simp-all

theorem update-cons-val: update (x # xs) opt (Val a) = Val a
  by simp

theorem update-cons-nil-env:
    update [x] opt (Env b es) = Env b (es (x := opt))
  by (cases es x) simp-all

theorem update-cons-cons-env:
  update (x # y # ys) opt (Env b es) =
    Env b (es (x :=
      (case es x of
        None => None
      | Some e => Some (update (y # ys) opt e))))
  by (cases es x) simp-all
```

```
lemmas update.simps [simp del]
  and update-simps [simp] = update-nil-none update-nil-some
    update-cons-val update-cons-nil-env update-cons-cons-env

lemma update-eq:
  update xs opt env =
    (case xs of
      [] =>
        (case opt of
          None => env
        | Some e => e)
    | x # xs =>
        (case env of
          Val a => Val a
        | Env b es =>
            (case xs of
              [] => Env b (es (x := opt))
            | y # ys =>
                Env b (es (x :=
                  (case es x of
                    None => None
                  | Some e => Some (update (y # ys) opt e)))))))
  by (simp split: list.split env.split option.split)
```

The most basic correspondence of *lookup* and *update* states that after *update* at a defined position, subsequent *lookup* operations would yield the new value.

```
theorem lookup-update-some:
  assumes lookup env xs = Some e
  shows lookup (update xs (Some env') env) xs = Some env'
  using assms
proof (induct xs arbitrary: env e)
  case Nil
  then have env = e by simp
  then show ?case by simp
next
  case (Cons x xs)
  note hyp = Cons.hyps
    and asm = ‹lookup env (x # xs) = Some e›
  show ?case
  proof (cases env)
    case (Val a)
    with asm have False by simp
    then show ?thesis ..
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
```

```
      with asm Env have False by simp
      then show ?thesis ..
    next
      case (Some e')
      note es = ‹es x = Some e'›
      show ?thesis
      proof (cases xs)
        case Nil
        with Env show ?thesis by simp
      next
        case (Cons x' xs')
        from asm Env es have lookup e' xs = Some e by simp
       then have lookup (update xs (Some env') e') xs = Some env' by (rule hyp)
        with Env es Cons show ?thesis by simp
      qed
    qed
  qed
qed
```

The properties of displaced *update* operations are analogous to those of *lookup* above. There are two cases: below an undefined position *update* is absorbed altogether, and below a defined positions *update* affects subsequent *lookup* operations in the obvious way.

```
theorem update-append-none:
  assumes lookup env xs = None
  shows update (xs @ y # ys) opt env = env
  using assms
proof (induct xs arbitrary: env)
  case Nil
  then have False by simp
  then show ?case ..
next
  case (Cons x xs)
  note hyp = Cons.hyps
    and asm = ‹lookup env (x # xs) = None›
  show update ((x # xs) @ y # ys) opt env = env
  proof (cases env)
    case (Val a)
    then show ?thesis by simp
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      note es = ‹es x = None›
      show ?thesis
        by (cases xs) (simp-all add: es Env fun-upd-idem-iff)
    next
      case (Some e)
```

```
      note es = ‹es x = Some e›
      show ?thesis
      proof (cases xs)
        case Nil
        with asm Env Some have False by simp
        then show ?thesis ..
      next
        case (Cons x' xs')
        from asm Env es have lookup e xs = None by simp
        then have update (xs @ y # ys) opt e = e by (rule hyp)
        with Env es Cons show update ((x # xs) @ y # ys) opt env = env
          by (simp add: fun-upd-idem-iff)
      qed
    qed
  qed
qed

theorem update-append-some:
  assumes lookup env xs = Some e
  shows lookup (update (xs @ y # ys) opt env) xs = Some (update (y # ys) opt
e)
  using assms
proof (induct xs arbitrary: env e)
  case Nil
  then have env = e by simp
  then show ?case by simp
next
  case (Cons x xs)
  note hyp = Cons.hyps
    and asm = ‹lookup env (x # xs) = Some e›
  show lookup (update ((x # xs) @ y # ys) opt env) (x # xs) =
      Some (update (y # ys) opt e)
  proof (cases env)
    case (Val a)
    with asm have False by simp
    then show ?thesis ..
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      with asm Env have False by simp
      then show ?thesis ..
    next
      case (Some e')
      note es = ‹es x = Some e'›
      show ?thesis
      proof (cases xs)
        case Nil
```

```
      with asm Env es have e = e' by simp
      with Env es Nil show ?thesis by simp
    next
      case (Cons x' xs')
      from asm Env es have lookup e' xs = Some e by simp
      then have lookup (update (xs @ y # ys) opt e') xs =
        Some (update (y # ys) opt e) by (rule hyp)
      with Env es Cons show ?thesis by simp
    qed
  qed
 qed
qed
```

Apparently, *update* does not affect the result of subsequent *lookup* operations at independent positions, i.e. in case that the paths for *update* and *lookup* fork at a certain point.

```
theorem lookup-update-other:
  assumes neq: y ≠ (z::'c)
  shows lookup (update (xs @ z # zs) opt env) (xs @ y # ys) =
    lookup env (xs @ y # ys)
proof (induct xs arbitrary: env)
  case Nil
  show ?case
  proof (cases env)
    case Val
    then show ?thesis by simp
  next
    case Env
    show ?thesis
    proof (cases zs)
      case Nil
      with neq Env show ?thesis by simp
    next
      case Cons
      with neq Env show ?thesis by simp
    qed
  qed
next
  case (Cons x xs)
  note hyp = Cons.hyps
  show ?case
  proof (cases env)
    case Val
    then show ?thesis by simp
  next
    case (Env y es)
    show ?thesis
    proof (cases xs)
      case Nil
```

```
    show ?thesis
    proof (cases es x)
      case None
      with Env Nil show ?thesis by simp
    next
      case Some
      with neq hyp and Env Nil show ?thesis by simp
    qed
  next
    case (Cons x' xs')
    show ?thesis
    proof (cases es x)
      case None
      with Env Cons show ?thesis by simp
    next
      case Some
      with neq hyp and Env Cons show ?thesis by simp
    qed
  qed
qed
```

Equality of environments for code generation

```
lemma [code func, code func del]:
  fixes e1 e2 :: ('b::eq, 'a::eq, 'c::eq) env
  shows eq-class.eq e1 e2 ⟷ eq-class.eq e1 e2 ..

lemma eq-env-code [code func]:
  fixes x y :: 'a::eq
    and f g :: 'c::{eq, finite} ⇒ ('b::eq, 'a, 'c) env option
  shows eq-class.eq (Env x f) (Env y g) ⟷
  eq-class.eq x y ∧ (∀ z∈UNIV. case f z
   of None ⇒ (case g z
        of None ⇒ True | Some - ⇒ False)
    | Some a ⇒ (case g z
        of None ⇒ False | Some b ⇒ eq-class.eq a b)) (is ?env)
    and eq-class.eq (Val a) (Val b) ⟷ eq-class.eq a b
    and eq-class.eq (Val a) (Env y g) ⟷ False
    and eq-class.eq (Env x f) (Val b) ⟷ False
proof (unfold eq)
  have f = g ⟷ (∀ z. case f z
   of None ⇒ (case g z
        of None ⇒ True | Some - ⇒ False)
    | Some a ⇒ (case g z
        of None ⇒ False | Some b ⇒ a = b)) (is ?lhs = ?rhs)
  proof
    assume ?lhs
    then show ?rhs by (auto split: option.splits)
  next
```

```
    assume assm: ?rhs (is ∀ z. ?prop z)
    show ?lhs
    proof
      fix z
      from assm have ?prop z ..
      then show f z = g z by (auto split: option.splits)
    qed
  qed
  then show Env x f = Env y g ⟷
    x = y ∧ (∀ z∈UNIV. case f z
     of None ⇒ (case g z
          of None ⇒ True | Some - ⇒ False)
      | Some a ⇒ (case g z
          of None ⇒ False | Some b ⇒ a = b)) by simp
qed simp-all

end
```

42 Numeral-Type: Numeral Syntax for Types

```
theory Numeral-Type
  imports ATP-Linkup
begin
```

42.1 Preliminary lemmas

```
lemma inj-Inl [simp]: inj-on Inl A
  by (rule inj-onI, simp)

lemma inj-Inr [simp]: inj-on Inr A
  by (rule inj-onI, simp)

lemma inj-Some [simp]: inj-on Some A
  by (rule inj-onI, simp)

lemma card-Plus:
  [| finite A; finite B |] ==> card (A <+> B) = card A + card B
  unfolding Plus-def
  apply (subgoal-tac Inl ` A ∩ Inr ` B = {})
  apply (simp add: card-Un-disjoint card-image)
  apply fast
  done

lemma (in type-definition) univ:
  UNIV = Abs ` A
proof
  show Abs ` A ⊆ UNIV by (rule subset-UNIV)
  show UNIV ⊆ Abs ` A
```

```
  proof
    fix x :: 'b
    have x = Abs (Rep x) by (rule Rep-inverse [symmetric])
    moreover have Rep x ∈ A by (rule Rep)
    ultimately show x ∈ Abs ' A by (rule image-eqI)
  qed
qed

lemma (in type-definition) card: card (UNIV :: 'b set) = card A
  by (simp add: univ card-image inj-on-def Abs-inject)
```

42.2 Cardinalities of types

```
syntax -type-card :: type => nat ((1CARD/(1'(-'))))

translations CARD(t) => CONST card (UNIV :: t set)

typed-print-translation ⟪
let
  fun card-univ-tr' show-sorts - [Const (@{const-name UNIV}, Type(-,[T]))] =
    Syntax.const -type-card $ Syntax.term-of-typ show-sorts T;
in [(card, card-univ-tr')]
end
⟫

lemma card-unit: CARD(unit) = 1
  unfolding UNIV-unit by simp

lemma card-bool: CARD(bool) = 2
  unfolding UNIV-bool by simp

lemma card-prod: CARD('a::finite × 'b::finite) = CARD('a) * CARD('b)
 unfolding UNIV-Times-UNIV [symmetric] by (simp only: card-cartesian-product)

lemma card-sum: CARD('a::finite + 'b::finite) = CARD('a) + CARD('b)
  unfolding UNIV-Plus-UNIV [symmetric] by (simp only: finite card-Plus)

lemma card-option: CARD('a::finite option) = Suc CARD('a)
  unfolding insert-None-conv-UNIV [symmetric]
  apply (subgoal-tac (None::'a option) ∉ range Some)
  apply (simp add: finite card-image)
  apply fast
  done

lemma card-set: CARD('a::finite set) = 2 ^ CARD('a)
  unfolding Pow-UNIV [symmetric]
  by (simp only: card-Pow finite numeral-2-eq-2)
```

42.3 Numeral Types

typedef (**open**) *num0* = *UNIV* :: *nat set* **..**
typedef (**open**) *num1* = *UNIV* :: *unit set* **..**
typedef (**open**) *'a bit0* = *UNIV* :: (*bool* ∗ *'a*) *set* **..**
typedef (**open**) *'a bit1* = *UNIV* :: (*bool* ∗ *'a*) *option set* **..**

instance *num1* :: *finite*
proof
  show *finite* (*UNIV*::*num1 set*)
    unfolding *type-definition.univ* [*OF type-definition-num1*]
    using *finite* **by** (*rule finite-imageI*)
qed

instance *bit0* :: (*finite*) *finite*
proof
  show *finite* (*UNIV*::*'a bit0 set*)
    unfolding *type-definition.univ* [*OF type-definition-bit0*]
    using *finite* **by** (*rule finite-imageI*)
qed

instance *bit1* :: (*finite*) *finite*
proof
  show *finite* (*UNIV*::*'a bit1 set*)
    unfolding *type-definition.univ* [*OF type-definition-bit1*]
    using *finite* **by** (*rule finite-imageI*)
qed

lemma *card-num1*: *CARD*(*num1*) = *1*
  unfolding *type-definition.card* [*OF type-definition-num1*]
  by (*simp only*: *card-unit*)

lemma *card-bit0*: *CARD*(*'a*::*finite bit0*) = *2* ∗ *CARD*(*'a*)
  unfolding *type-definition.card* [*OF type-definition-bit0*]
  by (*simp only*: *card-prod card-bool*)

lemma *card-bit1*: *CARD*(*'a*::*finite bit1*) = *Suc* (*2* ∗ *CARD*(*'a*))
  unfolding *type-definition.card* [*OF type-definition-bit1*]
  by (*simp only*: *card-prod card-option card-bool*)

lemma *card-num0*: *CARD* (*num0*) = *0*
 by (*simp add*: *infinite-UNIV-nat card-eq-0-iff type-definition.card* [*OF type-definition-num0*])

lemmas *card-univ-simps* [*simp*] =
  card-unit
  card-bool
  card-prod
  card-sum
  card-option
  card-set

card-num1
card-bit0
card-bit1
card-num0

42.4 Syntax

syntax
-NumeralType :: num-const => type (-)
-NumeralType0 :: type (*0*)
-NumeralType1 :: type (*1*)

translations
-NumeralType1 == (type) num1
-NumeralType0 == (type) num0

parse-translation ⟪
let

val num1-const = Syntax.const Numeral-Type.num1;
val num0-const = Syntax.const Numeral-Type.num0;
val B0-const = Syntax.const Numeral-Type.bit0;
val B1-const = Syntax.const Numeral-Type.bit1;

fun mk-bintype n =
let
fun mk-bit n = if n = 0 then B0-const else B1-const;
fun bin-of n =
if n = 1 then num1-const
else if n = 0 then num0-const
else if n = ~1 then raise TERM (*negative type numeral*, [])
else
let val (*q*, *r*) = *Integer.div-mod n 2*;
in mk-bit r $ *bin-of q end*;
in bin-of n end;

fun numeral-tr (∗*-NumeralType*∗) [*Const* (*str*, -)] =
mk-bintype (*valOf* (*Int.fromString str*))
| *numeral-tr* (∗*-NumeralType*∗) *ts = raise TERM* (*numeral-tr*, *ts*);

in [(*-NumeralType*, *numeral-tr*)] *end*;
⟫

print-translation ⟪
let
fun int-of [] = *0*
| *int-of* (*b* :: *bs*) = *b* + *2* ∗ *int-of bs*;

fun bin-of (*Const* (*num0*, -)) = []

```
  | bin-of (Const (num1, -)) = [1]
  | bin-of (Const (bit0, -) $ bs) = 0 :: bin-of bs
  | bin-of (Const (bit1, -) $ bs) = 1 :: bin-of bs
  | bin-of t = raise TERM(bin-of, [t]);

fun bit-tr' b [t] =
  let
    val rev-digs = b :: bin-of t handle TERM - => raise Match
    val i = int-of rev-digs;
    val num = string-of-int (abs i);
  in
    Syntax.const -NumeralType $ Syntax.free num
  end
  | bit-tr' b - = raise Match;

in [(bit0, bit-tr' 0), (bit1, bit-tr' 1)] end;
⟩⟩
```

42.5 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 (cases CARD('a::finite) = 0)
  case False thus ?thesis by (simp del: card-0-eq)
next
  case True
  thus ?thesis by (simp add: finite)
qed

lemma one-le-card-finite [simp]:
  Suc 0 <= CARD('a::finite)
  by (simp add: less-Suc-eq-le [symmetric] zero-less-card-finite)
```

Class for cardinality "at least 2"

```
class card2 = finite +
  assumes two-le-card: 2 <= CARD('a)

lemma one-less-card: Suc 0 < CARD('a::card2)
  using two-le-card [where 'a='a] by simp

instance bit0 :: (finite) card2
  by intro-classes (simp add: one-le-card-finite)

instance bit1 :: (finite) card2
  by intro-classes (simp add: one-le-card-finite)
```

42.6 Examples

lemma $CARD(0) = 0$ **by** *simp*
lemma $CARD(17) = 17$ **by** *simp*

end

43 Option-ord: Canonical order on option type

theory *Option-ord*
imports *ATP-Linkup*
begin

instantiation *option* :: (*order*) *order*
begin

definition *less-eq-option* **where**
[*code func del*]: $x \leq y \longleftrightarrow$ (*case x of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ (*case y of None* $\Rightarrow$ *False* | *Some y* $\Rightarrow$ $x \leq y$))

definition *less-option* **where**
[*code func del*]: $x < y \longleftrightarrow$ (*case y of None* $\Rightarrow$ *False* | *Some y* $\Rightarrow$ (*case x of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ $x < y$))

lemma *less-eq-option-None* [*simp*]: *None* $\leq x$
by (*simp add*: *less-eq-option-def*)

lemma *less-eq-option-None-code* [*code*]: *None* $\leq x \longleftrightarrow$ *True*
by *simp*

lemma *less-eq-option-None-is-None*: $x \leq$ *None* $\Longrightarrow x =$ *None*
by (*cases x*) (*simp-all add*: *less-eq-option-def*)

lemma *less-eq-option-Some-None* [*simp*, *code*]: *Some x* $\leq$ *None* $\longleftrightarrow$ *False*
by (*simp add*: *less-eq-option-def*)

lemma *less-eq-option-Some* [*simp*, *code*]: *Some x* $\leq$ *Some y* $\longleftrightarrow x \leq y$
by (*simp add*: *less-eq-option-def*)

lemma *less-option-None* [*simp*, *code*]: $x <$ *None* $\longleftrightarrow$ *False*
by (*simp add*: *less-option-def*)

lemma *less-option-None-is-Some*: *None* $< x \Longrightarrow \exists z.\ x =$ *Some z*
by (*cases x*) (*simp-all add*: *less-option-def*)

lemma *less-option-None-Some* [*simp*]: *None* $<$ *Some x*

by (*simp add*: *less-option-def*)

lemma *less-option-None-Some-code* [*code*]: *None* < *Some x* ⟷ *True*
by *simp*

lemma *less-option-Some* [*simp*, *code*]: *Some x* < *Some y* ⟷ *x* < *y*
by (*simp add*: *less-option-def*)

instance by *default*
(*auto simp add*: *less-eq-option-def less-option-def split*: *option.splits*)

end

instance *option* :: (*linorder*) *linorder*
by *default* (*auto simp add*: *less-eq-option-def less-option-def split*: *option.splits*)

end

44 Order-Relation: Orders as Relations

theory *Order-Relation*
imports *ATP-Linkup Hilbert-Choice*
begin

This prelude could be moved to theory Relation:

definition *irefl r* ≡ ∀ *x*. (*x*,*x*) ∉ *r*

definition *total-on A r* ≡ ∀ *x*∈*A*.∀ *y*∈*A*. *x*≠*y* ⟶ (*x*,*y*)∈*r* ∨ (*y*,*x*)∈*r*

abbreviation *total* ≡ *total-on UNIV*

lemma *total-on-empty*[*simp*]: *total-on* {} *r*
by(*simp add*:*total-on-def*)

lemma *refl-on-converse*[*simp*]: *refl A* (*r*^−*1*) = *refl A r*
by(*auto simp add*:*refl-def*)

lemma *total-on-converse*[*simp*]: *total-on A* (*r*^−*1*) = *total-on A r*
by (*auto simp*: *total-on-def*)

lemma *irefl-diff-Id*[*simp*]: *irefl*(*r*−*Id*)
by(*simp add*:*irefl-def*)

declare [[*simp-depth-limit* = *2*]]
lemma *trans-diff-Id*: *trans r* ⟹ *antisym r* ⟹ *trans* (*r*−*Id*)
by(*simp add*: *antisym-def trans-def*) *blast*
declare [[*simp-depth-limit* = *50*]]

lemma *total-on-diff-Id*[*simp*]: *total-on A* (*r*−*Id*) = *total-on A r*
by(*simp add*: *total-on-def*)

44.1 Orders on a set

definition *preorder-on A r* ≡ *refl A r* ∧ *trans r*

definition *partial-order-on A r* ≡ *preorder-on A r* ∧ *antisym r*

definition *linear-order-on A r* ≡ *partial-order-on A r* ∧ *total-on A r*

definition *strict-linear-order-on A r* ≡ *trans r* ∧ *irrefl r* ∧ *total-on A r*

definition *well-order-on A r* ≡ *linear-order-on A r* ∧ *wf*(*r* − *Id*)

lemmas *order-on-defs* =
preorder-on-def partial-order-on-def linear-order-on-def
strict-linear-order-on-def well-order-on-def

lemma *preorder-on-empty*[*simp*]: *preorder-on* {} {}
by(*simp add*:*preorder-on-def trans-def*)

lemma *partial-order-on-empty*[*simp*]: *partial-order-on* {} {}
by(*simp add*:*partial-order-on-def*)

lemma *lnear-order-on-empty*[*simp*]: *linear-order-on* {} {}
by(*simp add*:*linear-order-on-def*)

lemma *well-order-on-empty*[*simp*]: *well-order-on* {} {}
by(*simp add*:*well-order-on-def*)

lemma *preorder-on-converse*[*simp*]: *preorder-on A* (*r*ˆ−*1*) = *preorder-on A r*
by (*simp add*:*preorder-on-def*)

lemma *partial-order-on-converse*[*simp*]:
partial-order-on A (*r*ˆ−*1*) = *partial-order-on A r*
by (*simp add*: *partial-order-on-def*)

lemma *linear-order-on-converse*[*simp*]:
linear-order-on A (*r*ˆ−*1*) = *linear-order-on A r*
by (*simp add*: *linear-order-on-def*)

lemma *strict-linear-order-on-diff-Id*:
linear-order-on A r ⟹ *strict-linear-order-on A* (*r*−*Id*)
by(*simp add*: *order-on-defs trans-diff-Id*)

44.2 Orders on the field

abbreviation *Refl r* ≡ *refl* (*Field r*) *r*

abbreviation *Preorder r* ≡ *preorder-on* (*Field r*) *r*

abbreviation *Partial-order r* ≡ *partial-order-on* (*Field r*) *r*

abbreviation *Total r* ≡ *total-on* (*Field r*) *r*

abbreviation *Linear-order r* ≡ *linear-order-on* (*Field r*) *r*

abbreviation *Well-order r* ≡ *well-order-on* (*Field r*) *r*

lemma *subset-Image-Image-iff*:
⟦ *Preorder r*; *A* ⊆ *Field r*; *B* ⊆ *Field r*⟧ ⟹
r `` *A* ⊆ *r* `` *B* ⟷ (∀ *a*∈*A*.∃ *b*∈*B*. (*b*,*a*):*r*)
apply(*auto simp add*: *subset-eq preorder-on-def refl-def Image-def*)
apply *metis*
by(*metis trans-def*)

lemma *subset-Image1-Image1-iff*:
⟦ *Preorder r*; *a* : *Field r*; *b* : *Field r*⟧ ⟹ *r* `` {*a*} ⊆ *r* `` {*b*} ⟷ (*b*,*a*):*r*
by(*simp add*:*subset-Image-Image-iff*)

lemma *Refl-antisym-eq-Image1-Image1-iff*:
⟦*Refl r*; *antisym r*; *a*:*Field r*; *b*:*Field r*⟧ ⟹ *r* `` {*a*} = *r* `` {*b*} ⟷ *a*=*b*
by(*simp add*: *expand-set-eq antisym-def refl-def*) *metis*

lemma *Partial-order-eq-Image1-Image1-iff*:
⟦*Partial-order r*; *a*:*Field r*; *b*:*Field r*⟧ ⟹ *r* `` {*a*} = *r* `` {*b*} ⟷ *a*=*b*
by(*auto simp*:*order-on-defs Refl-antisym-eq-Image1-Image1-iff*)

44.3 Orders on a type

abbreviation *strict-linear-order* ≡ *strict-linear-order-on UNIV*

abbreviation *linear-order* ≡ *linear-order-on UNIV*

abbreviation *well-order r* ≡ *well-order-on UNIV*

end

45 Permutation: Permutations

theory *Permutation*
imports *Multiset*

begin

inductive
perm :: *'a list => 'a list => bool* (- <~~> - [*50*, *50*] *50*)
where
Nil [*intro*!]: [] <~~> []
| *swap* [*intro*!]: *y # x # l* <~~> *x # y # l*
| *Cons* [*intro*!]: *xs* <~~> *ys* ==> *z # xs* <~~> *z # ys*
| *trans* [*intro*]: *xs* <~~> *ys* ==> *ys* <~~> *zs* ==> *xs* <~~> *zs*

lemma *perm-refl* [*iff*]: *l* <~~> *l*
by (*induct l*) *auto*

45.1 Some examples of rule induction on permutations

lemma *xperm-empty-imp*: [] <~~> *ys* ==> *ys* = []
by (*induct xs* == []::*'a list ys pred*: *perm*) *simp-all*

This more general theorem is easier to understand!

lemma *perm-length*: *xs* <~~> *ys* ==> *length xs* = *length ys*
by (*induct pred*: *perm*) *simp-all*

lemma *perm-empty-imp*: [] <~~> *xs* ==> *xs* = []
by (*drule perm-length*) *auto*

lemma *perm-sym*: *xs* <~~> *ys* ==> *ys* <~~> *xs*
by (*induct pred*: *perm*) *auto*

45.2 Ways of making new permutations

We can insert the head anywhere in the list.

lemma *perm-append-Cons*: *a # xs @ ys* <~~> *xs @ a # ys*
by (*induct xs*) *auto*

lemma *perm-append-swap*: *xs @ ys* <~~> *ys @ xs*
apply (*induct xs*)
apply *simp-all*
apply (*blast intro*: *perm-append-Cons*)
done

lemma *perm-append-single*: *a # xs* <~~> *xs @* [*a*]
by (*rule perm.trans* [*OF - perm-append-swap*]) *simp*

lemma *perm-rev*: *rev xs* <~~> *xs*
apply (*induct xs*)
apply *simp-all*
apply (*blast intro*!: *perm-append-single intro*: *perm-sym*)
done

lemma *perm-append1*: *xs* <~~> *ys* ==> *l* @ *xs* <~~> *l* @ *ys*
by (*induct l*) *auto*

lemma *perm-append2*: *xs* <~~> *ys* ==> *xs* @ *l* <~~> *ys* @ *l*
by (*blast intro*!: *perm-append-swap perm-append1*)

45.3 Further results

lemma *perm-empty* [*iff*]: ([] <~~> *xs*) = (*xs* = [])
by (*blast intro*: *perm-empty-imp*)

lemma *perm-empty2* [*iff*]: (*xs* <~~> []) = (*xs* = [])
apply *auto*
apply (*erule perm-sym* [*THEN perm-empty-imp*])
done

lemma *perm-sing-imp*: *ys* <~~> *xs* ==> *xs* = [*y*] ==> *ys* = [*y*]
by (*induct pred*: *perm*) *auto*

lemma *perm-sing-eq* [*iff*]: (*ys* <~~> [*y*]) = (*ys* = [*y*])
by (*blast intro*: *perm-sing-imp*)

lemma *perm-sing-eq2* [*iff*]: ([*y*] <~~> *ys*) = (*ys* = [*y*])
by (*blast dest*: *perm-sym*)

45.4 Removing elements

consts
remove :: *'a* => *'a list* => *'a list*
primrec
remove x [] = []
remove x (*y* # *ys*) = (*if x* = *y then ys else y* # *remove x ys*)

lemma *perm-remove*: *x* ∈ *set ys* ==> *ys* <~~> *x* # *remove x ys*
by (*induct ys*) *auto*

lemma *remove-commute*: *remove x* (*remove y l*) = *remove y* (*remove x l*)
by (*induct l*) *auto*

lemma *multiset-of-remove* [*simp*]:
multiset-of (*remove a x*) = *multiset-of x* − {#*a*#}
apply (*induct x*)
apply (*auto simp*: *multiset-eq-conv-count-eq*)
done

Congruence rule

lemma *perm-remove-perm*: *xs* <~~> *ys* ==> *remove z xs* <~~> *remove z ys*
by (*induct pred*: *perm*) *auto*

lemma *remove-hd* [*simp*]: *remove z* (*z # xs*) = *xs*
by *auto*

lemma *cons-perm-imp-perm*: *z # xs* <~~> *z # ys* ==> *xs* <~~> *ys*
by (*drule-tac z = z* **in** *perm-remove-perm*) *auto*

lemma *cons-perm-eq* [*iff*]: (*z#xs* <~~> *z#ys*) = (*xs* <~~> *ys*)
by (*blast intro*: *cons-perm-imp-perm*)

lemma *append-perm-imp-perm*: *zs* @ *xs* <~~> *zs* @ *ys* ==> *xs* <~~> *ys*
apply (*induct zs arbitrary*: *xs ys rule*: *rev-induct*)
apply (*simp-all* (*no-asm-use*))
apply *blast*
done

lemma *perm-append1-eq* [*iff*]: (*zs* @ *xs* <~~> *zs* @ *ys*) = (*xs* <~~> *ys*)
by (*blast intro*: *append-perm-imp-perm perm-append1*)

lemma *perm-append2-eq* [*iff*]: (*xs* @ *zs* <~~> *ys* @ *zs*) = (*xs* <~~> *ys*)
apply (*safe intro*!: *perm-append2*)
apply (*rule append-perm-imp-perm*)
apply (*rule perm-append-swap* [*THEN perm.trans*])
— the previous step helps this *blast* call succeed quickly
apply (*blast intro*: *perm-append-swap*)
done

lemma *multiset-of-eq-perm*: (*multiset-of xs = multiset-of ys*) = (*xs* <~~> *ys*)
apply (*rule iffI*)
apply (*erule-tac* [*2*] *perm.induct*, *simp-all add*: *union-ac*)
apply (*erule rev-mp*, *rule-tac x=ys* **in** *spec*)
apply (*induct-tac xs*, *auto*)
apply (*erule-tac x = remove a x* **in** *allE*, *drule sym*, *simp*)
apply (*subgoal-tac a ∈ set x*)
apply (*drule-tac z=a* **in** *perm.Cons*)
apply (*erule perm.trans*, *rule perm-sym*, *erule perm-remove*)
apply (*drule-tac f=set-of* **in** *arg-cong*, *simp*)
done

lemma *multiset-of-le-perm-append*:
(*multiset-of xs* ≤# *multiset-of ys*) = (∃*zs*. *xs* @ *zs* <~~> *ys*)
apply (*auto simp*: *multiset-of-eq-perm*[*THEN sym*] *mset-le-exists-conv*)
apply (*insert surj-multiset-of*, *drule surjD*)
apply (*blast intro*: *sym*)+
done

lemma *perm-set-eq*: *xs* <~~> *ys* ==> *set xs = set ys*
by (*metis multiset-of-eq-perm multiset-of-eq-setD*)

lemma *perm-distinct-iff*: *xs* <~~> *ys* ==> *distinct xs = distinct ys*

```
  apply (induct pred: perm)
    apply simp-all
   apply fastsimp
  apply (metis perm-set-eq)
  done

lemma eq-set-perm-remdups: set xs = set ys ==> remdups xs <~~> remdups ys
  apply (induct xs arbitrary: ys rule: length-induct)
  apply (case-tac remdups xs, simp, simp)
  apply (subgoal-tac a : set (remdups ys))
   prefer 2 apply (metis set.simps(2) insert-iff set-remdups)
  apply (drule split-list) apply(elim exE conjE)
  apply (drule-tac x=list in spec) apply(erule impE) prefer 2
   apply (drule-tac x=ysa@zs in spec) apply(erule impE) prefer 2
    apply simp
    apply (subgoal-tac a#list <~~> a#ysa@zs)
     apply (metis Cons-eq-appendI perm-append-Cons trans)
    apply (metis Cons Cons-eq-appendI distinct.simps(2)
      distinct-remdups distinct-remdups-id perm-append-swap perm-distinct-iff)
  apply (subgoal-tac set (a#list) = set (ysa@a#zs) & distinct (a#list) & distinct
(ysa@a#zs))
    apply (fastsimp simp add: insert-ident)
   apply (metis distinct-remdups set-remdups)
 apply (metis le-less-trans Suc-length-conv length-remdups-leq less-Suc-eq nat-less-le)
  done

lemma perm-remdups-iff-eq-set: remdups x <~~> remdups y = (set x = set y)
  by (metis List.set-remdups perm-set-eq eq-set-perm-remdups)

end
```

46 Primes: Primality on nat

```
theory Primes
imports GCD Parity
begin

definition
  coprime :: nat => nat => bool where
  coprime m n ⟷ (gcd (m, n) = 1)

definition
  prime :: nat ⇒ bool where
  prime p ⟷ (1 < p ∧ (∀ m. m dvd p --> m = 1 ∨ m = p))


lemma two-is-prime: prime 2
  apply (auto simp add: prime-def)
```

apply (*case-tac m*)
apply (*auto dest*!: *dvd-imp-le*)
done

lemma *prime-imp-relprime*: *prime p ==> ¬ p dvd n ==> gcd (p, n) = 1*
apply (*auto simp add*: *prime-def*)
apply (*metis One-nat-def gcd-dvd1 gcd-dvd2*)
done

This theorem leads immediately to a proof of the uniqueness of factorization. If p divides a product of primes then it is one of those primes.

lemma *prime-dvd-mult*: *prime p ==> p dvd m * n ==> p dvd m ∨ p dvd n*
by (*blast intro*: *relprime-dvd-mult prime-imp-relprime*)

lemma *prime-dvd-square*: *prime p ==> p dvd m^Suc (Suc 0) ==> p dvd m*
by (*auto dest*: *prime-dvd-mult*)

lemma *prime-dvd-power-two*: *prime p ==> p dvd m^2 ==> p dvd m*
by (*rule prime-dvd-square*) (*simp-all add*: *power2-eq-square*)

lemma *exp-eq-1*:(*x*::*nat*) ^ *n = 1 ⟷ x = 1 ∨ n = 0* **by** (*induct n*, *auto*)
lemma *exp-mono-lt*: (*x*::*nat*) ^ (*Suc n*) < *y* ^ (*Suc n*) ⟷ *x < y*
using *power-less-imp-less-base*[*of x Suc n y*] *power-strict-mono*[*of x y Suc n*]
by *auto*
lemma *exp-mono-le*: (*x*::*nat*) ^ (*Suc n*) ≤ *y* ^ (*Suc n*) ⟷ *x ≤ y*
by (*simp only*: *linorder-not-less*[*symmetric*] *exp-mono-lt*)

lemma *exp-mono-eq*: (*x*::*nat*) ^ *Suc n = y* ^ *Suc n* ⟷ *x = y*
using *power-inject-base*[*of x n y*] **by** *auto*

lemma *even-square*: **assumes** *e*: *even* (*n*::*nat*) **shows** ∃ *x*. *n* ^ *2 = 4*x*
proof−
from *e* **have** *2 dvd n* **by** *presburger*
then obtain *k* **where** *k*: *n = 2*k* **using** *dvd-def* **by** *auto*
hence *n^2 = 4* (*k^2*) **by** (*simp add*: *power2-eq-square*)
thus *?thesis* **by** *blast*
qed

lemma *odd-square*: **assumes** *e*: *odd* (*n*::*nat*) **shows** ∃ *x*. *n* ^ *2 = 4*x + 1*
proof−
from *e* **have** *np*: *n > 0* **by** *presburger*
from *e* **have** *2 dvd* (*n − 1*) **by** *presburger*
then obtain *k* **where** *n − 1 = 2*k* **using** *dvd-def* **by** *auto*
hence *k*: *n = 2*k + 1* **using** *e* **by** *presburger*
hence *n^2 = 4* (*k^2 + k*) + *1* **by** *algebra*
thus *?thesis* **by** *blast*
qed

lemma *diff-square*: $(x::nat)\ \hat{}2 - y\ \hat{}2 = (x+y)*(x - y)$
proof−
have $x \leq y \vee y \leq x$ **by** (*rule nat-le-linear*)
moreover
{assume *le*: $x \leq y$
hence $x\ \hat{}2 \leq y\hat{}2$ **by** (*simp only*: *numeral-2-eq-2 exp-mono-le Let-def*)
with *le* **have** *?thesis* **by** *simp* **}**
moreover
{assume *le*: $y \leq x$
hence *le2*: $y\ \hat{}2 \leq x\hat{}2$ **by** (*simp only*: *numeral-2-eq-2 exp-mono-le Let-def*)
from *le* **have** $\exists z.\ y + z = x$ **by** *presburger*
then obtain *z* **where** *z*: $x = y + z$ **by** *blast*
from *le2* **have** $\exists z.\ x\hat{}2 = y\hat{}2 + z$ **by** *presburger*
then obtain *z2* **where** *z2*: $x\hat{}2 = y\hat{}2 + z2$ **by** *blast*
from *z z2* **have** *?thesis* **apply** *simp* **by** *algebra* **}**
ultimately show *?thesis* **by** *blast*
qed

Elementary theory of divisibility

lemma *divides-ge*: $(a::nat)$ *dvd* $b \Longrightarrow b = 0 \vee a \leq b$ **unfolding** *dvd-def* **by** *auto*
lemma *divides-antisym*: $(x::nat)$ *dvd* $y \wedge y$ *dvd* $x \longleftrightarrow x = y$
using *dvd-anti-sym*[*of x y*] **by** *auto*

lemma *divides-add-revr*: **assumes** *da*: $(d::nat)$ *dvd* a **and** *dab*:*d dvd* $(a + b)$
shows *d dvd b*
proof−
from *da* **obtain** *k* **where** *k*:$a = d*k$ **by** (*auto simp add*: *dvd-def*)
from *dab* **obtain** k' **where** k': $a + b = d*k'$ **by** (*auto simp add*: *dvd-def*)
from *k* k' **have** $b = d*(k' - k)$ **by** (*simp add* : *diff-mult-distrib2*)
thus *?thesis* **unfolding** *dvd-def* **by** *blast*
qed

declare *nat-mult-dvd-cancel-disj*[*presburger*]
lemma *nat-mult-dvd-cancel-disj'*[*presburger*]:
$(m::nat)*k$ *dvd* $n*k \longleftrightarrow k = 0 \vee m$ *dvd* n **unfolding** *mult-commute*[*of m k*] *mult-commute*[*of n k*] **by** *presburger*

lemma *divides-mul-l*: $(a::nat)$ *dvd* b ==> $(c * a)$ *dvd* $(c * b)$
by *presburger*

lemma *divides-mul-r*: $(a::nat)$ *dvd* b ==> $(a * c)$ *dvd* $(b * c)$ **by** *presburger*
lemma *divides-cases*: $(n::nat)$ *dvd* m ==> $m = 0 \vee m = n \vee 2 * n <= m$
by (*auto simp add*: *dvd-def*)
lemma *divides-le*: m *dvd* n ==> $m <= n \vee n = (0::nat)$ **by** (*auto simp add*: *dvd-def*)

lemma *divides-div-not*: $(x::nat) = (q * n) + r \Longrightarrow 0 < r \Longrightarrow r < n$ ==> ~$(n$ *dvd* $x)$

proof(*auto simp add*: *dvd-def*)
fix *k* **assume** *H*: *0* < *r r* < *n q* * *n* + *r* = *n* * *k*
from *H*(*3*) **have** *r*: *r* = *n** (*k* −*q*) **by**(*simp add*: *diff-mult-distrib2 mult-commute*)
{assume *k* − *q* = *0* **with** *r H*(*1*) **have** *False* **by** *simp***}**
moreover
{assume *k* − *q* ≠ *0* **with** *r* **have** *r* ≥ *n* **by** *auto*
with *H*(*2*) **have** *False* **by** *simp***}**
ultimately show *False* **by** *blast*
qed
lemma *divides-exp*: (*x*::*nat*) *dvd y* ==> *x* ^ *n dvd y* ^ *n*
by (*auto simp add*: *power-mult-distrib dvd-def*)

lemma *divides-exp2*: *n* ≠ *0* ⟹ (*x*::*nat*) ^ *n dvd y* ⟹ *x dvd y*
by (*induct n* ,*auto simp add*: *dvd-def*)

fun *fact* :: *nat* ⇒ *nat* **where**
fact 0 = *1*
| *fact* (*Suc n*) = *Suc n* * *fact n*

lemma *fact-lt*: *0* < *fact n* **by**(*induct n*, *simp-all*)
lemma *fact-le*: *fact n* ≥ *1* **using** *fact-lt*[*of n*] **by** *simp*
lemma *fact-mono*: **assumes** *le*: *m* ≤ *n* **shows** *fact m* ≤ *fact n*
proof−
from *le* **have** ∃ *i*. *n* = *m*+*i* **by** *presburger*
then obtain *i* **where** *i*: *n* = *m*+*i* **by** *blast*
have *fact m* ≤ *fact* (*m* + *i*)
proof(*induct m*)
case *0* **thus** *?case* **using** *fact-le*[*of i*] **by** *simp*
next
case (*Suc m*)
have *fact* (*Suc m*) = *Suc m* * *fact m* **by** *simp*
have *th1*: *Suc m* ≤ *Suc* (*m* + *i*) **by** *simp*
from *mult-le-mono*[*of Suc m Suc* (*m*+*i*) *fact m fact* (*m*+*i*), *OF th1 Suc.hyps*]
show *?case* **by** *simp*
qed
thus *?thesis* **using** *i* **by** *simp*
qed

lemma *divides-fact*: *1* <= *p* ⟹ *p* <= *n* ==> *p dvd fact n*
proof(*induct n arbitrary*: *p*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n p*)
from *Suc.prems* **have** *p* = *Suc n* ∨ *p* ≤ *n* **by** *presburger*
moreover
{assume *p* = *Suc n* **hence** *?case* **by** (*simp only*: *fact.simps dvd-triv-left*)**}**
moreover
{assume *p* ≤ *n*
with *Suc.prems*(*1*) *Suc.hyps* **have** *th*: *p dvd fact n* **by** *simp*

from *dvd-mult*[*OF th*] **have** *?case* **by** (*simp only*: *fact.simps*) **}**
ultimately show *?case* **by** *blast*
qed

declare *dvd-triv-left*[*presburger*]
declare *dvd-triv-right*[*presburger*]
lemma *divides-rexp*:
x dvd y $\Longrightarrow$ (*x*::*nat*) *dvd* (*y*^(*Suc n*)) **by** (*simp add*: *dvd-mult2*[*of x y*])

The Bezout theorem is a bit ugly for N; it'd be easier for Z

lemma *ind-euclid*:
assumes *c*: $\forall a\ b.\ P\ (a::nat)\ b \longleftrightarrow P\ b\ a$ **and** *z*: $\forall a.\ P\ a\ 0$
and *add*: $\forall a\ b.\ P\ a\ b \longrightarrow P\ a\ (a + b)$
shows *P a b*
proof(*induct n*≡*a*+*b arbitrary*: *a b rule*: *nat-less-induct*)
fix *n a b*
assume *H*: $\forall m < n.\ \forall a\ b.\ m = a + b \longrightarrow P\ a\ b\ n = a + b$
have $a = b \vee a < b \vee b < a$ **by** *arith*
moreover {assume *eq*: *a*= *b*
from *add*[*rule-format*, *OF z*[*rule-format*, *of a*]] **have** *P a b* **using** *eq* **by** *simp***}**
moreover
{assume *lt*: $a < b$
hence $a + b - a < n \vee a = 0$ **using** *H*(*2*) **by** *arith*
moreover
{assume *a* =*0* **with** *z c* **have** *P a b* **by** *blast* **}**
moreover
{assume *ab*: $a + b - a < n$
have *th0*: $a + b - a = a + (b - a)$ **using** *lt* **by** *arith*
from *add*[*rule-format*, *OF H*(*1*)[*rule-format*, *OF ab th0*]]
have *P a b* **by** (*simp add*: *th0*[*symmetric*])**}**
ultimately have *P a b* **by** *blast***}**
moreover
{assume *lt*: $a > b$
hence $b + a - b < n \vee b = 0$ **using** *H*(*2*) **by** *arith*
moreover
{assume *b* =*0* **with** *z c* **have** *P a b* **by** *blast* **}**
moreover
{assume *ab*: $b + a - b < n$
have *th0*: $b + a - b = b + (a - b)$ **using** *lt* **by** *arith*
from *add*[*rule-format*, *OF H*(*1*)[*rule-format*, *OF ab th0*]]
have *P b a* **by** (*simp add*: *th0*[*symmetric*])
hence *P a b* **using** *c* **by** *blast* **}**
ultimately have *P a b* **by** *blast***}**
ultimately show *P a b* **by** *blast*
qed

lemma *bezout-lemma*:
assumes *ex*: $\exists (d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (a * x = b * y + d \vee b * x = a * y + d)$

shows $\exists\, d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ a + b \wedge (a * x = (a + b) * y + d \vee (a + b) * x = a * y + d)$
using *ex*
apply *clarsimp*
apply (*rule-tac* x=*d* **in** *exI*, *simp add*: *dvd-add*)
apply (*case-tac* $a * x = b * y + d$, *simp-all*)
apply (*rule-tac* x=$x + y$ **in** *exI*)
apply (*rule-tac* x=*y* **in** *exI*)
apply *algebra*
apply (*rule-tac* x=*x* **in** *exI*)
apply (*rule-tac* x=$x + y$ **in** *exI*)
apply *algebra*
done

lemma *bezout-add*: $\exists (d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (a * x = b * y + d \vee b * x = a * y + d)$
apply(*induct a b rule*: *ind-euclid*)
apply *blast*
apply *clarify*
apply (*rule-tac* x=*a* **in** *exI*, *simp add*: *dvd-add*)
apply *clarsimp*
apply (*rule-tac* x=*d* **in** *exI*)
apply (*case-tac* $a * x = b * y + d$, *simp-all add*: *dvd-add*)
apply (*rule-tac* x=$x+y$ **in** *exI*)
apply (*rule-tac* x=*y* **in** *exI*)
apply *algebra*
apply (*rule-tac* x=*x* **in** *exI*)
apply (*rule-tac* x=$x+y$ **in** *exI*)
apply *algebra*
done

lemma *bezout*: $\exists (d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (a * x - b * y = d \vee b * x - a * y = d)$
using *bezout-add*[*of a b*]
apply *clarsimp*
apply (*rule-tac* x=*d* **in** *exI*, *simp*)
apply (*rule-tac* x=*x* **in** *exI*)
apply (*rule-tac* x=*y* **in** *exI*)
apply *auto*
done

We can get a stronger version with a nonzeroness assumption.

lemma *bezout-add-strong*: **assumes** *nz*: $a \neq (0::nat)$
shows $\exists\, d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge a * x = b * y + d$
proof−
from *nz* **have** *ap*: $a > 0$ **by** *simp*
from *bezout-add*[*of a b*]
have $(\exists\, d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge a * x = b * y + d) \vee (\exists\, d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge b * x = a * y + d)$ **by** *blast*

moreover
{fix *d x y* **assume** *H*: *d dvd a d dvd b a * x = b * y + d*
from *H* **have** *?thesis* **by** *blast* **}**
moreover
{fix *d x y* **assume** *H*: *d dvd a d dvd b b * x = a * y + d*
{assume *b0*: *b = 0* **with** *H* **have** *?thesis* **by** *simp***}**
moreover
{assume *b*: *b ≠ 0* **hence** *bp*: *b > 0* **by** *simp*
from *divides-le*[*OF H*(*2*)] *b* **have** *d < b ∨ d = b* **using** *le-less* **by** *blast*
moreover
{assume *db*: *d=b*
from *prems* **have** *?thesis* **apply** *simp*
apply (*rule exI*[**where** *x = b*], *simp*)
apply (*rule exI*[**where** *x = b*])
by (*rule exI*[**where** *x = a − 1*], *simp add*: *diff-mult-distrib2*)**}**
moreover
{assume *db*: *d < b*
{assume *x=0* **hence** *?thesis* **using** *prems* **by** *simp* **}**
moreover
{assume *x0*: *x ≠ 0* **hence** *xp*: *x > 0* **by** *simp*

from *db* **have** *d ≤ b − 1* **by** *simp*
hence *d*b ≤ b*(b − 1)* **by** *simp*
with *xp mult-mono*[*of 1 x d*b b*(b − 1)*]
have *dble*: *d*b ≤ x*b*(b − 1)* **using** *bp* **by** *simp*
from *H* (*3*) **have** *d + (b − 1) * (b*x) = d + (b − 1) * (a*y + d)* **by** *simp*
hence *d + (b − 1) * a * y + (b − 1) * d = d + (b − 1) * b * x*
by (*simp only*: *mult-assoc right-distrib*)
hence *a * ((b − 1) * y) + d * (b − 1 + 1) = d + x*b*(b − 1)* **by** *algebra*
hence *a * ((b − 1) * y) = d + x*b*(b − 1) − d*b* **using** *bp* **by** *simp*
hence *a * ((b − 1) * y) = d + (x*b*(b − 1) − d*b)*
by (*simp only*: *diff-add-assoc*[*OF dble*, *of d*, *symmetric*])
hence *a * ((b − 1) * y) = b*(x*(b − 1) − d) + d*
by (*simp only*: *diff-mult-distrib2 add-commute mult-ac*)
hence *?thesis* **using** *H*(*1*,*2*)
apply −
apply (*rule exI*[**where** *x=d*], *simp*)
apply (*rule exI*[**where** *x=(b − 1) * y*])
by (*rule exI*[**where** *x=x*(b − 1) − d*], *simp*)**}**
ultimately have *?thesis* **by** *blast***}**
ultimately have *?thesis* **by** *blast***}**
ultimately have *?thesis* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

Greatest common divisor.

lemma *gcd-unique*: *d dvd a∧d dvd b ∧ (∀ e. e dvd a ∧ e dvd b ⟶ e dvd d) ⟷*

$d = gcd(a,b)$
proof(*auto*)
assume *H*: d *dvd* a d *dvd* b $\forall e.$ e *dvd* $a \wedge e$ *dvd* $b \longrightarrow e$ *dvd* d
from *H*(*3*)[*rule-format*] *gcd-dvd1*[*of a b*] *gcd-dvd2*[*of a b*]
have *th*: *gcd* (a,b) *dvd* d **by** *blast*
from *dvd-anti-sym*[*OF th gcd-greatest*[*OF H*(*1*,*2*)]] **show** $d = gcd\ (a,b)$ **by** *blast*

qed

lemma *gcd-eq*: **assumes** *H*: $\forall d.$ d *dvd* $x \wedge d$ *dvd* $y \longleftrightarrow d$ *dvd* $u \wedge d$ *dvd* v
shows *gcd* $(x,y) = gcd(u,v)$
proof−
from *H* **have** $\forall d.$ d *dvd* $x \wedge d$ *dvd* $y \longleftrightarrow d$ *dvd gcd* (u,v) **by** *simp*
with *gcd-unique*[*of gcd*(u,v) x y] **show** *?thesis* **by** *auto*
qed

lemma *bezout-gcd*: $\exists x\ y.\ a * x - b * y = gcd(a,b) \vee b * x - a * y = gcd(a,b)$
proof−
let *?g* = *gcd* (a,b)
from *bezout*[*of a b*] **obtain** d x y **where** *d*: d *dvd* a d *dvd* b $a * x - b * y = d$
$\vee\ b * x - a * y = d$ **by** *blast*
from *d*(*1*,*2*) **have** d *dvd* *?g* **by** *simp*
then obtain k **where** *k*: *?g* = $d*k$ **unfolding** *dvd-def* **by** *blast*
from *d*(*3*) **have** $(a * x - b * y)*k = d*k \vee (b * x - a * y)*k = d*k$ **by** *blast*
hence $a * x * k - b * y*k = d*k \vee b * x * k - a * y*k = d*k$
by (*simp only*: *diff-mult-distrib*)
hence $a * (x * k) - b * (y*k) =$ *?g* $\vee\ b * (x * k) - a * (y*k) =$ *?g*
by (*simp add*: *k mult-assoc*)
thus *?thesis* **by** *blast*
qed

lemma *bezout-gcd-strong*: **assumes** *a*: $a \neq 0$
shows $\exists x\ y.\ a * x = b * y + gcd(a,b)$
proof−
let *?g* = *gcd* (a,b)
from *bezout-add-strong*[*OF a*, *of b*]
obtain d x y **where** *d*: d *dvd* a d *dvd* b $a * x = b * y + d$ **by** *blast*
from *d*(*1*,*2*) **have** d *dvd* *?g* **by** *simp*
then obtain k **where** *k*: *?g* = $d*k$ **unfolding** *dvd-def* **by** *blast*
from *d*(*3*) **have** $a * x * k = (b * y + d) * k$ **by** *auto*
hence $a * (x * k) = b * (y*k) +$ *?g* **by** (*algebra add*: *k*)
thus *?thesis* **by** *blast*
qed

lemma *gcd-mult-distrib*: $gcd(a * c,\ b * c) = c * gcd(a,b)$
by(*simp add*: *gcd-mult-distrib2 mult-commute*)

lemma *gcd-bezout*: $(\exists x\ y.\ a * x - b * y = d \vee b * x - a * y = d) \longleftrightarrow gcd(a,b)$
dvd d

```
  (is ?lhs ⟷ ?rhs)
proof−
  let ?g = gcd (a,b)
  {assume H: ?rhs then obtain k where k: d = ?g*k unfolding dvd-def by
blast
    from bezout-gcd[of a b] obtain x y where xy: a * x − b * y = ?g ∨ b * x −
a * y = ?g
      by blast
    hence (a * x − b * y)*k = ?g*k ∨ (b * x − a * y)*k = ?g*k by auto
    hence a * x*k − b * y*k = ?g*k ∨ b * x * k − a * y*k = ?g*k
      by (simp only: diff-mult-distrib)
    hence a * (x*k) − b * (y*k) = d ∨ b * (x * k) − a * (y*k) = d
      by (simp add: k[symmetric] mult-assoc)
    hence ?lhs by blast}
  moreover
  {fix x y assume H: a * x − b * y = d ∨ b * x − a * y = d
    have dv: ?g dvd a*x ?g dvd b * y ?g dvd b*x ?g dvd a * y
      using dvd-mult2[OF gcd-dvd1[of a b]] dvd-mult2[OF gcd-dvd2[of a b]] by
simp-all
    from dvd-diff[OF dv(1,2)] dvd-diff[OF dv(3,4)] H
    have ?rhs by auto}
  ultimately show ?thesis by blast
qed

lemma gcd-bezout-sum: assumes H:a * x + b * y = d shows gcd(a,b) dvd d
proof−
  let ?g = gcd (a,b)
    have dv: ?g dvd a*x ?g dvd b * y
      using dvd-mult2[OF gcd-dvd1[of a b]] dvd-mult2[OF gcd-dvd2[of a b]] by
simp-all
    from dvd-add[OF dv] H
    show ?thesis by auto
qed

lemma gcd-mult': gcd(b,a * b) = b
by (simp add: gcd-mult mult-commute[of a b])

lemma gcd-add: gcd(a + b,b) = gcd(a,b) gcd(b + a,b) = gcd(a,b) gcd(a,a + b)
= gcd(a,b) gcd(a,b + a) = gcd(a,b)
apply (simp-all add: gcd-add1)
by (simp add: gcd-commute gcd-add1)

lemma gcd-sub: b <= a ==> gcd(a − b,b) = gcd(a,b) a <= b ==> gcd(a,b −
a) = gcd(a,b)
proof−
  {fix a b assume H: b ≤ (a::nat)
    hence th: a − b + b = a by arith
    from gcd-add(1)[of a − b b] th  have gcd(a − b,b) = gcd(a,b) by simp}
  note th = this
```

{
assume *ab*: $b \leq a$
from *th*[*OF ab*] **show** *gcd* $(a - b, b) =$ *gcd* (a, b) **by** *blast*
next
assume *ab*: $a \leq b$
from *th*[*OF ab*] **show** *gcd* $(a,b - a) =$ *gcd* (a, b)
by (*simp add*: *gcd-commute*)**}**
qed

Coprimality

lemma *coprime*: *coprime a b* $\longleftrightarrow$ $(\forall d.$ *d dvd a* $\wedge$ *d dvd b* $\longleftrightarrow$ *d = 1*$)$
using *gcd-unique*[*of 1 a b*, *simplified*] **by** (*auto simp add*: *coprime-def*)
lemma *coprime-commute*: *coprime a b* $\longleftrightarrow$ *coprime b a* **by** (*simp add*: *coprime-def gcd-commute*)

lemma *coprime-bezout*: *coprime a b* $\longleftrightarrow$ $(\exists x\ y.$ *a* $*$ *x* $-$ *b* $*$ *y* $=$ *1* $\vee$ *b* $*$ *x* $-$ *a* $*$ *y* $=$ *1*$)$
using *coprime-def gcd-bezout* **by** *auto*

lemma *coprime-divprod*: *d dvd a* $*$ *b* $\Longrightarrow$ *coprime d a* $\Longrightarrow$ *d dvd b*
using *relprime-dvd-mult-iff*[*of d a b*] **by** (*auto simp add*: *coprime-def mult-commute*)

lemma *coprime-1*[*simp*]: *coprime a 1* **by** (*simp add*: *coprime-def*)
lemma *coprime-1'*[*simp*]: *coprime 1 a* **by** (*simp add*: *coprime-def*)
lemma *coprime-Suc0*[*simp*]: *coprime a* (*Suc 0*) **by** (*simp add*: *coprime-def*)
lemma *coprime-Suc0'*[*simp*]: *coprime* (*Suc 0*) *a* **by** (*simp add*: *coprime-def*)

lemma *gcd-coprime*:
assumes *z*: *gcd*(*a*,*b*) $\neq$ *0* **and** *a*: *a* $=$ *a'* $*$ *gcd*(*a*,*b*) **and** *b*: *b* $=$ *b'* $*$ *gcd*(*a*,*b*)
shows *coprime a' b'*
proof$-$
let *?g* $=$ *gcd*(*a*,*b*)
{assume *bz*: *a* $=$ *0* **from** *b bz z a* **have** *?thesis* **by** (*simp add*: *gcd-zero coprime-def*)**}**
moreover
{assume *az*: *a*$\neq$ *0*
from *z* **have** *z'*: *?g* $>$ *0* **by** *simp*
from *bezout-gcd-strong*[*OF az*, *of b*]
obtain *x y* **where** *xy*: *a*$*$*x* $=$ *b*$*$*y* $+$ *?g* **by** *blast*
from *xy a b* **have** *?g* $*$ *a'*$*$*x* $=$ *?g* $*$ (*b'*$*$*y* $+$ *1*) **by** (*simp add*: *ring-simps*)
hence *?g* $*$ (*a'*$*$*x*) $=$ *?g* $*$ (*b'*$*$*y* $+$ *1*) **by** (*simp add*: *mult-assoc*)
hence *a'*$*$*x* $=$ (*b'*$*$*y* $+$ *1*)
by (*simp only*: *nat-mult-eq-cancel1*[*OF z'*])
hence *a'*$*$*x* $-$ *b'*$*$*y* $=$ *1* **by** *simp*
with *coprime-bezout*[*of a' b'*] **have** *?thesis* **by** *auto***}**
ultimately show *?thesis* **by** *blast*
qed
lemma *coprime-0*: *coprime d 0* $\longleftrightarrow$ *d* $=$ *1* **by** (*simp add*: *coprime-def*)
lemma *coprime-mul*: **assumes** *da*: *coprime d a* **and** *db*: *coprime d b*
shows *coprime d* (*a* $*$ *b*)

proof−
from *da* **have** *th*: *gcd*(*a*, *d*) = *1* **by** (*simp add*: *coprime-def gcd-commute*)
from *gcd-mult-cancel*[*of a d b*, *OF th*] *db*[*unfolded coprime-def*] **have** *gcd* (*d*, *a*∗*b*) = *1*
by (*simp add*: *gcd-commute*)
thus *?thesis* **unfolding** *coprime-def* **.**
qed
lemma *coprime-lmul2*: **assumes** *dab*: *coprime d* (*a* ∗ *b*) **shows** *coprime d b*
using *prems* **unfolding** *coprime-bezout*
apply *clarsimp*
apply (*case-tac d* ∗ *x* − *a* ∗ *b* ∗ *y* = *Suc 0* , *simp-all*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*rule-tac x*=*a*∗*y* **in** *exI*)
apply (*simp add*: *mult-ac*)
apply (*rule-tac x*=*a*∗*x* **in** *exI*)
apply (*rule-tac x*=*y* **in** *exI*)
apply (*simp add*: *mult-ac*)
done

lemma *coprime-rmul2*: *coprime d* (*a* ∗ *b*) ⟹ *coprime d a*
unfolding *coprime-bezout*
apply *clarsimp*
apply (*case-tac d* ∗ *x* − *a* ∗ *b* ∗ *y* = *Suc 0* , *simp-all*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*rule-tac x*=*b*∗*y* **in** *exI*)
apply (*simp add*: *mult-ac*)
apply (*rule-tac x*=*b*∗*x* **in** *exI*)
apply (*rule-tac x*=*y* **in** *exI*)
apply (*simp add*: *mult-ac*)
done
lemma *coprime-mul-eq*: *coprime d* (*a* ∗ *b*) ⟷ *coprime d a* ∧ *coprime d b*
using *coprime-rmul2*[*of d a b*] *coprime-lmul2*[*of d a b*] *coprime-mul*[*of d a b*]
by *blast*

lemma *gcd-coprime-exists*:
assumes *nz*: *gcd*(*a*,*b*) ≠ *0*
shows ∃ *a′ b′*. *a* = *a′* ∗ *gcd*(*a*,*b*) ∧ *b* = *b′* ∗ *gcd*(*a*,*b*) ∧ *coprime a′ b′*
proof−
let *?g* = *gcd* (*a*,*b*)
from *gcd-dvd1*[*of a b*] *gcd-dvd2*[*of a b*]
obtain *a′ b′* **where** *a* = *?g*∗*a′* *b* = *?g*∗*b′* **unfolding** *dvd-def* **by** *blast*
hence *ab′*: *a* = *a′*∗*?g b* = *b′*∗*?g* **by** *algebra*+
from *ab′ gcd-coprime*[*OF nz ab′*] **show** *?thesis* **by** *blast*
qed

lemma *coprime-exp*: *coprime d a* ==> *coprime d* (*a*^*n*)
by(*induct n*, *simp-all add*: *coprime-mul*)

lemma *coprime-exp-imp*: *coprime a b* ==> *coprime* (*a* ^*n*) (*b* ^*n*)

by (*induct n*, *simp-all add*: *coprime-mul-eq coprime-commute coprime-exp*)
lemma *coprime-refl*[*simp*]: *coprime n n* ⟷ *n = 1* **by** (*simp add*: *coprime-def*)
lemma *coprime-plus1*[*simp*]: *coprime* (*n + 1*) *n*
apply (*simp add*: *coprime-bezout*)
apply (*rule exI*[**where** *x=1*])
apply (*rule exI*[**where** *x=1*])
apply *simp*
done
lemma *coprime-minus1*: *n ≠ 0 ==> coprime* (*n − 1*) *n*
using *coprime-plus1*[*of n − 1*] *coprime-commute*[*of n − 1 n*] **by** *auto*

lemma *bezout-gcd-pow*: ∃ *x y*. *a ^ n * x − b ^ n * y = gcd*(*a*,*b*) *^ n* ∨ *b ^ n * x − a ^ n * y = gcd*(*a*,*b*) *^ n*
proof−
let *?g = gcd* (*a*,*b*)
{assume *z*: *?g = 0* **hence** *?thesis*
apply (*cases n*, *simp*)
apply *arith*
apply (*simp only*: *z power-0-Suc*)
apply (*rule exI*[**where** *x=0*])
apply (*rule exI*[**where** *x=0*])
by *simp***}**
moreover
{assume *z*: *?g ≠ 0*
from *gcd-dvd1*[*of a b*] *gcd-dvd2*[*of a b*] **obtain** *a′ b′* **where**
ab′: *a = a′*?g b = b′*?g* **unfolding** *dvd-def* **by** (*auto simp add*: *mult-ac*)
hence *ab″*: *?g*a′ = a ?g * b′ = b* **by** *algebra+*
from *coprime-exp-imp*[*OF gcd-coprime*[*OF z ab′*], *unfolded coprime-bezout*, *of n*]
obtain *x y* **where** *a′^n * x − b′^n * y = 1 ∨ b′^n * x − a′^n * y = 1* **by** *blast*
hence *?g^n * (a′^n * x − b′^n * y) = ?g^n ∨ ?g^n*(b′^n * x − a′^n * y) = ?g^n*
using *z* **by** *auto*
then have *a^n * x − b^n * y = ?g^n ∨ b^n * x − a^n * y = ?g^n*
using *z ab″* **by** (*simp only*: *power-mult-distrib*[*symmetric*]
diff-mult-distrib2 mult-assoc[*symmetric*])
hence *?thesis* **by** *blast* **}**
ultimately show *?thesis* **by** *blast*
qed
lemma *gcd-exp*: *gcd* (*a^n*, *b^n*) = *gcd*(*a*,*b*) *^n*
proof−
let *?g = gcd*(*a^n*,*b^n*)
let *?gn = gcd*(*a*,*b*) *^n*
{fix *e* **assume** *H*: *e dvd a^n e dvd b^n*
from *bezout-gcd-pow*[*of a n b*] **obtain** *x y*
where *xy*: *a ^ n * x − b ^ n * y = ?gn ∨ b ^ n * x − a ^ n * y = ?gn* **by** *blast*
from *dvd-diff* [*OF dvd-mult2*[*OF H*(*1*), *of x*] *dvd-mult2*[*OF H*(*2*), *of y*]]

```
    dvd-diff [OF dvd-mult2[OF H(2), of x] dvd-mult2[OF H(1), of y]] xy
   have e dvd ?gn by (cases a ^ n * x - b ^ n * y = gcd (a, b) ^ n, simp-all)}
  hence th: ∀e. e dvd a^n ∧ e dvd b^n ⟶ e dvd ?gn by blast
  from divides-exp[OF gcd-dvd1[of a b], of n] divides-exp[OF gcd-dvd2[of a b], of
n] th
   gcd-unique have ?gn = ?g by blast thus ?thesis by simp
qed

lemma coprime-exp2: coprime (a ^ Suc n) (b^ Suc n) ⟷ coprime a b
by (simp only: coprime-def gcd-exp exp-eq-1) simp

lemma division-decomp: assumes dc: (a::nat) dvd b * c
  shows ∃b' c'. a = b' * c' ∧ b' dvd b ∧ c' dvd c
proof−
  let ?g = gcd (a,b)
  {assume ?g = 0 with dc have ?thesis apply (simp add: gcd-zero)
     apply (rule exI[where x=0])
     by (rule exI[where x=c], simp)}
  moreover
  {assume z: ?g ≠ 0
   from gcd-coprime-exists[OF z]
   obtain a' b' where ab': a = a' * ?g b = b' * ?g coprime a' b' by blast
   from gcd-dvd2[of a b] have thb: ?g dvd b .
   from ab'(1) have a' dvd a  unfolding dvd-def by blast
   with dc have th0: a' dvd b*c using dvd-trans[of a' a b*c] by simp
   from dc ab'(1,2) have a'*?g dvd (b'*?g) *c by auto
   hence ?g*a' dvd ?g * (b' * c) by (simp add: mult-assoc)
   with z have th-1: a' dvd b'*c by simp
   from coprime-divprod[OF th-1 ab'(3)] have thc: a' dvd c .
   from ab' have a = ?g*a' by algebra
   with thb thc have ?thesis by blast }
  ultimately show ?thesis by blast
qed

lemma nat-power-eq-0-iff: (m::nat) ^ n = 0 ⟷ n ≠ 0 ∧ m = 0 by (induct n,
auto)

lemma divides-rev: assumes ab: (a::nat) ^ n dvd b ^n and n:n ≠ 0 shows a dvd
b
proof−
  let ?g = gcd (a,b)
  from n obtain m where m: n = Suc m by (cases n, simp-all)
  {assume ?g = 0 with ab n have ?thesis by (simp add: gcd-zero)}
  moreover
  {assume z: ?g ≠ 0
   hence zn: ?g ^ n ≠ 0 using n by (simp add: neq0-conv)
   from gcd-coprime-exists[OF z]
   obtain a' b' where ab': a = a' * ?g b = b' * ?g coprime a' b' by blast
  from ab have (a' * ?g) ^ n dvd (b' * ?g) ^n by (simp add: ab'(1,2)[symmetric])
```

hence *?g^n*a'^n dvd ?g^n *b'^n* **by** (*simp only*: *power-mult-distrib mult-commute*)
with *zn z n* **have** *th0*:*a'^n dvd b'^n* **by** (*auto simp add*: *nat-power-eq-0-iff*)
have *a' dvd a'^n* **by** (*simp add*: *m*)
with *th0* **have** *a' dvd b'^n* **using** *dvd-trans*[*of a' a'^n b'^n*] **by** *simp*
hence *th1*: *a' dvd b'^m * b'* **by** (*simp add*: *m mult-commute*)
from *coprime-divprod*[*OF th1 coprime-exp*[*OF ab'*(*3*), *of m*]]
have *a' dvd b'* **.**
hence *a'*?g dvd b'*?g* **by** *simp*
with *ab'*(*1*,*2*) **have** *?thesis* **by** *simp* **}**
ultimately show *?thesis* **by** *blast*
qed

lemma *divides-mul*: **assumes** *mr*: *m dvd r* **and** *nr*: *n dvd r* **and** *mn*:*coprime m n*
shows *m * n dvd r*
proof−
from *mr nr* **obtain** *m' n'* **where** *m'*: *r = m*m'* **and** *n'*: *r = n*n'*
unfolding *dvd-def* **by** *blast*
from *mr n'* **have** *m dvd n'*n* **by** (*simp add*: *mult-commute*)
hence *m dvd n'* **using** *relprime-dvd-mult-iff*[*OF mn*[*unfolded coprime-def*]] **by** *simp*
then obtain *k* **where** *k*: *n' = m*k* **unfolding** *dvd-def* **by** *blast*
from *n' k* **show** *?thesis* **unfolding** *dvd-def* **by** *auto*
qed

A binary form of the Chinese Remainder Theorem.

lemma *chinese-remainder*: **assumes** *ab*: *coprime a b* **and** *a*:*a ≠ 0* **and** *b*:*b ≠ 0*
shows ∃*x q1 q2*. *x = u + q1 * a ∧ x = v + q2 * b*
proof−
from *bezout-add-strong*[*OF a*, *of b*] *bezout-add-strong*[*OF b*, *of a*]
obtain *d1 x1 y1 d2 x2 y2* **where** *dxy1*: *d1 dvd a d1 dvd b a * x1 = b * y1 + d1*
and *dxy2*: *d2 dvd b d2 dvd a b * x2 = a * y2 + d2* **by** *blast*
from *gcd-unique*[*of 1 a b*, *simplified ab*[*unfolded coprime-def*], *simplified*]
dxy1(*1*,*2*) *dxy2*(*1*,*2*) **have** *d12*: *d1 = 1 d2 =1* **by** *auto*
let *?x = v * a * x1 + u * b * x2*
let *?q1 = v * x1 + u * y2*
let *?q2 = v * y1 + u * x2*
from *dxy2*(*3*)[*simplified d12*] *dxy1*(*3*)[*simplified d12*]
have *?x = u + ?q1 * a ?x = v + ?q2 * b* **by** *algebra*+
thus *?thesis* **by** *blast*
qed

Primality

A few useful theorems about primes

lemma *prime-0*[*simp*]: ~*prime 0* **by** (*simp add*: *prime-def*)
lemma *prime-1*[*simp*]: ~ *prime 1* **by** (*simp add*: *prime-def*)
lemma *prime-Suc0*[*simp*]: ~ *prime* (*Suc 0*) **by** (*simp add*: *prime-def*)

lemma *prime-ge-2*: *prime p* ==> *p ≥ 2* **by** (*simp add*: *prime-def*)

lemma *prime-factor*: **assumes** *n*: $n \neq 1$ **shows** $\exists\ p.\ prime\ p \wedge p\ dvd\ n$
using *n*
proof(*induct n rule*: *nat-less-induct*)
fix *n*
assume *H*: $\forall\, m{<}n.\ m \neq 1 \longrightarrow (\exists\, p.\ prime\ p \wedge p\ dvd\ m)\ n \neq 1$
let *?ths* = $\exists\, p.\ prime\ p \wedge p\ dvd\ n$
{assume *n=0* **hence** *?ths* **using** *two-is-prime* **by** *auto***}**
moreover
{assume *nz*: *n≠0*
{assume *prime n* **hence** *?ths* **by** − (*rule exI*[**where** *x=n*], *simp*)**}**
moreover
{assume *n*: ¬ *prime n*
with *nz H*(*2*)
obtain *k* **where** *k*:*k dvd n k* ≠ *1 k* ≠ *n* **by** (*auto simp add*: *prime-def*)
from *dvd-imp-le*[*OF k*(*1*)] *nz k*(*3*) **have** *kn*: *k* < *n* **by** *simp*
from *H*(*1*)[*rule-format*, *OF kn k*(*2*)] **obtain** *p* **where** *p*: *prime p p dvd k* **by** *blast*
from *dvd-trans*[*OF p*(*2*) *k*(*1*)] *p*(*1*) **have** *?ths* **by** *blast***}**
ultimately have *?ths* **by** *blast***}**
ultimately show *?ths* **by** *blast*
qed

lemma *prime-factor-lt*: **assumes** *p*: *prime p* **and** *n*: *n* ≠ *0* **and** *npm*:*n* = *p* ∗ *m*
shows *m* < *n*
proof−
{assume *m=0* **with** *n* **have** *?thesis* **by** *simp***}**
moreover
{assume *m*: *m* ≠ *0*
from *npm* **have** *mn*: *m dvd n* **unfolding** *dvd-def* **by** *auto*
from *npm m* **have** *n* ≠ *m* **using** *p* **by** *auto*
with *dvd-imp-le*[*OF mn*] *n* **have** *?thesis* **by** *simp***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *euclid-bound*: $\exists\, p.\ prime\ p \wedge n < p \wedge\ p <= Suc\ (fact\ n)$
proof−
have *f1*: *fact n* + *1* ≠ *1* **using** *fact-le*[*of n*] **by** *arith*
from *prime-factor*[*OF f1*] **obtain** *p* **where** *p*: *prime p p dvd fact n* + *1* **by** *blast*
from *dvd-imp-le*[*OF p*(*2*)] **have** *pfn*: *p* ≤ *fact n* + *1* **by** *simp*
{assume *np*: *p* ≤ *n*
from *p*(*1*) **have** *p1*: *p* ≥ *1* **by** (*cases p*, *simp-all*)
from *divides-fact*[*OF p1 np*] **have** *pfn'*: *p dvd fact n* **.**
from *divides-add-revr*[*OF pfn' p*(*2*)] *p*(*1*) **have** *False* **by** *simp***}**
hence *n* < *p* **by** *arith*
with *p*(*1*) *pfn* **show** *?thesis* **by** *auto*
qed

lemma *euclid*: $\exists\, p.\ prime\ p \wedge p > n$ **using** *euclid-bound* **by** *auto*
lemma *primes-infinite*: ¬ (*finite* {*p*. *prime p*})

proof (*auto simp add*: *finite-conv-nat-seg-image*)
fix *n f*
assume *H*: *Collect prime* = *f* ‘ {*i*. *i* < (*n*::*nat*)}
let *?P* = *Collect prime*
let *?m* = *Max ?P*
have *P0*: *?P* ≠ {} **using** *two-is-prime* **by** *auto*
from *H* **have** *fP*: *finite ?P* **using** *finite-conv-nat-seg-image* **by** *blast*
from *Max-in* [*OF fP P0*] **have** *?m* ∈ *?P* **.**
from *Max-ge* [*OF fP*] **have** *contr*: ∀ *p*. *prime p* ⟶ *p* ≤ *?m* **by** *blast*
from *euclid* [*of ?m*] **obtain** *q* **where** *q*: *prime q q* > *?m* **by** *blast*
with *contr* **show** *False* **by** *auto*
qed

lemma *coprime-prime*: **assumes** *ab*: *coprime a b*
shows ~(*prime p* ∧ *p dvd a* ∧ *p dvd b*)
proof
assume *prime p* ∧ *p dvd a* ∧ *p dvd b*
thus *False* **using** *ab gcd-greatest*[*of p a b*] **by** (*simp add*: *coprime-def*)
qed
lemma *coprime-prime-eq*: *coprime a b* ⟷ (∀ *p*. ~(*prime p* ∧ *p dvd a* ∧ *p dvd b*))

(**is** *?lhs* = *?rhs*)
proof−
{assume *?lhs* **with** *coprime-prime* **have** *?rhs* **by** *blast***}**
moreover
{assume *r*: *?rhs* **and** *c*: ¬ *?lhs*
then obtain *g* **where** *g*: *g*≠*1 g dvd a g dvd b* **unfolding** *coprime-def* **by** *blast*
from *prime-factor*[*OF g*(*1*)] **obtain** *p* **where** *p*: *prime p p dvd g* **by** *blast*
from *dvd-trans* [*OF p*(*2*) *g*(*2*)] *dvd-trans* [*OF p*(*2*) *g*(*3*)]
have *p dvd a p dvd b* **. with** *p*(*1*) *r* **have** *False* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *prime-coprime*: **assumes** *p*: *prime p*
shows *n* = *1* ∨ *p dvd n* ∨ *coprime p n*
using *p prime-imp-relprime*[*of p n*] **by** (*auto simp add*: *coprime-def*)

lemma *prime-coprime-strong*: *prime p* ⟹ *p dvd n* ∨ *coprime p n*
using *prime-coprime*[*of p n*] **by** *auto*

declare *coprime-0*[*simp*]

lemma *coprime-0'*[*simp*]: *coprime 0 d* ⟷ *d* = *1* **by** (*simp add*: *coprime-commute*[*of 0 d*])
lemma *coprime-bezout-strong*: **assumes** *ab*: *coprime a b* **and** *b*: *b* ≠ *1*
shows ∃ *x y*. *a* ∗ *x* = *b* ∗ *y* + *1*
proof−
from *ab b* **have** *az*: *a* ≠ *0* **by** − (*rule ccontr*, *auto*)
from *bezout-gcd-strong*[*OF az*, *of b*] *ab*[*unfolded coprime-def*]

```
  show ?thesis by auto
qed

lemma bezout-prime: assumes p: prime p  and pa: ¬ p dvd a
  shows ∃ x y. a*x = p*y + 1
proof−
  from p have p1: p ≠ 1 using prime-1 by blast
  from prime-coprime[OF p, of a] p1 pa have ap: coprime a p
    by (auto simp add: coprime-commute)
  from coprime-bezout-strong[OF ap p1] show ?thesis .
qed
lemma prime-divprod: assumes p: prime p and pab: p dvd a*b
  shows p dvd a ∨ p dvd b
proof−
  {assume a=1 hence ?thesis using pab by simp }
  moreover
  {assume p dvd a hence ?thesis by blast}
  moreover
  {assume pa: coprime p a from coprime-divprod[OF pab pa]  have ?thesis .. }
  ultimately show ?thesis using prime-coprime[OF p, of a] by blast
qed

lemma prime-divprod-eq: assumes p: prime p
  shows p dvd a*b ⟷ p dvd a ∨ p dvd b
using p prime-divprod dvd-mult dvd-mult2 by auto

lemma prime-divexp: assumes p:prime p and px: p dvd x^n
  shows p dvd x
using px
proof(induct n)
  case 0 thus ?case by simp
next
  case (Suc n)
  hence th: p dvd x*x^n by simp
  {assume H: p dvd x^n
    from Suc.hyps[OF H] have ?case .}
  with prime-divprod[OF p th] show ?case by blast
qed

lemma prime-divexp-n: prime p ⟹ p dvd x^n ⟹ p^n dvd x^n
  using prime-divexp[of p x n] divides-exp[of p x n] by blast

lemma coprime-prime-dvd-ex: assumes xy: ¬coprime x y
  shows ∃ p. prime p ∧ p dvd x ∧ p dvd y
proof−
  from xy[unfolded coprime-def] obtain g where g: g ≠ 1 g dvd x g dvd y
    by blast
  from prime-factor[OF g(1)] obtain p where p: prime p p dvd g by blast
  from g(2,3) dvd-trans[OF p(2)] p(1) show ?thesis by auto
```

qed
lemma *coprime-sos*: **assumes** *xy*: *coprime x y*
shows *coprime* (*x* ∗ *y*) (*x^2* + *y^2*)
proof−
{assume *c*: ¬ *coprime* (*x* ∗ *y*) (*x^2* + *y^2*)
from *coprime-prime-dvd-ex*[*OF c*] **obtain** *p*
where *p*: *prime p p dvd x∗y p dvd x^2* + *y^2* **by** *blast*
{assume *px*: *p dvd x*
from *dvd-mult*[*OF px*, *of x*] *p*(*3*) **have** *p dvd y^2*
unfolding *dvd-def*
apply (*auto simp add*: *power2-eq-square*)
apply (*rule-tac x*= *ka* − *k* **in** *exI*)
by (*simp add*: *diff-mult-distrib2*)
with *prime-divexp*[*OF p*(*1*), *of y 2*] **have** *py*: *p dvd y* **.**
from *p*(*1*) *px py xy*[*unfolded coprime*, *rule-format*, *of p*] *prime-1*
have *False* **by** *simp* **}**
moreover
{assume *py*: *p dvd y*
from *dvd-mult*[*OF py*, *of y*] *p*(*3*) **have** *p dvd x^2*
unfolding *dvd-def*
apply (*auto simp add*: *power2-eq-square*)
apply (*rule-tac x*= *ka* − *k* **in** *exI*)
by (*simp add*: *diff-mult-distrib2*)
with *prime-divexp*[*OF p*(*1*), *of x 2*] **have** *px*: *p dvd x* **.**
from *p*(*1*) *px py xy*[*unfolded coprime*, *rule-format*, *of p*] *prime-1*
have *False* **by** *simp* **}**
ultimately have *False* **using** *prime-divprod*[*OF p*(*1*,*2*)] **by** *blast***}**
thus *?thesis* **by** *blast*
qed

lemma *distinct-prime-coprime*: *prime p* ⟹ *prime q* ⟹ *p* ≠ *q* ⟹ *coprime p q*
unfolding *prime-def coprime-prime-eq* **by** *blast*

lemma *prime-coprime-lt*: **assumes** *p*: *prime p* **and** *x*: *0* < *x* **and** *xp*: *x* < *p*
shows *coprime x p*
proof−
{assume *c*: ¬ *coprime x p*
then obtain *g* **where** *g*: *g* ≠ *1 g dvd x g dvd p* **unfolding** *coprime-def* **by** *blast*
from *dvd-imp-le*[*OF g*(*2*)] *x xp* **have** *gp*: *g* < *p* **by** *arith*
from *g*(*2*) *x* **have** *g* ≠ *0* **by** − (*rule ccontr*, *simp*)
with *g gp p*[*unfolded prime-def*] **have** *False* **by** *blast***}**
thus *?thesis* **by** *blast*
qed

lemma *even-dvd*[*simp*]: *even* (*n*::*nat*) ⟷ *2 dvd n* **by** *presburger*
lemma *prime-odd*: *prime p* ⟹ *p* = *2* ∨ *odd p* **unfolding** *prime-def* **by** *auto*

One property of coprimality is easier to prove via prime factors.

lemma *prime-divprod-pow*:
assumes *p*: *prime p* **and** *ab*: *coprime a b* **and** *pab*: *p ^ n dvd a * b*
shows *p ^ n dvd a ∨ p ^ n dvd b*
proof −
{assume *n = 0 ∨ a = 1 ∨ b = 1* **with** *pab* **have** *?thesis*
apply (*cases n=0*, *simp-all*)
apply (*cases a=1*, *simp-all*) **done}**
moreover
{assume *n*: *n ≠ 0* **and** *a*: *a≠1* **and** *b*: *b≠1*
then obtain *m* **where** *m*: *n = Suc m* **by** (*cases n*, *auto*)
from *divides-exp2*[*OF n pab*] **have** *pab'*: *p dvd a*b* **.**
from *prime-divprod*[*OF p pab'*]
have *p dvd a ∨ p dvd b* **.**
moreover
{assume *pa*: *p dvd a*
have *pnba*: *p ^ n dvd b*a* **using** *pab* **by** (*simp add*: *mult-commute*)
from *coprime-prime*[*OF ab*, *of p*] *p pa* **have** ¬ *p dvd b* **by** *blast*
with *prime-coprime*[*OF p*, *of b*] *b*
have *cpb*: *coprime b p* **using** *coprime-commute* **by** *blast*
from *coprime-exp*[*OF cpb*] **have** *pnb*: *coprime* (*p ^ n*) *b*
by (*simp add*: *coprime-commute*)
from *coprime-divprod*[*OF pnba pnb*] **have** *?thesis* **by** *blast* **}**
moreover
{assume *pb*: *p dvd b*
have *pnba*: *p ^ n dvd b*a* **using** *pab* **by** (*simp add*: *mult-commute*)
from *coprime-prime*[*OF ab*, *of p*] *p pb* **have** ¬ *p dvd a* **by** *blast*
with *prime-coprime*[*OF p*, *of a*] *a*
have *cpb*: *coprime a p* **using** *coprime-commute* **by** *blast*
from *coprime-exp*[*OF cpb*] **have** *pnb*: *coprime* (*p ^ n*) *a*
by (*simp add*: *coprime-commute*)
from *coprime-divprod*[*OF pab pnb*] **have** *?thesis* **by** *blast* **}**
ultimately have *?thesis* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *nat-mult-eq-one*: (*n*::*nat*) * *m = 1 ⟷ n = 1 ∧ m = 1* (**is** *?lhs ⟷ ?rhs*)
proof
assume *H*: *?lhs*
hence *n dvd 1 m dvd 1* **unfolding** *dvd-def* **by** (*auto simp add*: *mult-commute*)
thus *?rhs* **by** *auto*
next
assume *?rhs* **then show** *?lhs* **by** *auto*
qed

lemma *power-Suc0*[*simp*]: *Suc 0 ^ n = Suc 0*
unfolding *One-nat-def*[*symmetric*] *power-one* **..**
lemma *coprime-pow*: **assumes** *ab*: *coprime a b* **and** *abcn*: *a * b = c ^ n*
shows ∃ *r s*. *a = r ^ n ∧ b = s ^ n*

```
  using ab abcn
proof(induct c arbitrary: a b rule: nat-less-induct)
  fix c a b
  assume H: ∀ m<c. ∀ a b. coprime a b ⟶ a * b = m ^ n ⟶ (∃ r s. a = r ^ n
∧ b = s ^ n) coprime a b a * b = c ^ n
  let ?ths = ∃ r s. a = r^n  ∧ b = s ^n
  {assume n: n = 0
    with H(3) power-one have a*b = 1 by simp
    hence a = 1 ∧ b = 1 by simp
    hence ?ths
      apply −
      apply (rule exI[where x=1])
      apply (rule exI[where x=1])
      using power-one[of  n]
      by simp}
  moreover
  {assume n: n ≠ 0 then obtain m where m: n = Suc m by (cases n, auto)
    {assume c: c = 0
      with H(3) m H(2) have ?ths apply simp
        apply (cases a=0, simp-all)
        apply (rule exI[where x=0], simp)
        apply (rule exI[where x=0], simp)
        done}
    moreover
    {assume c=1 with H(3) power-one have a*b = 1 by simp
        hence a = 1 ∧ b = 1 by simp
        hence ?ths
      apply −
      apply (rule exI[where x=1])
      apply (rule exI[where x=1])
      using power-one[of  n]
      by simp}
  moreover
  {assume c: c≠1 c ≠ 0
    from prime-factor[OF c(1)] obtain p where p: prime p p dvd c by blast
     from prime-divprod-pow[OF p(1) H(2), unfolded H(3), OF divides-exp[OF
p(2), of n]]
    have pnab: p ^ n dvd a ∨ p^n dvd b .
    from p(2) obtain l where l: c = p*l unfolding dvd-def by blast
    have pn0: p^n ≠ 0 using n prime-ge-2 [OF p(1)] by (simp add: neq0-conv)
    {assume pa: p^n dvd a
      then obtain k where k: a = p^n * k unfolding dvd-def by blast
      from l have l dvd c by auto
      with dvd-imp-le[of l c] c have l ≤ c by auto
     moreover {assume l = c with l c  have p = 1 by simp with p have False
by simp}
      ultimately have lc: l < c by arith
      from coprime-lmul2 [OF H(2)[unfolded k coprime-commute[of p^n*k b]]]
      have kb: coprime k b by (simp add: coprime-commute)
```

```
    from H(3) l k pn0 have kbln: k * b = l ^ n
      by (auto simp add: power-mult-distrib)
    from H(1)[rule-format, OF lc kb kbln]
    obtain r s where rs: k = r ^n b = s^n by blast
    from k rs(1) have a = (p*r)^n by (simp add: power-mult-distrib)
    with rs(2) have ?ths by blast }
  moreover
  {assume pb: p^n dvd b
    then obtain k where k: b = p^n * k unfolding dvd-def by blast
    from l have l dvd c by auto
    with dvd-imp-le[of l c] c have l ≤ c by auto
   moreover {assume l = c with l c have p = 1 by simp with p have False
by simp}
    ultimately have lc: l < c by arith
    from coprime-lmul2 [OF H(2)[unfolded k coprime-commute[of p^n*k a]]]
    have kb: coprime k a by (simp add: coprime-commute)
    from H(3) l k pn0 n have kbln: k * a = l ^ n
      by (simp add: power-mult-distrib mult-commute)
    from H(1)[rule-format, OF lc kb kbln]
    obtain r s where rs: k = r ^n a = s^n by blast
    from k rs(1) have b = (p*r)^n by (simp add: power-mult-distrib)
    with rs(2) have ?ths by blast }
  ultimately have ?ths using pnab by blast}
 ultimately have ?ths by blast}
ultimately show ?ths by blast
qed
```

More useful lemmas.

```
lemma prime-product:
  prime (p*q) ⟹ p = 1 ∨ q = 1 unfolding prime-def by auto

lemma prime-exp: prime (p^n) ⟷ prime p ∧ n = 1
proof(induct n)
  case 0 thus ?case by simp
next
  case (Suc n)
  {assume p = 0 hence ?case by simp}
  moreover
  {assume p=1 hence ?case by simp}
  moreover
  {assume p: p ≠ 0 p≠1
    {assume pp: prime (p^Suc n)
      hence p = 1 ∨ p^n = 1 using prime-product[of p p^n] by simp
      with p have n: n = 0
        by (simp only: exp-eq-1 ) simp
      with pp have prime p ∧ Suc n = 1 by simp}
    moreover
    {assume n: prime p ∧ Suc n = 1 hence prime (p^Suc n) by simp}
    ultimately have ?case by blast}
```

```
  ultimately show ?case by blast
qed

lemma prime-power-mult:
  assumes p: prime p and xy: x * y = p ^ k
  shows ∃ i j. x = p ^i ∧ y = p ^ j
  using xy
proof(induct k arbitrary: x y)
  case 0 thus ?case apply simp by (rule exI[where x=0], simp)
next
  case (Suc k x y)
  from Suc.prems have pxy: p dvd x*y by auto
  from prime-divprod[OF p pxy] have pxyc: p dvd x ∨ p dvd y .
  from p have p0: p ≠ 0 by − (rule ccontr, simp)
  {assume px: p dvd x
    then obtain d where d: x = p*d unfolding dvd-def by blast
    from Suc.prems d  have p*d*y = p ^Suc k by simp
    hence th: d*y = p ^k using p0 by simp
    from Suc.hyps[OF th] obtain i j where ij: d = p ^i y = p ^j by blast
    with d have x = p ^Suc i by simp
    with ij(2) have ?case by blast}
  moreover
  {assume px: p dvd y
    then obtain d where d: y = p*d unfolding dvd-def by blast
    from Suc.prems d  have p*d*x = p ^Suc k by (simp add: mult-commute)
    hence th: d*x = p ^k using p0 by simp
    from Suc.hyps[OF th] obtain i j where ij: d = p ^i x = p ^j by blast
    with d have y = p ^Suc i by simp
    with ij(2) have ?case by blast}
  ultimately show ?case  using pxyc by blast
qed

lemma prime-power-exp: assumes p: prime p and n:n ≠ 0
  and xn: x ^n = p ^k shows ∃ i. x = p ^i
  using n xn
proof(induct n arbitrary: k)
  case 0 thus ?case by simp
next
  case (Suc n k) hence th: x*x ^n = p ^k by simp
  {assume n = 0 with prems have ?case apply simp
     by (rule exI[where x=k],simp)}
  moreover
  {assume n: n ≠ 0
    from prime-power-mult[OF p th]
    obtain i j where ij: x = p ^i x ^n = p ^jby blast
    from Suc.hyps[OF n ij(2)] have ?case .}
  ultimately show ?case by blast
qed
```

lemma *divides-primepow*: **assumes** *p*: *prime p*
shows *d dvd p^k* $\longleftrightarrow$ $(\exists\ i.\ i \leq k \wedge d = p\ \hat{}\ i)$
proof
assume *H*: *d dvd p^k* **then obtain** *e* **where** *e*: $d*e = p\hat{}k$
unfolding *dvd-def* **apply** (*auto simp add*: *mult-commute*) **by** *blast*
from *prime-power-mult*[*OF p e*] **obtain** *i j* **where** *ij*: $d = p\hat{}i$ $e=p\hat{}j$ **by** *blast*
from *prime-ge-2*[*OF p*] **have** *p1*: $p > 1$ **by** *arith*
from *e ij* **have** $p\hat{}(i + j) = p\hat{}k$ **by** (*simp add*: *power-add*)
hence $i + j = k$ **using** *power-inject-exp*[*of p i+j k*, *OF p1*] **by** *simp*
hence $i \leq k$ **by** *arith*
with *ij*(*1*) **show** $\exists i{\leq}k.\ d = p\ \hat{}\ i$ **by** *blast*
next
{fix *i* **assume** *H*: $i \leq k$ $d = p\hat{}i$
hence $\exists j.\ k = i + j$ **by** *arith*
then obtain *j* **where** *j*: $k = i + j$ **by** *blast*
hence $p\hat{}k = p\hat{}j*d$ **using** *H*(*2*) **by** (*simp add*: *power-add*)
hence *d dvd p^k* **unfolding** *dvd-def* **by** *auto***}**
thus $\exists i{\leq}k.\ d = p\ \hat{}\ i \Longrightarrow$ *d dvd p* $\hat{}$ *k* **by** *blast*
qed

lemma *coprime-divisors*: *d dvd a* $\Longrightarrow$ *e dvd b* $\Longrightarrow$ *coprime a b* $\Longrightarrow$ *coprime d e*
by (*auto simp add*: *dvd-def coprime*)

declare *power-Suc0*[*simp del*]
declare *even-dvd*[*simp del*]

end

47 Quicksort: Quicksort

theory *Quicksort*
imports *Multiset*
begin

context *linorder*
begin

function *quicksort* :: *'a list* $\Rightarrow$ *'a list* **where**
quicksort [] = [] |
quicksort (*x*#*xs*) = *quicksort*([*y*←*xs*. ~ $x{\leq}y$]) @ [*x*] @ *quicksort*([*y*←*xs*. $x{\leq}y$])
by *pat-completeness auto*

termination
by (*relation measure size*)
(*auto simp*: *length-filter-le*[*THEN order-class.le-less-trans*])

end
context *linorder*

begin

lemma *quicksort-permutes* [*simp*]:
  multiset-of (*quicksort xs*) = *multiset-of xs*
by (*induct xs rule*: *quicksort.induct*) (*auto simp*: *union-ac*)

lemma *set-quicksort* [*simp*]: *set* (*quicksort xs*) = *set xs*
by(*simp add*: *set-count-greater-0*)

lemma *sorted-quicksort*: *sorted*(*quicksort xs*)
apply (*induct xs rule*: *quicksort.induct*)
 apply *simp*
apply (*simp add*:*sorted-Cons sorted-append not-le less-imp-le*)
apply (*metis leD le-cases le-less-trans*)
done

end

end

48 Quotient: Quotient types

theory *Quotient*
imports *ATP-Linkup Hilbert-Choice*
begin

We introduce the notion of quotient types over equivalence relations via type classes.

48.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim$ :: *'a => 'a => bool*.

class *eqv* = *type* +
  fixes *eqv* :: *'a* ⇒ *'a* ⇒ *bool* (**infixl** $\sim$ *50*)

class *equiv* = *eqv* +
  assumes *equiv-refl* [*intro*]: $x \sim x$
  assumes *equiv-trans* [*trans*]: $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$
  assumes *equiv-sym* [*sym*]: $x \sim y \Longrightarrow y \sim x$

lemma *equiv-not-sym* [*sym*]: ¬ ($x \sim y$) ==> ¬ ($y \sim$ (x::*'a*::*equiv*))
proof −
  assume ¬ ($x \sim y$) **then show** ¬ ($y \sim x$)
    by (*rule contrapos-nn*) (*rule equiv-sym*)
qed

lemma *not-equiv-trans1* [*trans*]: ¬ ($x \sim y$) ==> $y \sim z$ ==> ¬ ($x \sim$ (z::*'a*::*equiv*))

proof −
  assume ¬ (*x* ∼ *y*) **and** *y* ∼ *z*
  show ¬ (*x* ∼ *z*)
  proof
    assume *x* ∼ *z*
    also from ‹*y* ∼ *z*› **have** *z* ∼ *y* **..**
    finally have *x* ∼ *y* **.**
    with ‹¬ (*x* ∼ *y*)› **show** *False* **by** *contradiction*
  qed
qed

lemma *not-equiv-trans2* [*trans*]: *x* ∼ *y* ==> ¬ (*y* ∼ *z*) ==> ¬ (*x* ∼ (*z*::'*a*::*equiv*))
proof −
  assume ¬ (*y* ∼ *z*) **then have** ¬ (*z* ∼ *y*) **..**
  also assume *x* ∼ *y* **then have** *y* ∼ *x* **..**
  finally have ¬ (*z* ∼ *x*) **. then show** (¬ *x* ∼ *z*) **..**
qed

The quotient type *'a quot* consists of all *equivalence classes* over elements of the base type *'a*.

typedef *'a quot* = {{*x*. *a* ∼ *x*} | *a*::'*a*::*eqv*. *True*}
  by *blast*

lemma *quotI* [*intro*]: {*x*. *a* ∼ *x*} ∈ *quot*
  unfolding *quot-def* **by** *blast*

lemma *quotE* [*elim*]: *R* ∈ *quot* ==> (!!*a*. *R* = {*x*. *a* ∼ *x*} ==> *C*) ==> *C*
  unfolding *quot-def* **by** *blast*

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

definition
  class :: '*a*::*equiv* => '*a quot* (⌊-⌋) **where**
  ⌊*a*⌋ = *Abs-quot* {*x*. *a* ∼ *x*}

theorem *quot-exhaust*: ∃ *a*. *A* = ⌊*a*⌋
proof (*cases A*)
  fix *R* **assume** *R*: *A* = *Abs-quot R*
  assume *R* ∈ *quot* **then have** ∃ *a*. *R* = {*x*. *a* ∼ *x*} **by** *blast*
  with *R* **have** ∃ *a*. *A* = *Abs-quot* {*x*. *a* ∼ *x*} **by** *blast*
  then show *?thesis* **unfolding** *class-def* **.**
qed

lemma *quot-cases* [*cases type*: *quot*]: (!!*a*. *A* = ⌊*a*⌋ ==> *C*) ==> *C*
  using *quot-exhaust* **by** *blast*

48.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $(\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim b)$
proof
  assume *eq*: $\lfloor a \rfloor = \lfloor b \rfloor$
  show $a \sim b$
  proof −
    from *eq* **have** $\{x.\ a \sim x\} = \{x.\ b \sim x\}$
      by (*simp only*: *class-def Abs-quot-inject quotI*)
    moreover have $a \sim a$ **..**
    ultimately have $a \in \{x.\ b \sim x\}$ **by** *blast*
    then have $b \sim a$ **by** *blast*
    then show *?thesis* **..**
  qed
next
  assume *ab*: $a \sim b$
  show $\lfloor a \rfloor = \lfloor b \rfloor$
  proof −
    have $\{x.\ a \sim x\} = \{x.\ b \sim x\}$
    proof (*rule Collect-cong*)
      fix x **show** $(a \sim x) = (b \sim x)$
      proof
        from *ab* **have** $b \sim a$ **..**
        also assume $a \sim x$
        finally show $b \sim x$ **.**
      next
        note *ab*
        also assume $b \sim x$
        finally show $a \sim x$ **.**
      qed
    qed
    then show *?thesis* **by** (*simp only*: *class-def*)
  qed
qed

48.3 Picking representing elements

definition
  pick :: *'a*::*equiv quot* => *'a* **where**
  pick A = (*SOME a*. *A* = $\lfloor a \rfloor$)

theorem *pick-equiv* [*intro*]: *pick* $\lfloor a \rfloor \sim a$
proof (*unfold pick-def*)
  show (*SOME x*. $\lfloor a \rfloor = \lfloor x \rfloor$) $\sim a$
  proof (*rule someI2*)
    show $\lfloor a \rfloor = \lfloor a \rfloor$ **..**
    fix x **assume** $\lfloor a \rfloor = \lfloor x \rfloor$
    then have $a \sim x$ **.. then show** $x \sim a$ **..**
  qed
qed

theorem *pick-inverse* [*intro*]: $\lfloor pick\ A \rfloor = A$
proof (*cases A*)
  fix *a* **assume** *a*: $A = \lfloor a \rfloor$
  then have $pick\ A \sim a$ **by** (*simp only*: *pick-equiv*)
  then have $\lfloor pick\ A \rfloor = \lfloor a \rfloor$ **..**
  with *a* **show** *?thesis* **by** *simp*
qed

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*: $!!X\ Y.\ P\ X\ Y \Longrightarrow f\ X\ Y == g\ (pick\ X)\ (pick\ Y)$
    and *cong*: $!!x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor$
      $\Longrightarrow P\ \lfloor x \rfloor\ \lfloor y \rfloor \Longrightarrow P\ \lfloor x' \rfloor\ \lfloor y' \rfloor \Longrightarrow g\ x\ y = g\ x'\ y'$
    and *P*: $P\ \lfloor a \rfloor\ \lfloor b \rfloor$
  shows $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
proof −
  from *eq* **and** *P* **have** $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ (pick\ \lfloor a \rfloor)\ (pick\ \lfloor b \rfloor)$ **by** (*simp only*:)
  also have $... = g\ a\ b$
  proof (*rule cong*)
    show $\lfloor pick\ \lfloor a \rfloor \rfloor = \lfloor a \rfloor$ **..**
    moreover
    show $\lfloor pick\ \lfloor b \rfloor \rfloor = \lfloor b \rfloor$ **..**
    moreover
    show $P\ \lfloor a \rfloor\ \lfloor b \rfloor$ **by** (*rule P*)
    ultimately show $P\ \lfloor pick\ \lfloor a \rfloor \rfloor\ \lfloor pick\ \lfloor b \rfloor \rfloor$ **by** (*simp only*:)
  qed
  finally show *?thesis* **.**
qed

theorem *quot-function*:
  assumes $!!X\ Y.\ f\ X\ Y == g\ (pick\ X)\ (pick\ Y)$
    and $!!x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor \Longrightarrow g\ x\ y = g\ x'\ y'$
  shows $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
  using *assms* **and** *TrueI*
  by (*rule quot-cond-function*)

theorem *quot-function'*:
  $(!!X\ Y.\ f\ X\ Y == g\ (pick\ X)\ (pick\ Y)) \Longrightarrow$
    $(!!x\ x'\ y\ y'.\ x \sim x' \Longrightarrow y \sim y' \Longrightarrow g\ x\ y = g\ x'\ y') \Longrightarrow$
    $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
  by (*rule quot-function*) (*simp-all only*: *quot-equality*)

end

49 Ramsey: Ramsey's Theorem

theory *Ramsey*
imports *ATP-Linkup Infinite-Set*
begin

49.1 Preliminaries

49.1.1 "Axiom" of Dependent Choice

consts *choice* :: $('a \Rightarrow bool) \Rightarrow ('a * 'a)\ set \Rightarrow nat \Rightarrow 'a$
— An integer-indexed chain of choices
primrec
choice-0: $choice\ P\ r\ 0 = (SOME\ x.\ P\ x)$

choice-Suc: $choice\ P\ r\ (Suc\ n) = (SOME\ y.\ P\ y\ \&\ (choice\ P\ r\ n,\ y) \in r)$

lemma *choice-n*:
assumes *P0*: *P x0*
and *Pstep*: $!!x.\ P\ x \Longrightarrow \exists y.\ P\ y\ \&\ (x,y) \in r$
shows *P* (*choice P r n*)
proof (*induct n*)
case *0* **show** *?case* **by** (*force intro*: *someI P0*)
next
case *Suc* **thus** *?case* **by** (*auto intro*: *someI2-ex* [*OF Pstep*])
qed

lemma *dependent-choice*:
assumes *trans*: *trans r*
and *P0*: *P x0*
and *Pstep*: $!!x.\ P\ x \Longrightarrow \exists y.\ P\ y\ \&\ (x,y) \in r$
obtains *f* :: $nat \Rightarrow 'a$ **where**
$!!n.\ P\ (f\ n)$ **and** $!!n\ m.\ n < m \Longrightarrow (f\ n, f\ m) \in r$
proof
fix *n*
show *P* (*choice P r n*) **by** (*blast intro*: *choice-n* [*OF P0 Pstep*])
next
have *PSuc*: $\forall n.\ (choice\ P\ r\ n,\ choice\ P\ r\ (Suc\ n)) \in r$
using *Pstep* [*OF choice-n* [*OF P0 Pstep*]]
by (*auto intro*: *someI2-ex*)
fix *n m* :: *nat*
assume *less*: $n < m$
show $(choice\ P\ r\ n,\ choice\ P\ r\ m) \in r$ **using** *PSuc*
by (*auto intro*: *less-Suc-induct* [*OF less*] *transD* [*OF trans*])
qed

49.1.2 Partitions of a Set

definition

part :: *nat* => *nat* => *'a set* => (*'a set* => *nat*) => *bool*
— the function f partitions the r-subsets of the typically infinite set Y into s distinct categories.
where
part r s Y f = (∀ *X*. *X* ⊆ *Y* & *finite X* & *card X* = *r* −−> *f X* < *s*)

For induction, we decrease the value of r in partitions.

lemma *part-Suc-imp-part*:
[| *infinite Y*; *part* (*Suc r*) *s Y f*; *y* ∈ *Y* |]
==> *part r s* (*Y* − {*y*}) (%*u*. *f* (*insert y u*))
apply(*simp add*: *part-def*, *clarify*)
apply(*drule-tac x*=*insert y X* **in** *spec*)
apply(*force*)
done

lemma *part-subset*: *part r s YY f* ==> *Y* ⊆ *YY* ==> *part r s Y f*
unfolding *part-def* **by** *blast*

49.2 Ramsey's Theorem: Infinitary Version

lemma *Ramsey-induction*:
fixes *s* **and** *r*::*nat*
shows
!!(*YY*::*'a set*) (*f*::*'a set* => *nat*).
[|*infinite YY*; *part r s YY f*|]
==> ∃ *Y' t'*. *Y'* ⊆ *YY* & *infinite Y'* & *t'* < *s* &
(∀ *X*. *X* ⊆ *Y'* & *finite X* & *card X* = *r* −−> *f X* = *t'*)
proof (*induct r*)
case *0*
thus *?case* **by** (*auto simp add*: *part-def card-eq-0-iff cong*: *conj-cong*)
next
case (*Suc r*)
show *?case*
proof −
from *Suc.prems infinite-imp-nonempty* **obtain** *yy* **where** *yy*: *yy* ∈ *YY* **by** *blast*
let *?ramr* = {((*y*,*Y*,*t*),(*y'*,*Y'*,*t'*)). *y'* ∈ *Y* & *Y'* ⊆ *Y*}
let *?propr* = %(*y*,*Y*,*t*).
y ∈ *YY* & *y* ∉ *Y* & *Y* ⊆ *YY* & *infinite Y* & *t* < *s*
& (∀ *X*. *X*⊆*Y* & *finite X* & *card X* = *r* −−> (*f o insert y*) *X* = *t*)
have *infYY'*: *infinite* (*YY*−{*yy*}) **using** *Suc.prems* **by** *auto*
have *partf'*: *part r s* (*YY* − {*yy*}) (*f* ∘ *insert yy*)
by (*simp add*: *o-def part-Suc-imp-part yy Suc.prems*)
have *transr*: *trans ?ramr* **by** (*force simp add*: *trans-def*)
from *Suc.hyps* [*OF infYY' partf'*]
obtain *Y0* **and** *t0*
where *Y0* ⊆ *YY* − {*yy*} *infinite Y0* *t0* < *s*
∀ *X*. *X*⊆*Y0* ∧ *finite X* ∧ *card X* = *r* ⟶ (*f* ∘ *insert yy*) *X* = *t0*
by *blast*
with *yy* **have** *propr0*: *?propr*(*yy*,*Y0*,*t0*) **by** *blast*

have *proprstep*: $\bigwedge x.$ *?propr* $x \Longrightarrow \exists\, y.$ *?propr* $y \wedge (x,\, y) \in$ *?ramr*
proof −
fix x
assume *px*: *?propr* x **thus** *?thesis* x
proof (*cases* x)
case (*fields* yx Yx tx)
then obtain yx' **where** yx': $yx' \in Yx$ **using** *px*
by (*blast dest*: *infinite-imp-nonempty*)
have *infYx'*: *infinite* $(Yx - \{yx'\})$ **using** *fields px* **by** *auto*
with *fields px* yx' *Suc.prems*
have *partfx'*: *part* $r\ s$ $(Yx - \{yx'\})$ $(f \circ$ *insert* $yx')$
by (*simp add*: *o-def part-Suc-imp-part part-subset* [**where** *?YY*=*YY*])
from *Suc.hyps* [*OF infYx' partfx'*]
obtain Y' **and** t'
where Y': $Y' \subseteq Yx - \{yx'\}$ *infinite* Y' $t' < s$
$\forall X.\ X \subseteq Y' \wedge$ *finite* $X \wedge$ *card* $X = r \longrightarrow (f \circ$ *insert* $yx')\ X = t'$
by *blast*
show *?thesis*
proof
show *?propr* (yx', Y', t') & $(x,\, (yx', Y', t')) \in$ *?ramr*
using *fields* Y' yx' *px* **by** *blast*
qed
qed
qed
from *dependent-choice* [*OF transr propr0 proprstep*]
obtain g **where** *pg*: $!!n{::}nat.$ *?propr* $(g\ n)$
and *rg*: $!!n\ m.\ n<m \Longrightarrow (g\ n,\ g\ m) \in$ *?ramr* **by** *blast*
let *?gy* $= (\lambda n.$ *let* $(y,Y,t) = g\ n$ *in* $y)$
let *?gt* $= (\lambda n.$ *let* $(y,Y,t) = g\ n$ *in* $t)$
have *rangeg*: $\exists\, k.$ *range ?gt* $\subseteq \{..<k\}$
proof (*intro exI subsetI*)
fix x
assume $x \in$ *range ?gt*
then obtain n **where** $x =$ *?gt* n **..**
with *pg* [*of n*] **show** $x \in \{..<s\}$ **by** (*cases g n*) *auto*
qed
have *finite* (*range ?gt*)
by (*simp add*: *finite-nat-iff-bounded rangeg*)
then obtain s' **and** n'
where s': $s' =$ *?gt* n'
and *infeqs'*: *infinite* $\{n.$ *?gt* $n = s'\}$
by (*rule inf-img-fin-domE*) (*auto simp add*: *vimage-def intro*: *nat-infinite*)
with *pg* [*of* n'] **have** *less'*: $s'<s$ **by** (*cases g* n') *auto*
have *inj-gy*: *inj ?gy*
proof (*rule linorder-injI*)
fix $m\ m'$ :: *nat* **assume** *less*: $m < m'$ **show** *?gy* $m \neq$ *?gy* m'
using *rg* [*OF less*] *pg* [*of m*] **by** (*cases g m*, *cases g* m') *auto*
qed
show *?thesis*

```
  proof (intro exI conjI)
    show ?gy ` {n. ?gt n = s'} ⊆ YY using pg
      by (auto simp add: Let-def split-beta)
    show infinite (?gy ` {n. ?gt n = s'}) using infeqs'
      by (blast intro: inj-gy [THEN subset-inj-on] dest: finite-imageD)
    show s' < s by (rule less')
    show ∀X. X ⊆ ?gy ` {n. ?gt n = s'} & finite X & card X = Suc r
       --> f X = s'
    proof -
      {fix X
       assume X ⊆ ?gy ` {n. ?gt n = s'}
          and cardX: finite X card X = Suc r
      then obtain AA where AA: AA ⊆ {n. ?gt n = s'} and Xeq: X = ?gy`AA

          by (auto simp add: subset-image-iff)
       with cardX have AA≠{} by auto
       hence AAleast: (LEAST x. x ∈ AA) ∈ AA by (auto intro: LeastI-ex)
       have f X = s'
       proof (cases g (LEAST x. x ∈ AA))
         case (fields ya Ya ta)
         with AAleast Xeq
         have ya: ya ∈ X by (force intro!: rev-image-eqI)
         hence f X = f (insert ya (X - {ya})) by (simp add: insert-absorb)
         also have ... = ta
         proof -
           have X - {ya} ⊆ Ya
           proof
             fix x assume x: x ∈ X - {ya}
             then obtain a' where xeq: x = ?gy a' and a': a' ∈ AA
               by (auto simp add: Xeq)
             hence a' ≠ (LEAST x. x ∈ AA) using x fields by auto
             hence lessa': (LEAST x. x ∈ AA) < a'
               using Least-le [of %x. x ∈ AA, OF a'] by arith
             show x ∈ Ya using xeq fields rg [OF lessa'] by auto
           qed
           moreover
           have card (X - {ya}) = r
             by (simp add: cardX ya)
           ultimately show ?thesis
             using pg [of LEAST x. x ∈ AA] fields cardX
             by (clarsimp simp del:insert-Diff-single)
         qed
         also have ... = s' using AA AAleast fields by auto
         finally show ?thesis .
       qed}
      thus ?thesis by blast
    qed
  qed
qed
```

qed

theorem *Ramsey*:
fixes *s r* :: *nat* **and** *Z*::'*a set* **and** *f*::'*a set* => *nat*
shows
[|*infinite Z*;
∀ *X*. *X* ⊆ *Z* & *finite X* & *card X* = *r* --> *f X* < *s*|]
==> ∃ *Y t*. *Y* ⊆ *Z* & *infinite Y* & *t* < *s*
& (∀ *X*. *X* ⊆ *Y* & *finite X* & *card X* = *r* --> *f X* = *t*)
by (*blast intro*: *Ramsey-induction* [*unfolded part-def*])

corollary *Ramsey2*:
fixes *s*::*nat* **and** *Z*::'*a set* **and** *f*::'*a set* => *nat*
assumes *infZ*: *infinite Z*
and *part*: ∀ *x*∈*Z*. ∀ *y*∈*Z*. *x*≠*y* --> *f*{*x*,*y*} < *s*
shows
∃ *Y t*. *Y* ⊆ *Z* & *infinite Y* & *t* < *s* & (∀ *x*∈*Y*. ∀ *y*∈*Y*. *x*≠*y* --> *f*{*x*,*y*} = *t*)
proof −
have *part2*: ∀ *X*. *X* ⊆ *Z* & *finite X* & *card X* = *2* --> *f X* < *s*
using *part* **by** (*fastsimp simp add*: *nat-number card-Suc-eq*)
obtain *Y t*
where *Y* ⊆ *Z infinite Y t* < *s*
(∀ *X*. *X* ⊆ *Y* & *finite X* & *card X* = *2* --> *f X* = *t*)
by (*insert Ramsey* [*OF infZ part2*]) *auto*
moreover from *this* **have** ∀ *x*∈*Y*. ∀ *y*∈*Y*. *x* ≠ *y* ⟶ *f* {*x*, *y*} = *t* **by** *auto*
ultimately show *?thesis* **by** *iprover*
qed

49.3 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [5].

definition
disj-wf :: ('*a* * '*a*)*set* => *bool*
where
disj-wf r = (∃ *T*. ∃ *n*::*nat*. (∀ *i*<*n*. *wf*(*T i*)) & *r* = (⋃ *i*<*n*. *T i*))

definition
transition-idx :: [*nat* => '*a*, *nat* => ('*a*∗'*a*)*set*, *nat set*] => *nat*
where
transition-idx s T A =
(*LEAST k*. ∃ *i j*. *A* = {*i*,*j*} & *i*<*j* & (*s j*, *s i*) ∈ *T k*)

lemma *transition-idx-less*:
[|*i*<*j*; (*s j*, *s i*) ∈ *T k*; *k*<*n*|] ==> *transition-idx s T* {*i*,*j*} < *n*
apply (*subgoal-tac transition-idx s T* {*i*, *j*} ≤ *k*, *simp*)
apply (*simp add*: *transition-idx-def*, *blast intro*: *Least-le*)

done

lemma *transition-idx-in*:
  $[|i<j;\ (s\ j,\ s\ i) \in T\ k|] ==> (s\ j,\ s\ i) \in T\ (transition\text{-}idx\ s\ T\ \{i,j\})$
apply (*simp add*: *transition-idx-def doubleton-eq-iff conj-disj-distribR*
    cong: *conj-cong*)
apply (*erule LeastI*)
done

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

lemma *disj-wf*:
  $disj\text{-}wf(r) = (\exists\, T.\ \exists\, n{::}nat.\ (\forall\, i{<}n.\ wf(T\ i))\ \&\ r \subseteq (\bigcup i{<}n.\ T\ i))$
apply (*auto simp add*: *disj-wf-def*)
apply (*rule-tac x*=%*i*. *T i Int r* **in** *exI*)
apply (*rule-tac x*=*n* **in** *exI*)
apply (*force simp add*: *wf-Int1*)
done

theorem *trans-disj-wf-implies-wf*:
 assumes *transr*: *trans r*
   and *dwf*: *disj-wf*(*r*)
 shows *wf r*
proof (*simp only*: *wf-iff-no-infinite-down-chain*, *rule notI*)
 assume $\exists\, s.\ \forall\, i.\ (s\ (Suc\ i),\ s\ i) \in r$
 then obtain *s* **where** *sSuc*: $\forall\, i.\ (s\ (Suc\ i),\ s\ i) \in r$ **..**
 have *s*: $!!i\ j.\ i < j ==> (s\ j,\ s\ i) \in r$
 proof −
  fix *i* **and** *j*::*nat*
  assume *less*: *i*<*j*
  thus $(s\ j,\ s\ i) \in r$
  proof (*rule less-Suc-induct*)
   show $\bigwedge i.\ (s\ (Suc\ i),\ s\ i) \in r$ **by** (*simp add*: *sSuc*)
   show $\bigwedge i\ j\ k.\ [\![(s\ j,\ s\ i) \in r;\ (s\ k,\ s\ j) \in r]\!] \Longrightarrow (s\ k,\ s\ i) \in r$
    using *transr* **by** (*unfold trans-def*, *blast*)
  qed
 qed
 from *dwf*
 obtain *T* **and** *n*::*nat* **where** *wfT*: $\forall\, k{<}n.\ wf(T\ k)$ **and** *r*: $r = (\bigcup k{<}n.\ T\ k)$
  by (*auto simp add*: *disj-wf-def*)
 have *s-in-T*: $\bigwedge i\ j.\ i{<}j ==> \exists\, k.\ (s\ j,\ s\ i) \in T\ k\ \&\ k{<}n$
 proof −
  fix *i* **and** *j*::*nat*
  assume *less*: *i*<*j*
  hence $(s\ j,\ s\ i) \in r$ **by** (*rule s* [*of i j*])
  thus $\exists\, k.\ (s\ j,\ s\ i) \in T\ k\ \&\ k{<}n$ **by** (*auto simp add*: *r*)
 qed
 have *trless*: $!!i\ j.\ i{\neq}j ==> transition\text{-}idx\ s\ T\ \{i,j\} < n$
  apply (*auto simp add*: *linorder-neq-iff*)

```
    apply (blast dest: s-in-T transition-idx-less)
    apply (subst insert-commute)
    apply (blast dest: s-in-T transition-idx-less)
    done
  have
  ∃K k. K ⊆ UNIV & infinite K & k < n &
        (∀i∈K. ∀j∈K. i≠j −−> transition-idx s T {i,j} = k)
    by (rule Ramsey2) (auto intro: trless nat-infinite)
  then obtain K and k
    where infK: infinite K and less: k < n and
        allk: ∀i∈K. ∀j∈K. i≠j −−> transition-idx s T {i,j} = k
    by auto
  have ∀m. (s (enumerate K (Suc m)), s(enumerate K m)) ∈ T k
  proof
    fix m::nat
    let ?j = enumerate K (Suc m)
    let ?i = enumerate K m
    have jK: ?j ∈ K by (simp add: enumerate-in-set infK)
    have iK: ?i ∈ K by (simp add: enumerate-in-set infK)
    have ij: ?i < ?j by (simp add: enumerate-step infK)
    have ijk: transition-idx s T {?i,?j} = k using iK jK ij
      by (simp add: allk)
    obtain k' where (s ?j, s ?i) ∈ T k' k'<n
      using s-in-T [OF ij] by blast
    thus (s ?j, s ?i) ∈ T k
      by (simp add: ijk [symmetric] transition-idx-in ij)
  qed
  hence ~ wf(T k) by (force simp add: wf-iff-no-infinite-down-chain)
  thus False using wfT less by blast
qed

end
```

50 RBT: Red-Black Trees

This theory defines purely functional red-black trees which can be used as an efficient representation of finite maps.

50.1 Data type and invariant

The type ('k, 'v) *rbt* denotes red-black trees with keys of type 'k and values of type '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 *isrbt t*. This theory provides lemmas to prove that the invariant is satisfied throughout the computation.

The interpretation function *RBT.map-of* returns the partial map represented by a red-black tree:

$RBT.map\text{-}of :: ('a, 'b)\ rbt \Rightarrow 'a \rightharpoonup 'b$

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 $O(\log n)$.

50.2 Operations

Currently, the following operations are supported:

$Empty :: ('a, 'b)\ rbt$

Returns the empty tree. $O(1)$

$insrt :: 'a \Rightarrow 'b \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

Updates the map at a given position. $O(\log n)$

$RBT.delete :: 'a \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

Deletes a map entry at a given position. $O(\log n)$

$union :: ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

Forms the union of two trees, preferring entries from the first one.

$RBT.map :: ('a \Rightarrow 'b) \Rightarrow ('c, 'a)\ rbt \Rightarrow ('c, 'b)\ rbt$

Maps a function over the values of a map. $O(n)$

50.3 Invariant preservation

$isrbt\ Empty$	(*Empty-isrbt*)
$isrbt\ ?t \Longrightarrow isrbt\ (insrt\ ?k\ ?v\ ?t)$	(*insrt-isrbt*)
$isrbt\ ?t \Longrightarrow isrbt\ (RBT.delete\ ?k\ ?t)$	(*delete-isrbt*)
$isrbt\ ?lt \Longrightarrow isrbt\ (union\ ?lt\ ?rt)$	(*union-isrbt*)
$isrbt\ (RBT.map\ ?f\ ?t) = isrbt\ ?t$	(*map-isrbt*)

50.4 Map Semantics

map-of-Empty

$RBT.map\text{-}of\ Empty = empty$

map-of-insert

$isrbt\ ?t \Longrightarrow RBT.map\text{-}of\ (insrt\ ?k\ ?v\ ?t) = RBT.map\text{-}of\ ?t(?k \mapsto ?v)$

map-of-delete

$isrbt\ ?t \Longrightarrow RBT.map\text{-}of\ (RBT.delete\ ?k\ ?t) = RBT.map\text{-}of\ ?t\restriction_{(-\ \{?k\})}$

map-of-union

$[\![isrbt\ ?s;\ st\ ?t]\!]$
$\Longrightarrow RBT.map\text{-}of\ (union\ ?s\ ?t) = RBT.map\text{-}of\ ?s\ ++\ RBT.map\text{-}of\ ?t$

map-of-map

$RBT.map\text{-}of\ (RBT.map\ ?f\ ?t) = option\text{-}map\ ?f \circ RBT.map\text{-}of\ ?t$

end

51 State-Monad: Combinators syntax for generic, open state monads (single threaded monads)

theory *State-Monad*
imports *List*
begin

51.1 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-threaded).

To enter from the Haskell world, http://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.

51.2 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 σ for types representing states and α, β, γ, ... 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 σ are used in a single-threaded way: after application of a transformation on a value of type σ, the former value should not be used again. To achieve this, we use a set of monad combinators:

```
notation fcomp (infixl >> 60)
notation (xsymbols) fcomp (infixl ≫ 60)
notation scomp (infixl >>= 60)
notation (xsymbols) scomp (infixl ≫= 60)

abbreviation (input)
  return ≡ Pair

definition
  run :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b where
  run f = f

print-ast-translation ⟪
  [(@{const-syntax run}, fn (f::ts) => Syntax.mk-appl f ts)]
⟫
```

Given two transformations f and g, they may be directly composed using the *op* $\gg$ combinator, forming a forward composition: $(f \gg g)\ s = f\ (g\ s)$.

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the *op* $\gg=$ combinator: $(f \gg= (\lambda x.\ g))\ s = (let\ (x,\ s') = f\ s\ in\ g\ s')$.

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 *Pair* for this purpose.

The *run* ist just a marker.

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.

51.3 Obsolete runs

run is just a doodle and should not occur nested:

lemma *run-simp* [*simp*]:
$\bigwedge f.\ run\ (run\ f) = run\ f$
$\bigwedge f\ g.\ run\ f \gg\!= g = f \gg\!= g$
$\bigwedge f\ g.\ run\ f \gg g = f \gg g$
$\bigwedge f\ g.\ f \gg\!= (\lambda x.\ run\ g) = f \gg\!= (\lambda x.\ g)$
$\bigwedge f\ g.\ f \gg run\ g = f \gg g$
$\bigwedge f.\ f = run\ f \longleftrightarrow True$
$\bigwedge f.\ run\ f = f \longleftrightarrow True$
unfolding *run-def* **by** *rule+*

51.4 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

lemma
return-scomp [*simp*]: $return\ x \gg\!= f = f\ x$
unfolding *scomp-def* **by** (*simp add*: *expand-fun-eq*)

lemma
scomp-return [*simp*]: $x \gg\!= return = x$
unfolding *scomp-def* **by** (*simp add*: *expand-fun-eq split-Pair*)

lemma
id-fcomp [*simp*]: $id \gg f = f$
unfolding *fcomp-def* **by** *simp*

lemma
fcomp-id [*simp*]: $f \gg id = f$
unfolding *fcomp-def* **by** *simp*

lemma
scomp-scomp [*simp*]: $(f \gg\!= g) \gg\!= h = f \gg\!= (\lambda x.\ g\ x \gg\!= h)$
unfolding *scomp-def* **by** (*simp add*: *split-def expand-fun-eq*)

lemma
scomp-fcomp [*simp*]: $(f \gg\!= g) \gg h = f \gg\!= (\lambda x.\ g\ x \gg h)$
unfolding *scomp-def fcomp-def* **by** (*simp add*: *split-def expand-fun-eq*)

lemma
fcomp-scomp [*simp*]: $(f \gg g) \gg\!= h = f \gg (g \gg\!= h)$
unfolding *scomp-def fcomp-def* **by** (*simp add*: *split-def expand-fun-eq*)

lemma
fcomp-fcomp [*simp*]: $(f \gg g) \gg h = f \gg (g \gg h)$
unfolding *fcomp-def o-assoc* **..**

lemmas *monad-simp* = *run-simp return-scomp scomp-return id-fcomp fcomp-id*
scomp-scomp scomp-fcomp fcomp-scomp fcomp-fcomp

Evaluation of monadic expressions by force:

lemmas *monad-collapse* = *monad-simp o-apply o-assoc split-Pair split-comp*
scomp-def fcomp-def run-def

51.5 ML abstract operations

ML ⟪
structure StateMonad =
struct

fun liftT T sT = *sT* −−> *HOLogic.mk-prodT* (*T*, *sT*);
fun liftT′ sT = *sT* −−> *sT*;

fun return T sT x = *Const* (@{*const-name return*}, *T* −−> *liftT T sT*) $ *x*;

fun scomp T1 T2 sT f g = *Const* (@{*const-name scomp*},
liftT T1 sT −−> (*T1* −−> *liftT T2 sT*) −−> *liftT T2 sT*) $ *f* $ *g*;

fun run T sT f = *Const* (@{*const-name run*}, *liftT′* (*liftT T sT*)) $ *f*;

end;
⟫

51.6 Syntax

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

nonterminals *do-expr*

syntax
-*do* :: *do-expr* ⇒ *'a*
(*do* - *done* [*12*] *12*)
-*scomp* :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(- <− -;// - [*1000*, *13*, *12*] *12*)

```
  -fcomp :: 'a ⇒ do-expr ⇒ do-expr
    (-;// - [13, 12] 12)
  -let :: pttrn ⇒ 'a ⇒ do-expr ⇒ do-expr
    (let - = -;// - [1000, 13, 12] 12)
  -nil :: 'a ⇒ do-expr
    (- [12] 12)

syntax (xsymbols)
  -scomp :: pttrn ⇒ 'a ⇒ do-expr ⇒ do-expr
    (- ← -;// - [1000, 13, 12] 12)

translations
  -do f => CONST run f
  -scomp x f g => f ≫= (λx. g)
  -fcomp f g => f ≫ g
  -let x t f => CONST Let t (λx. f)
  -nil f => f

print-translation ⟪⟪
let
  fun dest-abs-eta (Abs (abs as (-, ty, -))) =
        let
          val (v, t) = Syntax.variant-abs abs;
        in ((v, ty), t) end
    | dest-abs-eta t =
        let
          val (v, t) = Syntax.variant-abs (, dummyT, t $ Bound 0);
        in ((v, dummyT), t) end
  fun unfold-monad (Const (@{const-syntax scomp}, -) $ f $ g) =
        let
          val ((v, ty), g') = dest-abs-eta g;
        in Const (-scomp, dummyT) $ Free (v, ty) $ f $ unfold-monad g' end
    | unfold-monad (Const (@{const-syntax fcomp}, -) $ f $ g) =
        Const (-fcomp, dummyT) $ f $ unfold-monad g
    | unfold-monad (Const (@{const-syntax Let}, -) $ f $ g) =
        let
          val ((v, ty), g') = dest-abs-eta g;
        in Const (-let, dummyT) $ Free (v, ty) $ f $ unfold-monad g' end
    | unfold-monad (Const (@{const-syntax Pair}, -) $ f) =
        Const (return, dummyT) $ f
    | unfold-monad f = f;
  fun tr' (f::ts) =
    list-comb (Const (-do, dummyT) $ unfold-monad f, ts)
in [(@{const-syntax run}, tr')] end;
⟫⟫
```

51.7 Combinators

definition

lift :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c \Rightarrow 'b \times 'c$ **where**
lift f x = *return* (*f x*)

primrec
list :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'b$ **where**
list f [] = *id*
| *list f* (*x*#*xs*) = (*do f x*; *list f xs done*)

primrec
list-yield :: $('a \Rightarrow 'b \Rightarrow 'c \times 'b) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'c\ list \times 'b$ **where**
list-yield f [] = *return* []
| *list-yield f* (*x*#*xs*) = (*do y* ← *f x*; *ys* ← *list-yield f xs*; *return* (*y*#*ys*) *done*)

definition
collapse :: $('a \Rightarrow ('a \Rightarrow 'b \times 'a) \times 'a) \Rightarrow 'a \Rightarrow 'b \times 'a$ **where**
collapse f = (*do g* ← *f*; *g done*)

combinator properties

lemma *list-append* [*simp*]:
list f (*xs* @ *ys*) = *list f xs* ≫ *list f ys*
by (*induct xs*) (*simp-all del*: *id-apply*)

lemma *list-cong* [*fundef-cong*, *recdef-cong*]:
⟦ $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x$; *xs* = *ys* ⟧
⟹ *list f xs* = *list g ys*
proof (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons* **have** $\bigwedge y.\ y \in set\ (x\ \#\ xs) \Longrightarrow f\ y = g\ y$ **by** *auto*
then have $\bigwedge y.\ y \in set\ xs \Longrightarrow f\ y = g\ y$ **by** *auto*
with *Cons* **have** *list f xs* = *list g xs* **by** *auto*
with *Cons* **have** *list f* (*x* # *xs*) = *list g* (*x* # *xs*) **by** *auto*
with *Cons* **show** *list f* (*x* # *xs*) = *list g ys* **by** *auto*
qed

lemma *list-yield-cong* [*fundef-cong*, *recdef-cong*]:
⟦ $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x$; *xs* = *ys* ⟧
⟹ *list-yield f xs* = *list-yield g ys*
proof (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons* **have** $\bigwedge y.\ y \in set\ (x\ \#\ xs) \Longrightarrow f\ y = g\ y$ **by** *auto*
then have $\bigwedge y.\ y \in set\ xs \Longrightarrow f\ y = g\ y$ **by** *auto*
with *Cons* **have** *list-yield f xs* = *list-yield g xs* **by** *auto*
with *Cons* **have** *list-yield f* (*x* # *xs*) = *list-yield g* (*x* # *xs*) **by** *auto*
with *Cons* **show** *list-yield f* (*x* # *xs*) = *list-yield g ys* **by** *auto*
qed

For an example, see HOL/ex/Random.thy.

end

52 Univ-Poly: Univariate Polynomials

theory *Univ-Poly*
imports *Main*
begin

Application of polynomial as a function.

primrec (**in** *semiring-0*) *poly* :: *'a list => 'a => 'a* **where**
poly-Nil: *poly [] x = 0*
| *poly-Cons*: *poly (h#t) x = h + x * poly t x*

52.1 Arithmetic Operations on Polynomials

addition

primrec (**in** *semiring-0*) *padd* :: *'a list ⇒ 'a list ⇒ 'a list* (**infixl** *+++ 65*)
where
padd-Nil: *[] +++ l2 = l2*
| *padd-Cons*: *(h#t) +++ l2 = (if l2 = [] then h#t*
else (h + hd l2)#(t +++ tl l2))

Multiplication by a constant

primrec (**in** *semiring-0*) *cmult* :: *'a ⇒ 'a list ⇒ 'a list* (**infixl** *%** *70*) **where**
cmult-Nil: *c %* [] = []*
| *cmult-Cons*: *c %* (h#t) = (c * h)#(c %* t)*

Multiplication by a polynomial

primrec (**in** *semiring-0*) *pmult* :: *'a list ⇒ 'a list ⇒ 'a list* (**infixl** *** *70*)
where
pmult-Nil: *[] *** l2 = []*
| *pmult-Cons*: *(h#t) *** l2 = (if t = [] then h %* l2*
else (h % l2) +++ ((0) # (t *** l2)))*

Repeated multiplication by a polynomial

primrec (**in** *semiring-0*) *mulexp* :: *nat ⇒ 'a list ⇒ 'a list ⇒ 'a list* **where**
mulexp-zero: *mulexp 0 p q = q*
| *mulexp-Suc*: *mulexp (Suc n) p q = p *** mulexp n p q*

Exponential

primrec (**in** *semiring-1*) *pexp* :: *'a list ⇒ nat ⇒ 'a list* (**infixl** *%^ *80*) **where**
pexp-0: *p %^ 0 = [1]*
| *pexp-Suc*: *p %^ (Suc n) = p *** (p %^ n)*

Quotient related value of dividing a polynomial by x + a

primrec (**in** *field*) *pquot* :: *'a list ⇒ 'a ⇒ 'a list* **where**

pquot-Nil: *pquot* [] *a*= []
| *pquot-Cons*: *pquot* (*h*#*t*) *a* = (*if t* = [] *then* [*h*]
else (*inverse*(*a*) ∗ (*h* − *hd*(*pquot t a*)))#(*pquot t a*))

normalization of polynomials (remove extra 0 coeff)

primrec (**in** *semiring-0*) *pnormalize* :: *'a list* ⇒ *'a list* **where**
pnormalize-Nil: *pnormalize* [] = []
| *pnormalize-Cons*: *pnormalize* (*h*#*p*) = (*if* ((*pnormalize p*) = [])
then (*if* (*h* = *0*) *then* [] *else* [*h*])
else (*h*#(*pnormalize p*)))

definition (**in** *semiring-0*) *pnormal p* = ((*pnormalize p* = *p*) ∧ *p* ≠ [])
definition (**in** *semiring-0*) *nonconstant p* = (*pnormal p* ∧ (∀ *x*. *p* ≠ [*x*]))

Other definitions

definition (**in** *ring-1*)
poly-minus :: *'a list* => *'a list* (−− - [*80*] *80*) **where**
−− *p* = (− *1*) %∗ *p*

definition (**in** *semiring-0*)
divides :: *'a list* ⇒ *'a list* ⇒ *bool* (**infixl** *divides 70*) **where**
p1 divides p2 = (∃ *q*. *poly p2* = *poly*(*p1* ∗∗∗ *q*))

— order of a polynomial
definition (**in** *ring-1*) *order* :: *'a* => *'a list* => *nat* **where**
order a p = (*SOME n*. ([−*a*, *1*] %^ *n*) *divides p* &
~ (([−*a*, *1*] %^ (*Suc n*)) *divides p*))

— degree of a polynomial
definition (**in** *semiring-0*) *degree* :: *'a list* => *nat* **where**
degree p = *length* (*pnormalize p*) − *1*

— squarefree polynomials — NB with respect to real roots only.
definition (**in** *ring-1*)
rsquarefree :: *'a list* => *bool* **where**
rsquarefree p = (*poly p* ≠ *poly* [] &
(∀ *a*. (*order a p* = *0*) | (*order a p* = *1*)))

context *semiring-0*
begin

lemma *padd-Nil2*[*simp*]: *p* +++ [] = *p*
by (*induct p*) *auto*

lemma *padd-Cons-Cons*: (*h1* # *p1*) +++ (*h2* # *p2*) = (*h1* + *h2*) # (*p1* +++ *p2*)
by *auto*

lemma *pminus-Nil*[*simp*]: −− [] = []

by (*simp add*: *poly-minus-def*)

lemma *pmult-singleton*: [*h1*] *** *p1* = *h1* %* *p1* **by** *simp*
end

lemma (**in** *semiring-1*) *poly-ident-mult*[*simp*]: *1* %* *t* = *t* **by** (*induct t*, *auto*)

lemma (**in** *semiring-0*) *poly-simple-add-Cons*[*simp*]: [*a*] +++ ((*0*)#*t*) = (*a*#*t*)
by *simp*

Handy general properties

lemma (**in** *comm-semiring-0*) *padd-commut*: *b* +++ *a* = *a* +++ *b*
proof(*induct b arbitrary*: *a*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons b bs a*) **thus** *?case* **by** (*cases a*, *simp-all add*: *add-commute*)
qed

lemma (**in** *comm-semiring-0*) *padd-assoc*: ∀ *b c*. (*a* +++ *b*) +++ *c* = *a* +++ (*b* +++ *c*)
apply (*induct a arbitrary*: *b c*)
apply (*simp*, *clarify*)
apply (*case-tac b*, *simp-all add*: *add-ac*)
done

lemma (**in** *semiring-0*) *poly-cmult-distr*: *a* %* (*p* +++ *q*) = (*a* %* *p* +++ *a* %* *q*)
apply (*induct p arbitrary*: *q*,*simp*)
apply (*case-tac q*, *simp-all add*: *right-distrib*)
done

lemma (**in** *ring-1*) *pmult-by-x*[*simp*]: [*0*, *1*] *** *t* = ((*0*)#*t*)
apply (*induct t*, *simp*)
apply (*auto simp add*: *mult-zero-left poly-ident-mult padd-commut*)
apply (*case-tac t*, *auto*)
done

properties of evaluation of polynomials.

lemma (**in** *semiring-0*) *poly-add*: *poly* (*p1* +++ *p2*) *x* = *poly p1 x* + *poly p2 x*
proof(*induct p1 arbitrary*: *p2*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a as p2*) **thus** *?case*
by (*cases p2*, *simp-all add*: *add-ac right-distrib*)
qed

lemma (**in** *comm-semiring-0*) *poly-cmult*: *poly* (*c* %* *p*) *x* = *c* * *poly p x*
apply (*induct p*)
apply (*case-tac* [*2*] *x*=*zero*)

apply (*auto simp add*: *right-distrib mult-ac*)
done

lemma (**in** *comm-semiring-0*) *poly-cmult-map*: *poly* (*map* (*op* ∗ *c*) *p*) *x* = *c*∗*poly p x*
by (*induct p*, *auto simp add*: *right-distrib mult-ac*)

lemma (**in** *comm-ring-1*) *poly-minus*: *poly* (−− *p*) *x* = − (*poly p x*)
apply (*simp add*: *poly-minus-def*)
apply (*auto simp add*: *poly-cmult minus-mult-left*[*symmetric*])
done

lemma (**in** *comm-semiring-0*) *poly-mult*: *poly* (*p1* ∗∗∗ *p2*) *x* = *poly p1 x* ∗ *poly p2 x*
proof(*induct p1 arbitrary*: *p2*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a as p2*)
thus *?case* **by** (*cases as*,
simp-all add: *poly-cmult poly-add left-distrib right-distrib mult-ac*)
qed

class *recpower-semiring* = *semiring* + *recpower*
class *recpower-semiring-1* = *semiring-1* + *recpower*
class *recpower-semiring-0* = *semiring-0* + *recpower*
class *recpower-ring* = *ring* + *recpower*
class *recpower-ring-1* = *ring-1* + *recpower*
subclass (**in** *recpower-ring-1*) *recpower-ring* **by** *unfold-locales*
class *recpower-comm-semiring-1* = *recpower* + *comm-semiring-1*
class *recpower-comm-ring-1* = *recpower* + *comm-ring-1*
subclass (**in** *recpower-comm-ring-1*) *recpower-comm-semiring-1* **by** *unfold-locales*
class *recpower-idom* = *recpower* + *idom*
subclass (**in** *recpower-idom*) *recpower-comm-ring-1* **by** *unfold-locales*
class *idom-char-0* = *idom* + *ring-char-0*
class *recpower-idom-char-0* = *recpower* + *idom-char-0*
subclass (**in** *recpower-idom-char-0*) *recpower-idom* **by** *unfold-locales*

lemma (**in** *recpower-comm-ring-1*) *poly-exp*: *poly* (*p* %ˆ *n*) *x* = (*poly p x*) ˆ *n*
apply (*induct n*)
apply (*auto simp add*: *poly-cmult poly-mult power-Suc*)
done

More Polynomial Evaluation Lemmas

lemma (**in** *semiring-0*) *poly-add-rzero*[*simp*]: *poly* (*a* +++ []) *x* = *poly a x*
by *simp*

lemma (**in** *comm-semiring-0*) *poly-mult-assoc*: *poly* ((*a* ∗∗∗ *b*) ∗∗∗ *c*) *x* = *poly* (*a* ∗∗∗ (*b* ∗∗∗ *c*)) *x*
by (*simp add*: *poly-mult mult-assoc*)

lemma (**in** *semiring-0*) *poly-mult-Nil2*[*simp*]: *poly* (*p* *** []) *x* = *0*
by (*induct p*, *auto*)

lemma (**in** *comm-semiring-1*) *poly-exp-add*: *poly* (*p* %^ (*n* + *d*)) *x* = *poly*(*p* %^ *n* *** *p* %^ *d*) *x*
apply (*induct n*)
apply (*auto simp add*: *poly-mult mult-assoc*)
done

52.2 Key Property: if *f a* = (*0*::'*a*) then *x* − *a* divides *p x*

lemma (**in** *comm-ring-1*) *lemma-poly-linear-rem*: ∀ *h*. ∃ *q r*. *h*#*t* = [*r*] +++ [−*a*, *1*] *** *q*
proof(*induct t*)
case *Nil*
{fix *h* **have** [*h*] = [*h*] +++ [− *a*, *1*] *** [] **by** *simp***}**
thus *?case* **by** *blast*
next
case (*Cons x xs*)
{fix *h*
from *Cons.hyps*[*rule-format*, *of x*]
obtain *q r* **where** *qr*: *x*#*xs* = [*r*] +++ [− *a*, *1*] *** *q* **by** *blast*
have *h*#*x*#*xs* = [*a*∗*r* + *h*] +++ [−*a*, *1*] *** (*r*#*q*)
using *qr* **by**(*cases q*, *simp-all add*: *ring-simps diff-def*[*symmetric*]
minus-mult-left[*symmetric*] *right-minus*)
hence ∃ *q r*. *h*#*x*#*xs* = [*r*] +++ [−*a*, *1*] *** *q* **by** *blast***}**
thus *?case* **by** *blast*
qed

lemma (**in** *comm-ring-1*) *poly-linear-rem*: ∃ *q r*. *h*#*t* = [*r*] +++ [−*a*, *1*] *** *q*
by (*cut-tac t* = *t* **and** *a* = *a* **in** *lemma-poly-linear-rem*, *auto*)

lemma (**in** *comm-ring-1*) *poly-linear-divides*: (*poly p a* = *0*) = ((*p* = []) | (∃ *q*. *p* = [−*a*, *1*] *** *q*))
proof−
{assume *p*: *p* = [] **hence** *?thesis* **by** *simp***}**
moreover
{fix *x xs* **assume** *p*: *p* = *x*#*xs*
{fix *q* **assume** *p* = [−*a*, *1*] *** *q* **hence** *poly p a* = *0*
by (*simp add*: *poly-add poly-cmult minus-mult-left*[*symmetric*])**}**
moreover
{assume *p0*: *poly p a* = *0*
from *poly-linear-rem*[*of x xs a*] **obtain** *q r*
where *qr*: *x*#*xs* = [*r*] +++ [− *a*, *1*] *** *q* **by** *blast*
have *r* = *0* **using** *p0* **by** (*simp only*: *p qr poly-mult poly-add*) *simp*
hence ∃ *q*. *p* = [− *a*, *1*] *** *q* **using** *p qr* **apply** − **apply** (*rule exI*[**where** *x*=*q*])**apply** *auto* **apply** (*cases q*) **apply** *auto* **done}**

ultimately have *?thesis* **using** *p* **by** *blast*}
ultimately show *?thesis* **by** (*cases p*, *auto*)
qed

lemma (**in** *semiring-0*) *lemma-poly-length-mult*[*simp*]: ∀ *h k a*. *length* (*k %* p +++* (*h # (a %* p)*)) = *Suc* (*length p*)
by (*induct p*, *auto*)

lemma (**in** *semiring-0*) *lemma-poly-length-mult2*[*simp*]: ∀ *h k*. *length* (*k %* p +++* (*h # p*)) = *Suc* (*length p*)
by (*induct p*, *auto*)

lemma (**in** *ring-1*) *poly-length-mult*[*simp*]: *length*([−*a*,*1*] ***** *q*) = *Suc* (*length q*)
by *auto*

52.3 Polynomial length

lemma (**in** *semiring-0*) *poly-cmult-length*[*simp*]: *length* (*a %* p*) = *length p*
by (*induct p*, *auto*)

lemma (**in** *semiring-0*) *poly-add-length*: *length* (*p1 +++ p2*) = *max* (*length p1*) (*length p2*)
apply (*induct p1 arbitrary*: *p2*, *simp-all*)
apply *arith*
done

lemma (**in** *semiring-0*) *poly-root-mult-length*[*simp*]: *length*([*a*,*b*] ***** *p*) = *Suc* (*length p*)
by (*simp add*: *poly-add-length*)

lemma (**in** *idom*) *poly-mult-not-eq-poly-Nil*[*simp*]:
poly (*p *** q*) *x* ≠ *poly* [] *x* ⟷ *poly p x* ≠ *poly* [] *x* ∧ *poly q x* ≠ *poly* [] *x*
by (*auto simp add*: *poly-mult*)

lemma (**in** *idom*) *poly-mult-eq-zero-disj*: *poly* (*p *** q*) *x* = *0* ⟷ *poly p x* = *0* ∨ *poly q x* = *0*
by (*auto simp add*: *poly-mult*)

Normalisation Properties

lemma (**in** *semiring-0*) *poly-normalized-nil*: (*pnormalize p* = []) −−> (*poly p x* = *0*)
by (*induct p*, *auto*)

A nontrivial polynomial of degree n has no more than n roots

lemma (**in** *idom*) *poly-roots-index-lemma*:
assumes *p*: *poly p x* ≠ *poly* [] *x* **and** *n*: *length p* = *n*
shows ∃ *i*. ∀ *x*. *poly p x* = *0* ⟶ (∃ *m*≤*n*. *x* = *i m*)
using *p n*
proof(*induct n arbitrary*: *p x*)
case *0* **thus** *?case* **by** *simp*

next
case (*Suc n p x*)
{assume *C*: $\bigwedge i.\ \exists x.\ poly\ p\ x = 0 \wedge (\forall m \leq Suc\ n.\ x \neq i\ m)$
from *Suc.prems* **have** *p0*: *poly p x ≠ 0 p≠ []* **by** *auto*
from *p0(1)*[*unfolded poly-linear-divides*[*of p x*]]
have $\forall q.\ p \neq [-x, 1] *** q$ **by** *blast*
from *C* **obtain** *a* **where** *a*: *poly p a = 0* **by** *blast*
from *a*[*unfolded poly-linear-divides*[*of p a*]] *p0(2)*
obtain *q* **where** *q*: $p = [-a, 1] *** q$ **by** *blast*
have *lg*: *length q = n* **using** *q Suc.prems(2)* **by** *simp*
from *q p0* **have** *qx*: *poly q x ≠ poly [] x*
by (*auto simp add*: *poly-mult poly-add poly-cmult*)
from *Suc.hyps*[*OF qx lg*] **obtain** *i* **where**
i: $\forall x.\ poly\ q\ x = 0 \longrightarrow (\exists m \leq n.\ x = i\ m)$ **by** *blast*
let *?i* = $\lambda m.$ *if m = Suc n then a else i m*
from *C*[*of ?i*] **obtain** *y* **where** *y*: *poly p y = 0* $\forall m \leq$ *Suc n. y ≠ ?i m*
by *blast*
from *y* **have** *y = a ∨ poly q y = 0*
by (*simp only*: *q poly-mult-eq-zero-disj poly-add*) (*simp add*: *ring-simps*)
with *i*[*rule-format*, *of y*] *y(1) y(2)* **have** *False* **apply** *auto*
apply (*erule-tac x=m* **in** *allE*)
apply *auto*
done}
thus *?case* **by** *blast*
qed

lemma (**in** *idom*) *poly-roots-index-length*: *poly p x ≠ poly [] x ==>*
$\exists i.\ \forall x.\ (poly\ p\ x = 0) --> (\exists n.\ n \leq length\ p\ \&\ x = i\ n)$
by (*blast intro*: *poly-roots-index-lemma*)

lemma (**in** *idom*) *poly-roots-finite-lemma1*: *poly p x ≠ poly [] x ==>*
$\exists N\ i.\ \forall x.\ (poly\ p\ x = 0) --> (\exists n.\ (n::nat) < N\ \&\ x = i\ n)$
apply (*drule poly-roots-index-length*, *safe*)
apply (*rule-tac x = Suc* (*length p*) **in** *exI*)
apply (*rule-tac x = i* **in** *exI*)
apply (*simp add*: *less-Suc-eq-le*)
done

lemma (**in** *idom*) *idom-finite-lemma*:
assumes *P*: $\forall x.\ P\ x --> (\exists n.\ n < length\ j\ \&\ x = j!n)$
shows *finite* $\{x.\ P\ x\}$
proof−
let *?M* = $\{x.\ P\ x\}$
let *?N* = *set j*
have *?M* $\subseteq$ *?N* **using** *P* **by** *auto*
thus *?thesis* **using** *finite-subset* **by** *auto*
qed

lemma (**in** *idom*) *poly-roots-finite-lemma2*: *poly p x* ≠ *poly* [] *x* ==>
  ∃ *i*. ∀ *x*. (*poly p x* = *0*) −−> *x* ∈ *set i*
apply (*drule poly-roots-index-length*, *safe*)
apply (*rule-tac x*=*map* (λ*n*. *i n*) [*0* ..< *Suc* (*length p*)] **in** *exI*)
apply (*auto simp add*: *image-iff*)
apply (*erule-tac x*=*x* **in** *allE*, *clarsimp*)
by (*case-tac n*=*length p*, *auto simp add*: *order-le-less*)

lemma *UNIV-nat-infinite*: ¬ *finite* (*UNIV* :: *nat set*)
  unfolding *finite-conv-nat-seg-image*
proof(*auto simp add*: *expand-set-eq image-iff*)
  fix *n*::*nat* **and** *f*:: *nat* ⇒ *nat*
  let *?N* = {*i*. *i* < *n*}
  let *?fN* = *f* ‘ *?N*
  let *?y* = *Max ?fN* + *1*
  from *nat-seg-image-imp-finite*[*of ?fN f n*]
  have *thfN*: *finite ?fN* **by** *simp*
  {assume *n* =*0* **hence** ∃ *x*. ∀ *xa*<*n*. *x* ≠ *f xa* **by** *auto***}**
  moreover
  {assume *nz*: *n* ≠ *0*
    hence *thne*: *?fN* ≠ {} **by** (*auto simp add*: *neq0-conv*)
    have ∀ *x*∈ *?fN*. *Max ?fN* ≥ *x* **using** *nz Max-ge-iff*[*OF thfN thne*] **by** *auto*
    hence ∀ *x*∈ *?fN*. *?y* > *x* **by** *auto*
    hence *?y* ∉ *?fN* **by** *auto*
    hence ∃ *x*. ∀ *xa*<*n*. *x* ≠ *f xa* **by** *auto* **}**
  ultimately show ∃ *x*. ∀ *xa*<*n*. *x* ≠ *f xa* **by** *blast*
qed

lemma (**in** *ring-char-0*) *UNIV-ring-char-0-infinte*:
  ¬ (*finite* (*UNIV*:: *'a set*))
proof
  assume *F*: *finite* (*UNIV* :: *'a set*)
  have *th0*: *of-nat* ‘ *UNIV* ⊆ *UNIV* **by** *simp*
  from *finite-subset*[*OF th0*] **have** *th*: *finite* (*of-nat* ‘ *UNIV* :: *'a set*) **.**
  have *th'*: *inj-on* (*of-nat*::*nat* ⇒ *'a*) (*UNIV*)
    unfolding *inj-on-def* **by** *auto*
  from *finite-imageD*[*OF th th'*] *UNIV-nat-infinite*
  show *False* **by** *blast*
qed

lemma (**in** *idom-char-0*) *poly-roots-finite*: (*poly p* ≠ *poly* []) =
  finite {*x*. *poly p x* = *0*}
proof
  assume *H*: *poly p* ≠ *poly* []
  show *finite* {*x*. *poly p x* = (*0*::*'a*)}
    using *H*
    apply −

apply (*erule contrapos-np*, *rule ext*)
apply (*rule ccontr*)
apply (*clarify dest*!: *poly-roots-finite-lemma2*)
using *finite-subset*
proof−
fix *x i*
assume *F*: ¬ *finite* {*x*. *poly p x* = (*0*::′*a*)}
and *P*: ∀ *x*. *poly p x* = (*0*::′*a*) ⟶ *x* ∈ *set i*
let *?M*= {*x*. *poly p x* = (*0*::′*a*)}
from *P* **have** *?M* ⊆ *set i* **by** *auto*
with *finite-subset F* **show** *False* **by** *auto*
qed
next
assume *F*: *finite* {*x*. *poly p x* = (*0*::′*a*)}
show *poly p* ≠ *poly* [] **using** *F UNIV-ring-char-0-infinte* **by** *auto*
qed

Entirety and Cancellation for polynomials

lemma (**in** *idom-char-0*) *poly-entire-lemma2*:
assumes *p0*: *poly p* ≠ *poly* [] **and** *q0*: *poly q* ≠ *poly* []
shows *poly* (*p*∗∗∗*q*) ≠ *poly* []
proof−
let *?S* = λ*p*. {*x*. *poly p x* = *0*}
have *?S* (*p* ∗∗∗ *q*) = *?S p* ∪ *?S q* **by** (*auto simp add*: *poly-mult*)
with *p0 q0* **show** *?thesis* **unfolding** *poly-roots-finite* **by** *auto*
qed

lemma (**in** *idom-char-0*) *poly-entire*:
poly (*p* ∗∗∗ *q*) = *poly* [] ⟷ *poly p* = *poly* [] ∨ *poly q* = *poly* []
using *poly-entire-lemma2*[*of p q*]
by *auto* (*rule ext*, *simp add*: *poly-mult*)+

lemma (**in** *idom-char-0*) *poly-entire-neg*: (*poly* (*p* ∗∗∗ *q*) ≠ *poly* []) = ((*poly p* ≠ *poly* []) & (*poly q* ≠ *poly* []))
by (*simp add*: *poly-entire*)

lemma *fun-eq*: (*f* = *g*) = (∀ *x*. *f x* = *g x*)
by (*auto intro*!: *ext*)

lemma (**in** *comm-ring-1*) *poly-add-minus-zero-iff*: (*poly* (*p* +++ −− *q*) = *poly* []) = (*poly p* = *poly q*)
by (*auto simp add*: *ring-simps poly-add poly-minus-def fun-eq poly-cmult minus-mult-left*[*symmetric*])

lemma (**in** *comm-ring-1*) *poly-add-minus-mult-eq*: *poly* (*p* ∗∗∗ *q* +++ −−(*p* ∗∗∗ *r*)) = *poly* (*p* ∗∗∗ (*q* +++ −− *r*))
by (*auto simp add*: *poly-add poly-minus-def fun-eq poly-mult poly-cmult right-distrib minus-mult-left*[*symmetric*] *minus-mult-right*[*symmetric*])

subclass (**in** *idom-char-0*) *comm-ring-1* **by** *unfold-locales*

lemma (**in** *idom-char-0*) *poly-mult-left-cancel*: (*poly* (*p* *** *q*) = *poly* (*p* *** *r*)) = (*poly p* = *poly* [] | *poly q* = *poly r*)
proof−
have *poly* (*p* *** *q*) = *poly* (*p* *** *r*) ⟷ *poly* (*p* *** *q* +++ −− (*p* *** *r*)) = *poly* [] **by** (*simp only*: *poly-add-minus-zero-iff*)
also have ... ⟷ *poly p* = *poly* [] | *poly q* = *poly r*
by (*auto intro*: *ext simp add*: *poly-add-minus-mult-eq poly-entire poly-add-minus-zero-iff*)
finally show *?thesis* **.**
qed

lemma (**in** *recpower-idom*) *poly-exp-eq-zero*[*simp*]:
(*poly* (*p* %^ *n*) = *poly* []) = (*poly p* = *poly* [] & *n* ≠ *0*)
apply (*simp only*: *fun-eq add*: *all-simps* [*symmetric*])
apply (*rule arg-cong* [**where** *f* = *All*])
apply (*rule ext*)
apply (*induct n*)
apply (*auto simp add*: *poly-exp poly-mult*)
done

lemma (**in** *semiring-1*) *one-neq-zero*[*simp*]: *1* ≠ *0* **using** *zero-neq-one* **by** *blast*
lemma (**in** *comm-ring-1*) *poly-prime-eq-zero*[*simp*]: *poly* [*a*,*1*] ≠ *poly* []
apply (*simp add*: *fun-eq*)
apply (*rule-tac x* = *minus one a* **in** *exI*)
apply (*unfold diff-minus*)
apply (*subst add-commute*)
apply (*subst add-assoc*)
apply *simp*
done

lemma (**in** *recpower-idom*) *poly-exp-prime-eq-zero*: (*poly* ([*a*, *1*] %^ *n*) ≠ *poly* [])
by *auto*

A more constructive notion of polynomials being trivial

lemma (**in** *idom-char-0*) *poly-zero-lemma'*: *poly* (*h* # *t*) = *poly* [] ==> *h* = *0* & *poly t* = *poly* []
apply(*simp add*: *fun-eq*)
apply (*case-tac h* = *zero*)
apply (*drule-tac* [*2*] *x* = *zero* **in** *spec*, *auto*)
apply (*cases poly t* = *poly* [], *simp*)
proof−
fix *x*
assume *H*: ∀ *x*. *x* = (*0*::'*a*) ∨ *poly t x* = (*0*::'*a*) **and** *pnz*: *poly t* ≠ *poly* []
let *?S* = {*x*. *poly t x* = *0*}
from *H* **have** ∀ *x*. *x* ≠*0* ⟶ *poly t x* = *0* **by** *blast*
hence *th*: *?S* ⊇ *UNIV* − {*0*} **by** *auto*
from *poly-roots-finite pnz* **have** *th'*: *finite ?S* **by** *blast*
from *finite-subset*[*OF th th'*] *UNIV-ring-char-0-infinte*
show *poly t x* = (*0*::'*a*) **by** *simp*
qed

lemma (**in** *idom-char-0*) *poly-zero*: (*poly p* = *poly* []) = *list-all* (%*c*. *c* = *0*) *p*
apply (*induct p*, *simp*)
apply (*rule iffI*)
apply (*drule poly-zero-lemma'*, *auto*)
done

lemma (**in** *idom-char-0*) *poly-0*: *list-all* (λ*c*. *c* = *0*) *p* ⟹ *poly p x* = *0*
unfolding *poly-zero*[*symmetric*] **by** *simp*

Basics of divisibility.

lemma (**in** *idom*) *poly-primes*: ([*a*, *1*] *divides* (*p* ∗∗∗ *q*)) = ([*a*, *1*] *divides p* | [*a*, *1*] *divides q*)
apply (*auto simp add*: *divides-def fun-eq poly-mult poly-add poly-cmult left-distrib* [*symmetric*])
apply (*drule-tac x* = *uminus a* **in** *spec*)
apply (*simp add*: *poly-linear-divides poly-add poly-cmult left-distrib* [*symmetric*])
apply (*cases p* = [])
apply (*rule exI*[**where** *x*=[]])
apply *simp*
apply (*cases q* = [])
apply (*erule allE*[**where** *x*=[]], *simp*)

apply *clarsimp*
apply (*cases* ∃ *q*::'*a list*. *p* = *a* %∗ *q* +++ ((*0*::'*a*) # *q*))
apply (*clarsimp simp add*: *poly-add poly-cmult*)
apply (*rule-tac x*=*qa* **in** *exI*)
apply (*simp add*: *left-distrib* [*symmetric*])
apply *clarsimp*

apply (*auto simp add*: *right-minus poly-linear-divides poly-add poly-cmult left-distrib* [*symmetric*])
apply (*rule-tac x* = *pmult qa q* **in** *exI*)
apply (*rule-tac* [*2*] *x* = *pmult p qa* **in** *exI*)
apply (*auto simp add*: *poly-add poly-mult poly-cmult mult-ac*)
done

lemma (**in** *comm-semiring-1*) *poly-divides-refl*[*simp*]: *p divides p*
apply (*simp add*: *divides-def*)
apply (*rule-tac x* = [*one*] **in** *exI*)
apply (*auto simp add*: *poly-mult fun-eq*)
done

lemma (**in** *comm-semiring-1*) *poly-divides-trans*: [| *p divides q*; *q divides r* |] ==> *p divides r*
apply (*simp add*: *divides-def*, *safe*)
apply (*rule-tac x* = *pmult qa qaa* **in** *exI*)
apply (*auto simp add*: *poly-mult fun-eq mult-assoc*)
done

lemma (**in** *recpower-comm-semiring-1*) *poly-divides-exp*: $m \leq n$ ==> $(p \,\%\hat{}\, m)$ *divides* $(p \,\%\hat{}\, n)$
apply (*auto simp add*: *le-iff-add*)
apply (*induct-tac k*)
apply (*rule-tac* [*2*] *poly-divides-trans*)
apply (*auto simp add*: *divides-def*)
apply (*rule-tac x* = *p* **in** *exI*)
apply (*auto simp add*: *poly-mult fun-eq mult-ac*)
done

lemma (**in** *recpower-comm-semiring-1*) *poly-exp-divides*: [| $(p \,\%\hat{}\, n)$ *divides q*; $m \leq n$ |] ==> $(p \,\%\hat{}\, m)$ *divides q*
by (*blast intro*: *poly-divides-exp poly-divides-trans*)

lemma (**in** *comm-semiring-0*) *poly-divides-add*:
[| *p divides q*; *p divides r* |] ==> *p divides* $(q +\!\!+\!\!+ r)$
apply (*simp add*: *divides-def*, *auto*)
apply (*rule-tac x* = *padd qa qaa* **in** *exI*)
apply (*auto simp add*: *poly-add fun-eq poly-mult right-distrib*)
done

lemma (**in** *comm-ring-1*) *poly-divides-diff*:
[| *p divides q*; *p divides* $(q +\!\!+\!\!+ r)$ |] ==> *p divides r*
apply (*simp add*: *divides-def*, *auto*)
apply (*rule-tac x* = *padd qaa* (*poly-minus qa*) **in** *exI*)
apply (*auto simp add*: *poly-add fun-eq poly-mult poly-minus right-diff-distrib compare-rls add-ac*)
done

lemma (**in** *comm-ring-1*) *poly-divides-diff2*: [| *p divides r*; *p divides* $(q +\!\!+\!\!+ r)$ |] ==> *p divides q*
apply (*erule poly-divides-diff*)
apply (*auto simp add*: *poly-add fun-eq poly-mult divides-def add-ac*)
done

lemma (**in** *semiring-0*) *poly-divides-zero*: *poly p* = *poly* [] ==> *q divides p*
apply (*simp add*: *divides-def*)
apply (*rule exI*[**where** *x*=[]])
apply (*auto simp add*: *fun-eq poly-mult*)
done

lemma (**in** *semiring-0*) *poly-divides-zero2*[*simp*]: *q divides* []
apply (*simp add*: *divides-def*)
apply (*rule-tac x* = [] **in** *exI*)
apply (*auto simp add*: *fun-eq*)
done

At last, we can consider the order of a root.

lemma (**in** *idom-char-0*) *poly-order-exists-lemma*:
assumes *lp*: *length p* = *d* **and** *p*: *poly p* ≠ *poly* []
shows ∃ *n q*. *p* = *mulexp n* [−*a*, *1*] *q* ∧ *poly q a* ≠ *0*
using *lp p*
proof(*induct d arbitrary*: *p*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n p*)
{assume *p0*: *poly p a* = *0*
from *Suc.prems* **have** *h*: *length p* = *Suc n poly p* ≠ *poly* [] **by** *blast*
hence *pN*: *p* ≠ [] **by** − (*rule ccontr*, *simp*)
from *p0*[*unfolded poly-linear-divides*] *pN* **obtain** *q* **where**
q: *p* = [−*a*, *1*] **** q* **by** *blast*
from *q h p0* **have** *qh*: *length q* = *n poly q* ≠ *poly* []
apply −
apply *simp*
apply (*simp only*: *fun-eq*)
apply (*rule ccontr*)
apply (*simp add*: *fun-eq poly-add poly-cmult minus-mult-left*[*symmetric*])
done
from *Suc.hyps*[*OF qh*] **obtain** *m r* **where**
mr: *q* = *mulexp m* [−*a*,*1*] *r poly r a* ≠ *0* **by** *blast*
from *mr q* **have** *p* = *mulexp* (*Suc m*) [−*a*,*1*] *r* ∧ *poly r a* ≠ *0* **by** *simp*
hence *?case* **by** *blast***}**
moreover
{assume *p0*: *poly p a* ≠ *0*
hence *?case* **using** *Suc.prems* **apply** *simp* **by** (*rule exI*[**where** *x*=*0*::*nat*], *simp*)**}**
ultimately show *?case* **by** *blast*
qed

lemma (**in** *recpower-comm-semiring-1*) *poly-mulexp*: *poly* (*mulexp n p q*) *x* = (*poly p x*) ^ *n* ∗ *poly q x*
by(*induct n*, *auto simp add*: *poly-mult power-Suc mult-ac*)

lemma (**in** *comm-semiring-1*) *divides-left-mult*:
assumes *d*:(*p***q*) *divides r* **shows** *p divides r* ∧ *q divides r*
proof−
from *d* **obtain** *t* **where** *r*:*poly r* = *poly* (*p***q *** t*)
unfolding *divides-def* **by** *blast*
hence *poly r* = *poly* (*p *** q *** t*))
poly r = *poly* (*q *** p***t*)) **by**(*auto simp add*: *fun-eq poly-mult mult-ac*)
thus *?thesis* **unfolding** *divides-def* **by** *blast*
qed

lemma (**in** *recpower-semiring-1*)
zero-power-iff: *0 ^ n = (if n = 0 then 1 else 0)*
by (*induct n*, *simp-all add*: *power-Suc*)

lemma (**in** *recpower-idom-char-0*) *poly-order-exists*:
assumes *lp*: *length p = d* **and** *p0*: *poly p ≠ poly []*
shows ∃ *n*. ([−*a*, *1*] %^ *n*) *divides p* & ~(([−*a*, *1*] %^ (*Suc n*)) *divides p*)
proof−
let *?poly = poly*
let *?mulexp = mulexp*
let *?pexp = pexp*
from *lp p0*
show *?thesis*
apply −
apply (*drule poly-order-exists-lemma* [**where** *a*=*a*], *assumption*, *clarify*)
apply (*rule-tac x = n* **in** *exI*, *safe*)
apply (*unfold divides-def*)
apply (*rule-tac x = q* **in** *exI*)
apply (*induct-tac n*, *simp*)
apply (*simp* (*no-asm-simp*) *add*: *poly-add poly-cmult poly-mult right-distrib mult-ac*)
apply *safe*
apply (*subgoal-tac ?poly* (*?mulexp n* [*uminus a*, *one*] *q*) ≠ *?poly* (*pmult* (*?pexp* [*uminus a*, *one*] (*Suc n*)) *qa*))
apply *simp*
apply (*induct-tac n*)
apply (*simp del*: *pmult-Cons pexp-Suc*)
apply (*erule-tac Q = ?poly q a = zero* **in** *contrapos-np*)
apply (*simp add*: *poly-add poly-cmult minus-mult-left*[*symmetric*])
apply (*rule pexp-Suc* [*THEN ssubst*])
apply (*rule ccontr*)
apply (*simp add*: *poly-mult-left-cancel poly-mult-assoc del*: *pmult-Cons pexp-Suc*)
done
qed

lemma (**in** *semiring-1*) *poly-one-divides*[*simp*]: [*1*] *divides p*
by (*simp add*: *divides-def*, *auto*)

lemma (**in** *recpower-idom-char-0*) *poly-order*: *poly p ≠ poly []*
==> *EX*! *n*. ([−*a*, *1*] %^ *n*) *divides p* &
~(([−*a*, *1*] %^ (*Suc n*)) *divides p*)
apply (*auto intro*: *poly-order-exists simp add*: *less-linear simp del*: *pmult-Cons pexp-Suc*)
apply (*cut-tac x = y* **and** *y = n* **in** *less-linear*)
apply (*drule-tac m = n* **in** *poly-exp-divides*)
apply (*auto dest*: *Suc-le-eq* [*THEN iffD2*, *THEN* [*2*] *poly-exp-divides*]
simp del: *pmult-Cons pexp-Suc*)
done

Order

lemma *some1-equalityD*: [| *n* = (@*n*. *P n*); *EX*! *n*. *P n* |] ==> *P n*
by (*blast intro*: *someI2*)

lemma (**in** *recpower-idom-char-0*) *order*:
((|[−*a*, *1*] % ^ *n*) *divides p* &
~((|[−*a*, *1*] % ^ (*Suc n*)) *divides p*)) =
((*n* = *order a p*) & ~(*poly p* = *poly* []))
apply (*unfold order-def*)
apply (*rule iffI*)
apply (*blast dest*: *poly-divides-zero intro*!: *some1-equality* [*symmetric*] *poly-order*)
apply (*blast intro*!: *poly-order* [*THEN* [*2*] *some1-equalityD*])
done

lemma (**in** *recpower-idom-char-0*) *order2*: [| *poly p* ≠ *poly* [] |]
==> (|[−*a*, *1*] % ^ (*order a p*)) *divides p* &
~((|[−*a*, *1*] % ^ (*Suc*(*order a p*))) *divides p*)
by (*simp add*: *order del*: *pexp-Suc*)

lemma (**in** *recpower-idom-char-0*) *order-unique*: [| *poly p* ≠ *poly* []; (|[−*a*, *1*] % ^ *n*) *divides p*;
~((|[−*a*, *1*] % ^ (*Suc n*)) *divides p*)
|] ==> (*n* = *order a p*)
by (*insert order* [*of a n p*], *auto*)

lemma (**in** *recpower-idom-char-0*) *order-unique-lemma*: (*poly p* ≠ *poly* [] & (|[−*a*, *1*] % ^ *n*) *divides p* &
~((|[−*a*, *1*] % ^ (*Suc n*)) *divides p*))
==> (*n* = *order a p*)
by (*blast intro*: *order-unique*)

lemma (**in** *ring-1*) *order-poly*: *poly p* = *poly q* ==> *order a p* = *order a q*
by (*auto simp add*: *fun-eq divides-def poly-mult order-def*)

lemma (**in** *semiring-1*) *pexp-one*[*simp*]: *p* % ^ (*Suc 0*) = *p*
apply (*induct p*)
apply (*auto simp add*: *numeral-1-eq-1*)
done

lemma (**in** *comm-ring-1*) *lemma-order-root*:
0 < *n* & [− *a*, *1*] % ^ *n divides p* & ~ [− *a*, *1*] % ^ (*Suc n*) *divides p*
==> *poly p a* = *0*
apply (*induct n arbitrary*: *a p*, *blast*)
apply (*auto simp add*: *divides-def poly-mult simp del*: *pmult-Cons*)
done

lemma (**in** *recpower-idom-char-0*) *order-root*: (*poly p a* = *0*) = ((*poly p* = *poly* []) | *order a p* ≠ *0*)
proof −

let *?poly = poly*
show *?thesis*
apply (*case-tac ?poly p = ?poly* [], *auto*)
apply (*simp add*: *poly-linear-divides del*: *pmult-Cons*, *safe*)
apply (*drule-tac* [!] *a = a* **in** *order2*)
apply (*rule ccontr*)
apply (*simp add*: *divides-def poly-mult fun-eq del*: *pmult-Cons*, *blast*)
using *neq0-conv*
apply (*blast intro*: *lemma-order-root*)
done
qed

lemma (**in** *recpower-idom-char-0*) *order-divides*: (([−*a*, *1*] %^ *n*) *divides p*) = ((*poly p = poly* []) | *n* ≤ *order a p*)
proof−
let *?poly = poly*
show *?thesis*
apply (*case-tac ?poly p = ?poly* [], *auto*)
apply (*simp add*: *divides-def fun-eq poly-mult*)
apply (*rule-tac x = []* **in** *exI*)
apply (*auto dest*!: *order2* [**where** *a*=*a*]
intro: *poly-exp-divides simp del*: *pexp-Suc*)
done
qed

lemma (**in** *recpower-idom-char-0*) *order-decomp*:
poly p ≠ *poly* []
==> ∃ *q*. (*poly p = poly* (([−*a*, *1*] %^ (*order a p*)) *** *q*)) &
~([−*a*, *1*] *divides q*)
apply (*unfold divides-def*)
apply (*drule order2* [**where** *a = a*])
apply (*simp add*: *divides-def del*: *pexp-Suc pmult-Cons*, *safe*)
apply (*rule-tac x = q* **in** *exI*, *safe*)
apply (*drule-tac x = qa* **in** *spec*)
apply (*auto simp add*: *poly-mult fun-eq poly-exp mult-ac simp del*: *pmult-Cons*)
done

Important composition properties of orders.

lemma *order-mult*: *poly* (*p* *** *q*) ≠ *poly* []
==> *order a* (*p* *** *q*) = *order a p* + *order* (*a*::'*a*::{*recpower-idom-char-0*}) *q*
apply (*cut-tac a = a* **and** *p = p* *** *q* **and** *n = order a p* + *order a q* **in** *order*)
apply (*auto simp add*: *poly-entire simp del*: *pmult-Cons*)
apply (*drule-tac a = a* **in** *order2*)+
apply *safe*
apply (*simp add*: *divides-def fun-eq poly-exp-add poly-mult del*: *pmult-Cons*, *safe*)
apply (*rule-tac x = qa* *** *qaa* **in** *exI*)
apply (*simp add*: *poly-mult mult-ac del*: *pmult-Cons*)
apply (*drule-tac a = a* **in** *order-decomp*)+

apply *safe*
apply (*subgoal-tac* [−*a*,*1*] *divides* (*qa* *** *qaa*))
apply (*simp add*: *poly-primes del*: *pmult-Cons*)
apply (*auto simp add*: *divides-def simp del*: *pmult-Cons*)
apply (*rule-tac x* = *qb* **in** *exI*)
apply (*subgoal-tac poly* ([−*a*, *1*] % ^ (*order a p*) *** (*qa* *** *qaa*)) = *poly* ([−*a*, *1*] % ^ (*order a p*) *** ([−*a*, *1*] *** *qb*)))
apply (*drule poly-mult-left-cancel* [*THEN iffD1*], *force*)
apply (*subgoal-tac poly* ([−*a*, *1*] % ^ (*order a q*) *** ([−*a*, *1*] % ^ (*order a p*) *** (*qa* *** *qaa*))) = *poly* ([−*a*, *1*] % ^ (*order a q*) *** ([−*a*, *1*] % ^ (*order a p*) *** ([−*a*, *1*] *** *qb*))))
apply (*drule poly-mult-left-cancel* [*THEN iffD1*], *force*)
apply (*simp add*: *fun-eq poly-exp-add poly-mult mult-ac del*: *pmult-Cons*)
done

lemma (**in** *recpower-idom-char-0*) *order-mult*:
  assumes *pq0*: *poly* (*p* *** *q*) ≠ *poly* []
  shows *order a* (*p* *** *q*) = *order a p* + *order a q*
proof−
  let *?order* = *order*
  let *?divides* = *op divides*
  let *?poly* = *poly*
from *pq0*
show *?thesis*
apply (*cut-tac a* = *a* **and** *p* = *pmult p q* **and** *n* = *?order a p* + *?order a q* **in** *order*)
apply (*auto simp add*: *poly-entire simp del*: *pmult-Cons*)
apply (*drule-tac a* = *a* **in** *order2*)+
apply *safe*
apply (*simp add*: *divides-def fun-eq poly-exp-add poly-mult del*: *pmult-Cons*, *safe*)
apply (*rule-tac x* = *pmult qa qaa* **in** *exI*)
apply (*simp add*: *poly-mult mult-ac del*: *pmult-Cons*)
apply (*drule-tac a* = *a* **in** *order-decomp*)+
apply *safe*
apply (*subgoal-tac ?divides* [*uminus a*,*one*] (*pmult qa qaa*))
apply (*simp add*: *poly-primes del*: *pmult-Cons*)
apply (*auto simp add*: *divides-def simp del*: *pmult-Cons*)
apply (*rule-tac x* = *qb* **in** *exI*)
apply (*subgoal-tac ?poly* (*pmult* (*pexp* [*uminus a*, *one*] (*?order a p*)) (*pmult qa qaa*)) = *?poly* (*pmult* (*pexp* [*uminus a*, *one*] (*?order a p*)) (*pmult* [*uminus a*, *one*] *qb*)))
apply (*drule poly-mult-left-cancel* [*THEN iffD1*], *force*)
apply (*subgoal-tac ?poly* (*pmult* (*pexp* [*uminus a*, *one*] (*order a q*)) (*pmult* (*pexp* [*uminus a*, *one*] (*order a p*)) (*pmult qa qaa*))) = *?poly* (*pmult* (*pexp* [*uminus a*, *one*] (*order a q*)) (*pmult* (*pexp* [*uminus a*, *one*] (*order a p*)) (*pmult* [*uminus a*, *one*] *qb*))))
apply (*drule poly-mult-left-cancel* [*THEN iffD1*], *force*)
apply (*simp add*: *fun-eq poly-exp-add poly-mult mult-ac del*: *pmult-Cons*)
done

qed

lemma (**in** *recpower-idom-char-0*) *order-root2*: *poly p* ≠ *poly* [] ==> (*poly p a* = *0*) = (*order a p* ≠ *0*)
by (*rule order-root* [*THEN ssubst*], *auto*)

lemma (**in** *semiring-1*) *pmult-one*[*simp*]: [*1*] *** *p* = *p* **by** *auto*

lemma (**in** *semiring-0*) *poly-Nil-zero*: *poly* [] = *poly* [*0*]
by (*simp add*: *fun-eq*)

lemma (**in** *recpower-idom-char-0*) *rsquarefree-decomp*:
  [| *rsquarefree p*; *poly p a* = *0* |]
  ==> ∃ *q*. (*poly p* = *poly* ([−*a*, *1*] *** *q*)) & *poly q a* ≠ *0*
apply (*simp add*: *rsquarefree-def*, *safe*)
apply (*frule-tac a* = *a* **in** *order-decomp*)
apply (*drule-tac x* = *a* **in** *spec*)
apply (*drule-tac a* = *a* **in** *order-root2* [*symmetric*])
apply (*auto simp del*: *pmult-Cons*)
apply (*rule-tac x* = *q* **in** *exI*, *safe*)
apply (*simp add*: *poly-mult fun-eq*)
apply (*drule-tac p1* = *q* **in** *poly-linear-divides* [*THEN iffD1*])
apply (*simp add*: *divides-def del*: *pmult-Cons*, *safe*)
apply (*drule-tac x* = [] **in** *spec*)
apply (*auto simp add*: *fun-eq*)
done

Normalization of a polynomial.

lemma (**in** *semiring-0*) *poly-normalize*[*simp*]: *poly* (*pnormalize p*) = *poly p*
apply (*induct p*)
apply (*auto simp add*: *fun-eq*)
done

The degree of a polynomial.

lemma (**in** *semiring-0*) *lemma-degree-zero*:
  list-all (%*c*. *c* = *0*) *p* ⟷ *pnormalize p* = []
by (*induct p*, *auto*)

lemma (**in** *idom-char-0*) *degree-zero*:
 assumes *pN*: *poly p* = *poly* [] **shows***degree p* = *0*
proof−
 let *?pn* = *pnormalize*
 from *pN*
 show *?thesis*
  apply (*simp add*: *degree-def*)
  apply (*case-tac ?pn p* = [])
  apply (*auto simp add*: *poly-zero lemma-degree-zero*)
  done
qed

lemma (**in** *semiring-0*) *pnormalize-sing*: (*pnormalize* [*x*] = [*x*]) ⟷ *x* ≠ *0* **by** *simp*
lemma (**in** *semiring-0*) *pnormalize-pair*: *y* ≠ *0* ⟷ (*pnormalize* [*x*, *y*] = [*x*, *y*]) **by** *simp*
lemma (**in** *semiring-0*) *pnormal-cons*: *pnormal p* ⟹ *pnormal* (*c*#*p*)
  unfolding *pnormal-def* **by** *simp*
lemma (**in** *semiring-0*) *pnormal-tail*: *p*≠[] ⟹ *pnormal* (*c*#*p*) ⟹ *pnormal p*
  unfolding *pnormal-def*
  apply (*cases pnormalize p* = [], *auto*)
  by (*cases c* = *0*, *auto*)

lemma (**in** *semiring-0*) *pnormal-last-nonzero*: *pnormal p* ==> *last p* ≠ *0*
proof(*induct p*)
  case *Nil* **thus** *?case* **by** (*simp add*: *pnormal-def*)
next
  case (*Cons a as*) **thus** *?case*
    apply (*simp add*: *pnormal-def*)
    apply (*cases pnormalize as* = [], *simp-all*)
    apply (*cases as* = [], *simp-all*)
    apply (*cases a*=*0*, *simp-all*)
    apply (*cases a*=*0*, *simp-all*)
    done
qed

lemma (**in** *semiring-0*) *pnormal-length*: *pnormal p* ⟹ *0* < *length p*
  unfolding *pnormal-def length-greater-0-conv* **by** *blast*

lemma (**in** *semiring-0*) *pnormal-last-length*: ⟦*0* < *length p* ; *last p* ≠ *0*⟧ ⟹ *pnormal p*
  apply (*induct p*, *auto*)
  apply (*case-tac p* = [], *auto*)
  apply (*simp add*: *pnormal-def*)
  by (*rule pnormal-cons*, *auto*)

lemma (**in** *semiring-0*) *pnormal-id*: *pnormal p* ⟷ (*0* < *length p* ∧ *last p* ≠ *0*)
  using *pnormal-last-length pnormal-length pnormal-last-nonzero* **by** *blast*

lemma (**in** *idom-char-0*) *poly-Cons-eq*: *poly* (*c*#*cs*) = *poly* (*d*#*ds*) ⟷ *c*=*d* ∧ *poly cs* = *poly ds* (**is** *?lhs* ⟷ *?rhs*)
proof
  assume *eq*: *?lhs*
  hence ⋀*x*. *poly* ((*c*#*cs*) +++ −− (*d*#*ds*)) *x* = *0*
    by (*simp only*: *poly-minus poly-add ring-simps*) *simp*
  hence *poly* ((*c*#*cs*) +++ −− (*d*#*ds*)) = *poly* [] **by** − (*rule ext*, *simp*)
  hence *c* = *d* ∧ *list-all* (λ*x*. *x*=*0*) ((*cs* +++ −− *ds*))
   unfolding *poly-zero* **by** (*simp add*: *poly-minus-def ring-simps minus-mult-left*[*symmetric*])
  hence *c* = *d* ∧ (∀ *x*. *poly* (*cs* +++ −− *ds*) *x* = *0*)

unfolding *poly-zero*[*symmetric*] **by** *simp*
thus *?rhs* **apply** (*simp add*: *poly-minus poly-add ring-simps*) **apply** (*rule ext*, *simp*) **done**
next
assume *?rhs* **then show** *?lhs* **by** − (*rule ext*,*simp*)
qed

lemma (**in** *idom-char-0*) *pnormalize-unique*: *poly p* = *poly q* ⟹ *pnormalize p* = *pnormalize q*
proof(*induct q arbitrary*: *p*)
case *Nil* **thus** *?case* **by** (*simp only*: *poly-zero lemma-degree-zero*) *simp*
next
case (*Cons c cs p*)
thus *?case*
proof(*induct p*)
case *Nil*
hence *poly* [] = *poly* (*c*#*cs*) **by** *blast*
then have *poly* (*c*#*cs*) = *poly* [] **by** *simp*
thus *?case* **by** (*simp only*: *poly-zero lemma-degree-zero*) *simp*
next
case (*Cons d ds*)
hence *eq*: *poly* (*d* # *ds*) = *poly* (*c* # *cs*) **by** *blast*
hence *eq'*: ⋀*x*. *poly* (*d* # *ds*) *x* = *poly* (*c* # *cs*) *x* **by** *simp*
hence *poly* (*d* # *ds*) *0* = *poly* (*c* # *cs*) *0* **by** *blast*
hence *dc*: *d* = *c* **by** *auto*
with *eq* **have** *poly ds* = *poly cs*
unfolding *poly-Cons-eq* **by** *simp*
with *Cons.prems* **have** *pnormalize ds* = *pnormalize cs* **by** *blast*
with *dc* **show** *?case* **by** *simp*
qed
qed

lemma (**in** *idom-char-0*) *degree-unique*: **assumes** *pq*: *poly p* = *poly q*
shows *degree p* = *degree q*
using *pnormalize-unique*[*OF pq*] **unfolding** *degree-def* **by** *simp*

lemma (**in** *semiring-0*) *pnormalize-length*: *length* (*pnormalize p*) ≤ *length p* **by** (*induct p*, *auto*)

lemma (**in** *semiring-0*) *last-linear-mul-lemma*:
last ((*a* %* *p*) +++ (*x*#(*b* %* *p*))) = (*if p*=[] *then x else b*∗*last p*)

apply (*induct p arbitrary*: *a x b*, *auto*)
apply (*subgoal-tac padd* (*cmult aa p*) (*times b a* # *cmult b p*) ≠ [], *simp*)
apply (*induct-tac p*, *auto*)
done

lemma (**in** *semiring-1*) *last-linear-mul*: **assumes** *p*:*p*≠[] **shows** *last* ([*a*,*1*] ∗∗∗ *p*) = *last p*

proof−
from *p* **obtain** *c cs* **where** *cs*: *p* = *c*#*cs* **by** (*cases p*, *auto*)
from *cs* **have** *eq*:[*a*,*1*] *** *p* = (*a* %* (*c*#*cs*)) +++ (*0*#(*1* %* (*c*#*cs*)))
by (*simp add*: *poly-cmult-distr*)
show *?thesis* **using** *cs*
unfolding *eq last-linear-mul-lemma* **by** *simp*
qed

lemma (**in** *semiring-0*) *pnormalize-eq*: *last p* ≠ *0* ⟹ *pnormalize p* = *p*
apply (*induct p*, *auto*)
apply (*case-tac p*, *auto*)+
done

lemma (**in** *semiring-0*) *last-pnormalize*: *pnormalize p* ≠ [] ⟹ *last* (*pnormalize p*) ≠ *0*
by (*induct p*, *auto*)

lemma (**in** *semiring-0*) *pnormal-degree*: *last p* ≠ *0* ⟹ *degree p* = *length p* − *1*
using *pnormalize-eq*[*of p*] **unfolding** *degree-def* **by** *simp*

lemma (**in** *semiring-0*) *poly-Nil-ext*: *poly* [] = (λ*x*. *0*) **by** (*rule ext*) *simp*

lemma (**in** *idom-char-0*) *linear-mul-degree*: **assumes** *p*: *poly p* ≠ *poly* []
shows *degree* ([*a*,*1*] *** *p*) = *degree p* + *1*
proof−
from *p* **have** *pnz*: *pnormalize p* ≠ []
unfolding *poly-zero lemma-degree-zero* **.**

from *last-linear-mul*[*OF pnz*, *of a*] *last-pnormalize*[*OF pnz*]
have *l0*: *last* ([*a*, *1*] *** *pnormalize p*) ≠ *0* **by** *simp*
from *last-pnormalize*[*OF pnz*] *last-linear-mul*[*OF pnz*, *of a*]
pnormal-degree[*OF l0*] *pnormal-degree*[*OF last-pnormalize*[*OF pnz*]] *pnz*

have *th*: *degree* ([*a*,*1*] *** *pnormalize p*) = *degree* (*pnormalize p*) + *1*
by (*auto simp add*: *poly-length-mult*)

have *eqs*: *poly* ([*a*,*1*] *** *pnormalize p*) = *poly* ([*a*,*1*] *** *p*)
by (*rule ext*) (*simp add*: *poly-mult poly-add poly-cmult*)
from *degree-unique*[*OF eqs*] *th*
show *?thesis* **by** (*simp add*: *degree-unique*[*OF poly-normalize*])
qed

lemma (**in** *idom-char-0*) *linear-pow-mul-degree*:
degree([*a*,*1*] %^*n* *** *p*) = (*if poly p* = *poly* [] *then 0 else degree p* + *n*)
proof(*induct n arbitrary*: *a p*)
case (*0 a p*)
{assume *p*: *poly p* = *poly* []
hence *?case* **using** *degree-unique*[*OF p*] **by** (*simp add*: *degree-def*)**}**

moreover
{assume *p*: *poly p* ≠ *poly* [] **hence** *?case* **by** (*auto simp add*: *poly-Nil-ext*) **}**
ultimately show *?case* **by** *blast*
next
case (*Suc n a p*)
have *eq*: *poly* ([*a*,*1*] %^(*Suc n*) *** *p*) = *poly* ([*a*,*1*] %^ *n* *** ([*a*,*1*] *** *p*))
apply (*rule ext*, *simp add*: *poly-mult poly-add poly-cmult*)
by (*simp add*: *mult-ac add-ac right-distrib*)
note *deq* = *degree-unique*[*OF eq*]
{assume *p*: *poly p* = *poly* []
with *eq* **have** *eq'*: *poly* ([*a*,*1*] %^(*Suc n*) *** *p*) = *poly* []
by − (*rule ext*,*simp add*: *poly-mult poly-cmult poly-add*)
from *degree-unique*[*OF eq'*] *p* **have** *?case* **by** (*simp add*: *degree-def*)**}**
moreover
{assume *p*: *poly p* ≠ *poly* []
from *p* **have** *ap*: *poly* ([*a*,*1*] *** *p*) ≠ *poly* []
using *poly-mult-not-eq-poly-Nil* **unfolding** *poly-entire* **by** *auto*
have *eq*: *poly* ([*a*,*1*] %^(*Suc n*) *** *p*) = *poly* ([*a*,*1*]%^*n* *** ([*a*,*1*] *** *p*))
by (*rule ext*, *simp add*: *poly-mult poly-add poly-exp poly-cmult mult-ac add-ac right-distrib*)
from *ap* **have** *ap'*: (*poly* ([*a*,*1*] *** *p*) = *poly* []) = *False* **by** *blast*
have *th0*: *degree* ([*a*,*1*]%^*n* *** ([*a*,*1*] *** *p*)) = *degree* ([*a*,*1*] *** *p*) + *n*
apply (*simp only*: *Suc.hyps*[*of a pmult* [*a*,*one*] *p*] *ap'*)
by *simp*

from *degree-unique*[*OF eq*] *ap p th0 linear-mul-degree*[*OF p*, *of a*]
have *?case* **by** (*auto simp del*: *poly.simps*)**}**
ultimately show *?case* **by** *blast*
qed

lemma (**in** *recpower-idom-char-0*) *order-degree*:
assumes *p0*: *poly p* ≠ *poly* []
shows *order a p* ≤ *degree p*
proof−
from *order2*[*OF p0*, *unfolded divides-def*]
obtain *q* **where** *q*: *poly p* = *poly* ([− *a*, *1*]%^ (*order a p*) *** *q*) **by** *blast*
{assume *poly q* = *poly* []
with *q p0* **have** *False* **by** (*simp add*: *poly-mult poly-entire*)**}**
with *degree-unique*[*OF q*, *unfolded linear-pow-mul-degree*]
show *?thesis* **by** *auto*
qed

Tidier versions of finiteness of roots.

lemma (**in** *idom-char-0*) *poly-roots-finite-set*: *poly p* ≠ *poly* [] ==> *finite* {*x*. *poly p x* = *0*}
unfolding *poly-roots-finite* **.**

bound for polynomial.

lemma *poly-mono*: *abs*(*x*) ≤ *k* ==> *abs*(*poly p* (*x*::'*a*::{*ordered-idom*})) ≤ *poly* (*map abs p*) *k*

apply (*induct p*, *auto*)
apply (*rule-tac y* = *abs a* + *abs* (*x* ∗ *poly p x*) **in** *order-trans*)
apply (*rule abs-triangle-ineq*)
apply (*auto intro*!: *mult-mono simp add*: *abs-mult*)
done

lemma (**in** *semiring-0*) *poly-Sing*: *poly* [*c*] *x* = *c* **by** *simp*

end

53 While-Combinator: A general "while" combinator

theory *While-Combinator*
imports *Main*
begin

We define the while combinator as the "mother of all tail recursive functions".

function (*tailrec*) *while* :: ('*a* ⇒ *bool*) ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a*
where
while-unfold[*simp del*]: *while b c s* = (*if b s then while b c* (*c s*) *else s*)
by *auto*

declare *while-unfold*[*code*]

lemma *def-while-unfold*:
assumes *fdef*: *f* == *while test do*
shows *f x* = (*if test x then f*(*do x*) *else x*)
proof −
have *f x* = *while test do x* **using** *fdef* **by** *simp*
also have ... = (*if test x then while test do* (*do x*) *else x*)
by(*rule while-unfold*)
also have ... = (*if test x then f*(*do x*) *else x*) **by**(*simp add*:*fdef*[*symmetric*])
finally show *?thesis* **.**
qed

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

theorem *while-rule-lemma*:
assumes *invariant*: !!*s*. *P s* ==> *b s* ==> *P* (*c s*)
and *terminate*: !!*s*. *P s* ==> ¬ *b s* ==> *Q s*
and *wf*: *wf* {(*t*, *s*). *P s* ∧ *b s* ∧ *t* = *c s*}
shows *P s* ⟹ *Q* (*while b c s*)
using *wf*
apply (*induct s*)
apply *simp*
apply (*subst while-unfold*)

apply (*simp add*: *invariant terminate*)
done

theorem *while-rule*:
[| *P s*;
!!*s*. [| *P s*; *b s* |] ==> *P* (*c s*);
!!*s*. [| *P s*; ¬ *b s* |] ==> *Q s*;
wf r;
!!*s*. [| *P s*; *b s* |] ==> (*c s*, *s*) ∈ *r* |] ==>
Q (*while b c s*)
apply (*rule while-rule-lemma*)
prefer *4* **apply** *assumption*
apply *blast*
apply *blast*
apply (*erule wf-subset*)
apply *blast*
done

An application: computation of the *lfp* on finite sets via iteration.

theorem *lfp-conv-while*:
[| *mono f*; *finite U*; *f U* = *U* |] ==>
lfp f = *fst* (*while* (λ(*A*, *fA*). *A* ≠ *fA*) (λ(*A*, *fA*). (*fA*, *f fA*)) ({}, *f* {}))
apply (*rule-tac P* = λ(*A*, *B*). (*A* ⊆ *U* ∧ *B* = *f A* ∧ *A* ⊆ *B* ∧ *B* ⊆ *lfp f*) **and**
r = ((*Pow U* × *UNIV*) × (*Pow U* × *UNIV*)) ∩
inv-image finite-psubset (*op* − *U o fst*) **in** *while-rule*)
apply (*subst lfp-unfold*)
apply *assumption*
apply (*simp add*: *monoD*)
apply (*subst lfp-unfold*)
apply *assumption*
apply *clarsimp*
apply (*blast dest*: *monoD*)
apply (*fastsimp intro*!: *lfp-lowerbound*)
apply (*blast intro*: *wf-finite-psubset Int-lower2* [*THEN* [*2*] *wf-subset*])
apply (*clarsimp simp add*: *finite-psubset-def order-less-le*)
apply (*blast intro*!: *finite-Diff dest*: *monoD*)
done

An example of using the *while* combinator.

Cannot use *set-eq-subset* because it leads to looping because the anti-symmetry simproc turns the subset relationship back into equality.

theorem *P* (*lfp* (λ*N*::*int set*. {*0*} ∪ {(*n* + *2*) *mod 6* | *n*. *n* ∈ *N*})) =
P {*0*, *4*, *2*}
proof −
have *seteq*: !!*A B*. (*A* = *B*) = ((!*a* : *A*. *a*:*B*) & (!*b*:*B*. *b*:*A*))
by *blast*
have *aux*: !!*f A B*. {*f n* | *n*. *A n* ∨ *B n*} = {*f n* | *n*. *A n*} ∪ {*f n* | *n*. *B n*}
apply *blast*

done
show *?thesis*
apply (*subst lfp-conv-while* [**where** *?U* = {*0*, *1*, *2*, *3*, *4*, *5*}])
apply (*rule monoI*)
apply *blast*
apply *simp*
apply (*simp add*: *aux set-eq-subset*)

The fixpoint computation is performed purely by rewriting:

apply (*simp add*: *while-unfold aux seteq del*: *subset-empty*)
done
qed

end

54 Word: Binary Words

theory *Word*
imports *List*
begin

54.1 Auxilary Lemmas

lemma *max-le* [*intro*!]: [| $x \le z$; $y \le z$ |] ==> *max x y* $\le z$
by (*simp add*: *max-def*)

lemma *max-mono*:
fixes *x* :: *'a*::*linorder*
assumes *mf*: *mono f*
shows *max* (*f x*) (*f y*) $\le$ *f* (*max x y*)
proof −
from *mf* **and** *le-maxI1* [*of x y*]
have *fx*: *f x* $\le$ *f* (*max x y*) **by** (*rule monoD*)
from *mf* **and** *le-maxI2* [*of y x*]
have *fy*: *f y* $\le$ *f* (*max x y*) **by** (*rule monoD*)
from *fx* **and** *fy*
show *max* (*f x*) (*f y*) $\le$ *f* (*max x y*) **by** *auto*
qed

declare *zero-le-power* [*intro*]
and *zero-less-power* [*intro*]

lemma *int-nat-two-exp*: *2* ^ *k* = *int* (*2* ^ *k*)
by (*simp add*: *zpower-int* [*symmetric*])

54.2 Bits

datatype *bit* =

Zero (**0**)
| *One* (**1**)

primrec
bitval :: *bit* => *nat*
where
bitval **0** = *0*
| *bitval* **1** = *1*

consts
bitnot :: *bit* => *bit*
bitand :: *bit* => *bit* => *bit* (**infixr** *bitand 35*)
bitor :: *bit* => *bit* => *bit* (**infixr** *bitor 30*)
bitxor :: *bit* => *bit* => *bit* (**infixr** *bitxor 30*)

notation (*xsymbols*)
bitnot ($\neg_b$ - [*40*] *40*) **and**
bitand (**infixr** $\wedge_b$ *35*) **and**
bitor (**infixr** $\vee_b$ *30*) **and**
bitxor (**infixr** $\oplus_b$ *30*)

notation (*HTML* **output**)
bitnot ($\neg_b$ - [*40*] *40*) **and**
bitand (**infixr** $\wedge_b$ *35*) **and**
bitor (**infixr** $\vee_b$ *30*) **and**
bitxor (**infixr** $\oplus_b$ *30*)

primrec
bitnot-zero: (*bitnot* **0**) = **1**
bitnot-one : (*bitnot* **1**) = **0**

primrec
bitand-zero: (**0** *bitand y*) = **0**
bitand-one: (**1** *bitand y*) = *y*

primrec
bitor-zero: (**0** *bitor y*) = *y*
bitor-one: (**1** *bitor y*) = **1**

primrec
bitxor-zero: (**0** *bitxor y*) = *y*
bitxor-one: (**1** *bitxor y*) = (*bitnot y*)

lemma *bitnot-bitnot* [*simp*]: (*bitnot* (*bitnot b*)) = *b*
by (*cases b*) *simp-all*

lemma *bitand-cancel* [*simp*]: (*b bitand b*) = *b*
by (*cases b*) *simp-all*

lemma *bitor-cancel* [*simp*]: (*b bitor b*) = *b*
by (*cases b*) *simp-all*

lemma *bitxor-cancel* [*simp*]: (*b bitxor b*) = **0**
by (*cases b*) *simp-all*

54.3 Bit Vectors

First, a couple of theorems expressing case analysis and induction principles for bit vectors.

lemma *bit-list-cases*:
assumes *empty*: *w* = [] ==> *P w*
and *zero*: !!*bs*. *w* = **0** # *bs* ==> *P w*
and *one*: !!*bs*. *w* = **1** # *bs* ==> *P w*
shows *P w*
proof (*cases w*)
assume *w* = []
thus *?thesis* **by** (*rule empty*)
next
fix *b bs*
assume [*simp*]: *w* = *b* # *bs*
show *P w*
proof (*cases b*)
assume *b* = **0**
hence *w* = **0** # *bs* **by** *simp*
thus *?thesis* **by** (*rule zero*)
next
assume *b* = **1**
hence *w* = **1** # *bs* **by** *simp*
thus *?thesis* **by** (*rule one*)
qed
qed

lemma *bit-list-induct*:
assumes *empty*: *P* []
and *zero*: !!*bs*. *P bs* ==> *P* (**0**#*bs*)
and *one*: !!*bs*. *P bs* ==> *P* (**1**#*bs*)
shows *P w*
proof (*induct w*, *simp-all add*: *empty*)
fix *b bs*
assume *P bs*
then show *P* (*b*#*bs*)
by (*cases b*) (*auto intro*!: *zero one*)
qed

definition
bv-msb :: *bit list* => *bit* **where**
bv-msb w = (*if w* = [] *then* **0** *else hd w*)

definition
bv-extend :: [*nat*,*bit*,*bit list*]=>*bit list* **where**
bv-extend i b w = (*replicate* (*i* − *length w*) *b*) @ *w*

definition
bv-not :: *bit list* => *bit list* **where**
bv-not w = *map bitnot w*

lemma *bv-length-extend* [*simp*]: *length w* ≤ *i* ==> *length* (*bv-extend i b w*) = *i*
by (*simp add*: *bv-extend-def*)

lemma *bv-not-Nil* [*simp*]: *bv-not* [] = []
by (*simp add*: *bv-not-def*)

lemma *bv-not-Cons* [*simp*]: *bv-not* (*b*#*bs*) = (*bitnot b*) # *bv-not bs*
by (*simp add*: *bv-not-def*)

lemma *bv-not-bv-not* [*simp*]: *bv-not* (*bv-not w*) = *w*
by (*rule bit-list-induct* [*of* - *w*]) *simp-all*

lemma *bv-msb-Nil* [*simp*]: *bv-msb* [] = **0**
by (*simp add*: *bv-msb-def*)

lemma *bv-msb-Cons* [*simp*]: *bv-msb* (*b*#*bs*) = *b*
by (*simp add*: *bv-msb-def*)

lemma *bv-msb-bv-not* [*simp*]: *0* < *length w* ==> *bv-msb* (*bv-not w*) = (*bitnot* (*bv-msb w*))
by (*cases w*) *simp-all*

lemma *bv-msb-one-length* [*simp*,*intro*]: *bv-msb w* = **1** ==> *0* < *length w*
by (*cases w*) *simp-all*

lemma *length-bv-not* [*simp*]: *length* (*bv-not w*) = *length w*
by (*induct w*) *simp-all*

definition
bv-to-nat :: *bit list* => *nat* **where**
bv-to-nat = *foldl* (%*bn b*. *2* ∗ *bn* + *bitval b*) *0*

lemma *bv-to-nat-Nil* [*simp*]: *bv-to-nat* [] = *0*
by (*simp add*: *bv-to-nat-def*)

lemma *bv-to-nat-helper* [*simp*]: *bv-to-nat* (*b* # *bs*) = *bitval b* ∗ *2* ^ *length bs* + *bv-to-nat bs*
proof −
let *?bv-to-nat'* = *foldl* (λ*bn b*. *2* ∗ *bn* + *bitval b*)
have *helper*: ⋀*base*. *?bv-to-nat' base bs* = *base* ∗ *2* ^ *length bs* + *?bv-to-nat' 0 bs*

```
  proof (induct bs)
    case Nil
    show ?case by simp
  next
    case (Cons x xs base)
    show ?case
      apply (simp only: foldl.simps)
      apply (subst Cons [of 2 * base + bitval x])
      apply simp
      apply (subst Cons [of bitval x])
      apply (simp add: add-mult-distrib)
      done
  qed
  show ?thesis by (simp add: bv-to-nat-def) (rule helper)
qed

lemma bv-to-nat0 [simp]: bv-to-nat (0#bs) = bv-to-nat bs
  by simp

lemma bv-to-nat1 [simp]: bv-to-nat (1#bs) = 2 ^ length bs + bv-to-nat bs
  by simp

lemma bv-to-nat-upper-range: bv-to-nat w < 2 ^ length w
proof (induct w, simp-all)
  fix b bs
  assume bv-to-nat bs < 2 ^ length bs
  show bitval b * 2 ^ length bs + bv-to-nat bs < 2 * 2 ^ length bs
  proof (cases b, simp-all)
    have bv-to-nat bs < 2 ^ length bs by fact
    also have ... < 2 * 2 ^ length bs by auto
    finally show bv-to-nat bs < 2 * 2 ^ length bs by simp
  next
    have bv-to-nat bs < 2 ^ length bs by fact
    hence 2 ^ length bs + bv-to-nat bs < 2 ^ length bs + 2 ^ length bs by arith
    also have ... = 2 * (2 ^ length bs) by simp
    finally show bv-to-nat bs < 2 ^ length bs by simp
  qed
qed

lemma bv-extend-longer [simp]:
  assumes wn: n ≤ length w
  shows       bv-extend n b w = w
  by (simp add: bv-extend-def wn)

lemma bv-extend-shorter [simp]:
  assumes wn: length w < n
  shows       bv-extend n b w = bv-extend n b (b#w)
proof −
  from wn
```

```
  have s: n − Suc (length w) + 1 = n − length w
    by arith
  have bv-extend n b w = replicate (n − length w) b @ w
    by (simp add: bv-extend-def)
  also have ... = replicate (n − Suc (length w) + 1) b @ w
    by (subst s) rule
  also have ... = (replicate (n − Suc (length w)) b @ replicate 1 b) @ w
    by (subst replicate-add) rule
  also have ... = replicate (n − Suc (length w)) b @ b # w
    by simp
  also have ... = bv-extend n b (b#w)
    by (simp add: bv-extend-def)
  finally show bv-extend n b w = bv-extend n b (b#w) .
qed

consts
  rem-initial :: bit => bit list => bit list
primrec
  rem-initial b [] = []
  rem-initial b (x#xs) = (if b = x then rem-initial b xs else x#xs)

lemma rem-initial-length: length (rem-initial b w) ≤ length w
  by (rule bit-list-induct [of - w],simp-all (no-asm),safe,simp-all)

lemma rem-initial-equal:
  assumes p: length (rem-initial b w) = length w
  shows      rem-initial b w = w
proof −
  have length (rem-initial b w) = length w −−> rem-initial b w = w
  proof (induct w, simp-all, clarify)
    fix xs
    assume length (rem-initial b xs) = length xs −−> rem-initial b xs = xs
    assume f: length (rem-initial b xs) = Suc (length xs)
    with rem-initial-length [of b xs]
    show rem-initial b xs = b#xs
      by auto
  qed
  from this and p show ?thesis ..
qed

lemma bv-extend-rem-initial: bv-extend (length w) b (rem-initial b w) = w
proof (induct w, simp-all, safe)
  fix xs
  assume ind: bv-extend (length xs) b (rem-initial b xs) = xs
  from rem-initial-length [of b xs]
  have [simp]: Suc (length xs) − length (rem-initial b xs) =
      1 + (length xs − length (rem-initial b xs))
    by arith
  have bv-extend (Suc (length xs)) b (rem-initial b xs) =
```

```
    replicate (Suc (length xs) − length (rem-initial b xs)) b @ (rem-initial b xs)
   by (simp add: bv-extend-def)
  also have ... =
    replicate (1 + (length xs − length (rem-initial b xs))) b @ rem-initial b xs
   by simp
  also have ... =
   (replicate 1 b @ replicate (length xs − length (rem-initial b xs)) b) @ rem-initial
b xs
   by (subst replicate-add) (rule refl)
  also have ... = b # bv-extend (length xs) b (rem-initial b xs)
   by (auto simp add: bv-extend-def [symmetric])
  also have ... = b # xs
   by (simp add: ind)
  finally show bv-extend (Suc (length xs)) b (rem-initial b xs) = b # xs .
qed

lemma rem-initial-append1:
  assumes rem-initial b xs ~= []
  shows   rem-initial b (xs @ ys) = rem-initial b xs @ ys
  using assms by (induct xs) auto

lemma rem-initial-append2:
  assumes rem-initial b xs = []
  shows   rem-initial b (xs @ ys) = rem-initial b ys
  using assms by (induct xs) auto

definition
  norm-unsigned :: bit list => bit list where
  norm-unsigned = rem-initial 0

lemma norm-unsigned-Nil [simp]: norm-unsigned [] = []
  by (simp add: norm-unsigned-def)

lemma norm-unsigned-Cons0 [simp]: norm-unsigned (0#bs) = norm-unsigned bs
  by (simp add: norm-unsigned-def)

lemma norm-unsigned-Cons1 [simp]: norm-unsigned (1#bs) = 1#bs
  by (simp add: norm-unsigned-def)

lemma norm-unsigned-idem [simp]: norm-unsigned (norm-unsigned w) = norm-unsigned
w
  by (rule bit-list-induct [of - w],simp-all)

consts
  nat-to-bv-helper :: nat => bit list => bit list
recdef nat-to-bv-helper measure (λn. n)
  nat-to-bv-helper n = (%bs. (if n = 0 then bs
                              else nat-to-bv-helper (n div 2) ((if n mod 2 = 0 then 0
else 1)#bs)))
```

definition
nat-to-bv :: *nat* => *bit list* **where**
nat-to-bv n = *nat-to-bv-helper n* []

lemma *nat-to-bv0* [*simp*]: *nat-to-bv 0* = []
by (*simp add*: *nat-to-bv-def*)

lemmas [*simp del*] = *nat-to-bv-helper.simps*

lemma *n-div-2-cases*:
assumes *zero*: (*n*::*nat*) = *0* ==> *R*
and *div* : [| *n div 2* < *n* ; *0* < *n* |] ==> *R*
shows *R*
proof (*cases n* = *0*)
assume *n* = *0*
thus *R* **by** (*rule zero*)
next
assume *n* ~= *0*
hence *0* < *n* **by** *simp*
hence *n div 2* < *n* **by** *arith*
from *this* **and** ‹*0* < *n*› **show** *R* **by** (*rule div*)
qed

lemma *int-wf-ge-induct*:
assumes *ind* : !!*i*::*int*. (!!*j*. [| *k* ≤ *j* ; *j* < *i* |] ==> *P j*) ==> *P i*
shows *P i*
proof (*rule wf-induct-rule* [*OF wf-int-ge-less-than*])
fix *x*
assume *ih*: (⋀*y*::*int*. (*y*, *x*) ∈ *int-ge-less-than k* ⟹ *P y*)
thus *P x*
by (*rule ind*) (*simp add*: *int-ge-less-than-def*)
qed

lemma *unfold-nat-to-bv-helper*:
nat-to-bv-helper b l = *nat-to-bv-helper b* [] @ *l*
proof −
have ∀ *l*. *nat-to-bv-helper b l* = *nat-to-bv-helper b* [] @ *l*
proof (*induct b rule*: *less-induct*)
fix *n*
assume *ind*: !!*j*. *j* < *n* ⟹ ∀ *l*. *nat-to-bv-helper j l* = *nat-to-bv-helper j* [] @ *l*
show ∀ *l*. *nat-to-bv-helper n l* = *nat-to-bv-helper n* [] @ *l*
proof
fix *l*
show *nat-to-bv-helper n l* = *nat-to-bv-helper n* [] @ *l*
proof (*cases n* < *0*)
assume *n* < *0*
thus *?thesis*
by (*simp add*: *nat-to-bv-helper.simps*)

```
    next
      assume ~n < 0
      show ?thesis
      proof (rule n-div-2-cases [of n])
        assume [simp]: n = 0
        show ?thesis
          apply (simp only: nat-to-bv-helper.simps [of n])
          apply simp
          done
      next
        assume n2n: n div 2 < n
        assume [simp]: 0 < n
        hence n20: 0 ≤ n div 2
          by arith
        from ind [of n div 2] and n2n n20
        have ind': ∀ l. nat-to-bv-helper (n div 2) l = nat-to-bv-helper (n div 2) []
@ l
          by blast
        show ?thesis
          apply (simp only: nat-to-bv-helper.simps [of n])
          apply (cases n=0)
          apply simp
          apply (simp only: if-False)
          apply simp
          apply (subst spec [OF ind',of 0#l])
          apply (subst spec [OF ind',of 1#l])
          apply (subst spec [OF ind',of [1]])
          apply (subst spec [OF ind',of [0]])
          apply simp
          done
      qed
    qed
  qed
  thus ?thesis ..
qed

lemma nat-to-bv-non0 [simp]: n≠0 ==> nat-to-bv n = nat-to-bv (n div 2) @ [if
n mod 2 = 0 then 0 else 1]
proof -
  assume [simp]: n≠0
  show ?thesis
    apply (subst nat-to-bv-def [of n])
    apply (simp only: nat-to-bv-helper.simps [of n])
    apply (subst unfold-nat-to-bv-helper)
    using prems
    apply (simp)
    apply (subst nat-to-bv-def [of n div 2])
    apply auto
```

```
    done
qed

lemma bv-to-nat-dist-append:
  bv-to-nat (l1 @ l2) = bv-to-nat l1 * 2 ^ length l2 + bv-to-nat l2
proof −
  have ∀ l2. bv-to-nat (l1 @ l2) = bv-to-nat l1 * 2 ^ length l2 + bv-to-nat l2
  proof (induct l1,simp-all)
    fix x xs
   assume ind: ∀ l2. bv-to-nat (xs @ l2) = bv-to-nat xs * 2 ^ length l2 + bv-to-nat
l2
    show ∀ l2. bv-to-nat xs * 2 ^ length l2 + bitval x * 2 ^ (length xs + length l2)
= (bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
    proof
      fix l2
      show bv-to-nat xs * 2 ^ length l2 + bitval x * 2 ^ (length xs + length l2) =
(bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
      proof −
        have (2::nat) ^ (length xs + length l2) = 2 ^ length xs * 2 ^ length l2
          by (induct length xs,simp-all)
       hence bitval x * 2 ^ (length xs + length l2) + bv-to-nat xs * 2 ^ length l2 =
          bitval x * 2 ^ length xs * 2 ^ length l2 + bv-to-nat xs * 2 ^ length l2
          by simp
        also have ... = (bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
          by (simp add: ring-distribs)
        finally show ?thesis by simp
      qed
    qed
  qed
  thus ?thesis ..
qed

lemma bv-nat-bv [simp]: bv-to-nat (nat-to-bv n) = n
proof (induct n rule: less-induct)
  fix n
  assume ind: !!j. j < n ⟹ bv-to-nat (nat-to-bv j) = j
  show bv-to-nat (nat-to-bv n) = n
  proof (rule n-div-2-cases [of n])
    assume n = 0 then show ?thesis by simp
  next
    assume nn: n div 2 < n
    assume n0: 0 < n
    from ind and nn
    have ind': bv-to-nat (nat-to-bv (n div 2)) = n div 2 by blast
    from n0 have n0': n ≠ 0 by simp
    show ?thesis
      apply (subst nat-to-bv-def)
      apply (simp only: nat-to-bv-helper.simps [of n])
      apply (simp only: n0' if-False)
```

```
    apply (subst unfold-nat-to-bv-helper)
    apply (subst bv-to-nat-dist-append)
    apply (fold nat-to-bv-def)
    apply (simp add: ind' split del: split-if)
    apply (cases n mod 2 = 0)
    proof (simp-all)
      assume n mod 2 = 0
      with mod-div-equality [of n 2]
      show n div 2 * 2 = n by simp
    next
      assume n mod 2 = Suc 0
      with mod-div-equality [of n 2]
      show Suc (n div 2 * 2) = n by arith
    qed
  qed
qed

lemma bv-to-nat-type [simp]: bv-to-nat (norm-unsigned w) = bv-to-nat w
  by (rule bit-list-induct) simp-all

lemma length-norm-unsigned-le [simp]: length (norm-unsigned w) ≤ length w
  by (rule bit-list-induct) simp-all

lemma bv-to-nat-rew-msb: bv-msb w = 1 ==> bv-to-nat w = 2 ^ (length w - 1)
+ bv-to-nat (tl w)
  by (rule bit-list-cases [of w]) simp-all

lemma norm-unsigned-result: norm-unsigned xs = [] ∨ bv-msb (norm-unsigned xs)
= 1
proof (rule length-induct [of - xs])
  fix xs :: bit list
  assume ind: ∀ ys. length ys < length xs --> norm-unsigned ys = [] ∨ bv-msb
(norm-unsigned ys) = 1
  show norm-unsigned xs = [] ∨ bv-msb (norm-unsigned xs) = 1
  proof (rule bit-list-cases [of xs],simp-all)
    fix bs
    assume [simp]: xs = 0#bs
    from ind
   have length bs < length xs --> norm-unsigned bs = [] ∨ bv-msb (norm-unsigned
bs) = 1 ..
    thus norm-unsigned bs = [] ∨ bv-msb (norm-unsigned bs) = 1 by simp
  qed
qed

lemma norm-empty-bv-to-nat-zero:
  assumes nw: norm-unsigned w = []
  shows       bv-to-nat w = 0
proof -
  have bv-to-nat w = bv-to-nat (norm-unsigned w) by simp
```

also have ... = *bv-to-nat* [] **by** (*subst nw*) (*rule refl*)
also have ... = *0* **by** *simp*
finally show *?thesis* **.**
qed

lemma *bv-to-nat-lower-limit*:
assumes *w0*: *0* < *bv-to-nat w*
shows *2* ^ (*length* (*norm-unsigned w*) − *1*) ≤ *bv-to-nat w*
proof −
from *w0* **and** *norm-unsigned-result* [*of w*]
have *msbw*: *bv-msb* (*norm-unsigned w*) = **1**
by (*auto simp add*: *norm-empty-bv-to-nat-zero*)
have *2* ^ (*length* (*norm-unsigned w*) − *1*) ≤ *bv-to-nat* (*norm-unsigned w*)
by (*subst bv-to-nat-rew-msb* [*OF msbw*],*simp*)
thus *?thesis* **by** *simp*
qed

lemmas [*simp del*] = *nat-to-bv-non0*

lemma *norm-unsigned-length* [*intro*!]: *length* (*norm-unsigned w*) ≤ *length w*
by (*subst norm-unsigned-def*,*rule rem-initial-length*)

lemma *norm-unsigned-equal*:
length (*norm-unsigned w*) = *length w* ==> *norm-unsigned w* = *w*
by (*simp add*: *norm-unsigned-def*,*rule rem-initial-equal*)

lemma *bv-extend-norm-unsigned*: *bv-extend* (*length w*) **0** (*norm-unsigned w*) = *w*
by (*simp add*: *norm-unsigned-def*,*rule bv-extend-rem-initial*)

lemma *norm-unsigned-append1* [*simp*]:
norm-unsigned xs ≠ [] ==> *norm-unsigned* (*xs* @ *ys*) = *norm-unsigned xs* @ *ys*
by (*simp add*: *norm-unsigned-def*,*rule rem-initial-append1*)

lemma *norm-unsigned-append2* [*simp*]:
norm-unsigned xs = [] ==> *norm-unsigned* (*xs* @ *ys*) = *norm-unsigned ys*
by (*simp add*: *norm-unsigned-def*,*rule rem-initial-append2*)

lemma *bv-to-nat-zero-imp-empty*:
bv-to-nat w = *0* ⟹ *norm-unsigned w* = []
by (*atomize* (*full*), *induct w rule*: *bit-list-induct*) *simp-all*

lemma *bv-to-nat-nzero-imp-nempty*:
bv-to-nat w ≠ *0* ⟹ *norm-unsigned w* ≠ []
by (*induct w rule*: *bit-list-induct*) *simp-all*

lemma *nat-helper1*:
assumes *ass*: *nat-to-bv* (*bv-to-nat w*) = *norm-unsigned w*
shows *nat-to-bv* (*2* ∗ *bv-to-nat w* + *bitval x*) = *norm-unsigned* (*w* @ [*x*])
proof (*cases x*)

```
  assume [simp]: x = 1
  show ?thesis
    apply (simp add: nat-to-bv-non0)
    apply safe
  proof −
    fix q
    assume Suc (2 * bv-to-nat w) = 2 * q
    hence orig: (2 * bv-to-nat w + 1) mod 2 = 2 * q mod 2 (is ?lhs = ?rhs)
      by simp
    have ?lhs = (1 + 2 * bv-to-nat w) mod 2
      by (simp add: add-commute)
    also have ... = 1
      by (subst mod-add1-eq) simp
    finally have eq1: ?lhs = 1 .
    have ?rhs  = 0 by simp
    with orig and eq1
   show nat-to-bv (Suc (2 * bv-to-nat w) div 2) @ [0] = norm-unsigned (w @ [1])
      by simp
  next
    have nat-to-bv (Suc (2 * bv-to-nat w) div 2) @ [1] =
        nat-to-bv ((1 + 2 * bv-to-nat w) div 2) @ [1]
      by (simp add: add-commute)
    also have ... = nat-to-bv (bv-to-nat w) @ [1]
      by (subst div-add1-eq) simp
    also have ... = norm-unsigned w @ [1]
      by (subst ass) (rule refl)
    also have ... = norm-unsigned (w @ [1])
      by (cases norm-unsigned w) simp-all
    finally show nat-to-bv (Suc (2 * bv-to-nat w) div 2) @ [1] = norm-unsigned
(w @ [1]) .
  qed
next
  assume [simp]: x = 0
  show ?thesis
  proof (cases bv-to-nat w = 0)
    assume bv-to-nat w = 0
    thus ?thesis
      by (simp add: bv-to-nat-zero-imp-empty)
  next
    assume bv-to-nat w ≠ 0
    thus ?thesis
      apply simp
      apply (subst nat-to-bv-non0)
      apply simp
      apply auto
      apply (subst ass)
      apply (cases norm-unsigned w)
      apply (simp-all add: norm-empty-bv-to-nat-zero)
      done
```

```
  qed
qed

lemma nat-helper2: nat-to-bv (2 ^ length xs + bv-to-nat xs) = 1 # xs
proof −
  have ∀ xs. nat-to-bv (2 ^ length (rev xs) + bv-to-nat (rev xs)) = 1 # (rev xs)
(is ∀ xs. ?P xs)
  proof
    fix xs
    show ?P xs
    proof (rule length-induct [of - xs])
      fix xs :: bit list
      assume ind: ∀ ys. length ys < length xs −−> ?P ys
      show ?P xs
      proof (cases xs)
        assume xs = []
        then show ?thesis by (simp add: nat-to-bv-non0)
      next
        fix y ys
        assume [simp]: xs = y # ys
        show ?thesis
          apply simp
          apply (subst bv-to-nat-dist-append)
          apply simp
        proof −
          have nat-to-bv (2 * 2 ^ length ys + (bv-to-nat (rev ys) * 2 + bitval y)) =
            nat-to-bv (2 * (2 ^ length ys + bv-to-nat (rev ys)) + bitval y)
            by (simp add: add-ac mult-ac)
          also have ... = nat-to-bv (2 * (bv-to-nat (1#rev ys)) + bitval y)
            by simp
          also have ... = norm-unsigned (1#rev ys) @ [y]
          proof −
            from ind
            have nat-to-bv (2 ^ length (rev ys) + bv-to-nat (rev ys)) = 1 # rev ys
              by auto
           hence [simp]: nat-to-bv (2 ^ length ys + bv-to-nat (rev ys)) = 1 # rev ys
              by simp
            show ?thesis
              apply (subst nat-helper1)
              apply simp-all
              done
          qed
          also have ... = (1#rev ys) @ [y]
            by simp
          also have ... = 1 # rev ys @ [y]
            by simp
           finally show nat-to-bv (2 * 2 ^ length ys + (bv-to-nat (rev ys) * 2 +
bitval y)) =
              1 # rev ys @ [y] .
```

```
      qed
    qed
  qed
 qed
 hence nat-to-bv (2 ^ length (rev (rev xs)) + bv-to-nat (rev (rev xs))) =
    1 # rev (rev xs) ..
 thus ?thesis by simp
qed

lemma nat-bv-nat [simp]: nat-to-bv (bv-to-nat w) = norm-unsigned w
proof (rule bit-list-induct [of - w],simp-all)
 fix xs
 assume nat-to-bv (bv-to-nat xs) = norm-unsigned xs
 have bv-to-nat xs = bv-to-nat (norm-unsigned xs) by simp
 have bv-to-nat xs < 2 ^ length xs
  by (rule bv-to-nat-upper-range)
 show nat-to-bv (2 ^ length xs + bv-to-nat xs) = 1 # xs
  by (rule nat-helper2)
qed

lemma bv-to-nat-qinj:
 assumes one: bv-to-nat xs = bv-to-nat ys
 and     len: length xs = length ys
 shows       xs = ys
proof -
 from one
 have nat-to-bv (bv-to-nat xs) = nat-to-bv (bv-to-nat ys)
  by simp
 hence xsys: norm-unsigned xs = norm-unsigned ys
  by simp
 have xs = bv-extend (length xs) 0 (norm-unsigned xs)
  by (simp add: bv-extend-norm-unsigned)
 also have ... = bv-extend (length ys) 0 (norm-unsigned ys)
  by (simp add: xsys len)
 also have ... = ys
  by (simp add: bv-extend-norm-unsigned)
 finally show ?thesis .
qed

lemma norm-unsigned-nat-to-bv [simp]:
 norm-unsigned (nat-to-bv n) = nat-to-bv n
proof -
 have norm-unsigned (nat-to-bv n) = nat-to-bv (bv-to-nat (norm-unsigned (nat-to-bv
n)))
  by (subst nat-bv-nat) simp
 also have ... = nat-to-bv n by simp
 finally show ?thesis .
qed
```

lemma *length-nat-to-bv-upper-limit*:
assumes *nk*: $n \leq 2 \hat{} k - 1$
shows $length\ (nat\text{-}to\text{-}bv\ n) \leq k$
proof (*cases* $n = 0$)
case *True*
thus *?thesis*
by (*simp add*: *nat-to-bv-def nat-to-bv-helper.simps*)
next
case *False*
hence *n0*: $0 < n$ **by** *simp*
show *?thesis*
proof (*rule ccontr*)
assume ~ $length\ (nat\text{-}to\text{-}bv\ n) \leq k$
hence $k < length\ (nat\text{-}to\text{-}bv\ n)$ **by** *simp*
hence $k \leq length\ (nat\text{-}to\text{-}bv\ n) - 1$ **by** *arith*
hence $(2::nat) \hat{} k \leq 2 \hat{} (length\ (nat\text{-}to\text{-}bv\ n) - 1)$ **by** *simp*
also have ... $= 2 \hat{} (length\ (norm\text{-}unsigned\ (nat\text{-}to\text{-}bv\ n)) - 1)$ **by** *simp*
also have ... $\leq bv\text{-}to\text{-}nat\ (nat\text{-}to\text{-}bv\ n)$
by (*rule bv-to-nat-lower-limit*) (*simp add*: *n0*)
also have ... $= n$ **by** *simp*
finally have $2 \hat{} k \leq n$ **.**
with *n0* **have** $2 \hat{} k - 1 < n$ **by** *arith*
with *nk* **show** *False* **by** *simp*
qed
qed

lemma *length-nat-to-bv-lower-limit*:
assumes *nk*: $2 \hat{} k \leq n$
shows $k < length\ (nat\text{-}to\text{-}bv\ n)$
proof (*rule ccontr*)
assume ~ $k < length\ (nat\text{-}to\text{-}bv\ n)$
hence *lnk*: $length\ (nat\text{-}to\text{-}bv\ n) \leq k$ **by** *simp*
have $n = bv\text{-}to\text{-}nat\ (nat\text{-}to\text{-}bv\ n)$ **by** *simp*
also have ... $< 2 \hat{} length\ (nat\text{-}to\text{-}bv\ n)$
by (*rule bv-to-nat-upper-range*)
also from *lnk* **have** ... $\leq 2 \hat{} k$ **by** *simp*
finally have $n < 2 \hat{} k$ **.**
with *nk* **show** *False* **by** *simp*
qed

54.4 Unsigned Arithmetic Operations

definition
bv-add :: [*bit list*, *bit list*] => *bit list* **where**
bv-add w1 w2 = *nat-to-bv* (*bv-to-nat w1* + *bv-to-nat w2*)

lemma *bv-add-type1* [*simp*]: *bv-add* (*norm-unsigned w1*) *w2* = *bv-add w1 w2*
by (*simp add*: *bv-add-def*)

lemma *bv-add-type2* [*simp*]: *bv-add w1* (*norm-unsigned w2*) = *bv-add w1 w2*
by (*simp add*: *bv-add-def*)

lemma *bv-add-returntype* [*simp*]: *norm-unsigned* (*bv-add w1 w2*) = *bv-add w1 w2*
by (*simp add*: *bv-add-def*)

lemma *bv-add-length*: *length* (*bv-add w1 w2*) ≤ *Suc* (*max* (*length w1*) (*length w2*))
proof (*unfold bv-add-def*,*rule length-nat-to-bv-upper-limit*)
from *bv-to-nat-upper-range* [*of w1*] **and** *bv-to-nat-upper-range* [*of w2*]
have *bv-to-nat w1* + *bv-to-nat w2* ≤ (2 ^ *length w1* − 1) + (2 ^ *length w2* − 1)
by *arith*
also have ... ≤
max (2 ^ *length w1* − 1) (2 ^ *length w2* − 1) + *max* (2 ^ *length w1* − 1) (2 ^ *length w2* − 1)
by (*rule add-mono*,*safe intro*!: *le-maxI1 le-maxI2*)
also have ... = 2 ∗ *max* (2 ^ *length w1* − 1) (2 ^ *length w2* − 1) **by** *simp*
also have ... ≤ 2 ^ *Suc* (*max* (*length w1*) (*length w2*)) − 2
proof (*cases length w1* ≤ *length w2*)
assume *w1w2*: *length w1* ≤ *length w2*
hence (2::*nat*) ^ *length w1* ≤ 2 ^ *length w2* **by** *simp*
hence (2::*nat*) ^ *length w1* − 1 ≤ 2 ^ *length w2* − 1 **by** *arith*
with *w1w2* **show** *?thesis*
by (*simp add*: *diff-mult-distrib2 split*: *split-max*)
next
assume [*simp*]: ~ (*length w1* ≤ *length w2*)
have ~ ((2::*nat*) ^ *length w1* − 1 ≤ 2 ^ *length w2* − 1)
proof
assume (2::*nat*) ^ *length w1* − 1 ≤ 2 ^ *length w2* − 1
hence ((2::*nat*) ^ *length w1* − 1) + 1 ≤ (2 ^ *length w2* − 1) + 1
by (*rule add-right-mono*)
hence (2::*nat*) ^ *length w1* ≤ 2 ^ *length w2* **by** *simp*
hence *length w1* ≤ *length w2* **by** *simp*
thus *False* **by** *simp*
qed
thus *?thesis*
by (*simp add*: *diff-mult-distrib2 split*: *split-max*)
qed
finally show *bv-to-nat w1* + *bv-to-nat w2* ≤ 2 ^ *Suc* (*max* (*length w1*) (*length w2*)) − 1
by *arith*
qed

definition
bv-mult :: [*bit list*, *bit list*] => *bit list* **where**
bv-mult w1 w2 = *nat-to-bv* (*bv-to-nat w1* ∗ *bv-to-nat w2*)

lemma *bv-mult-type1* [*simp*]: *bv-mult* (*norm-unsigned w1*) *w2* = *bv-mult w1 w2*
by (*simp add*: *bv-mult-def*)

lemma *bv-mult-type2* [*simp*]: *bv-mult w1* (*norm-unsigned w2*) = *bv-mult w1 w2*
by (*simp add*: *bv-mult-def*)

lemma *bv-mult-returntype* [*simp*]: *norm-unsigned* (*bv-mult w1 w2*) = *bv-mult w1 w2*
by (*simp add*: *bv-mult-def*)

lemma *bv-mult-length*: *length* (*bv-mult w1 w2*) ≤ *length w1* + *length w2*
proof (*unfold bv-mult-def*,*rule length-nat-to-bv-upper-limit*)
from *bv-to-nat-upper-range* [*of w1*] **and** *bv-to-nat-upper-range* [*of w2*]
have *h*: *bv-to-nat w1* ≤ 2 ^ *length w1* − 1 ∧ *bv-to-nat w2* ≤ 2 ^ *length w2* − 1
by *arith*
have *bv-to-nat w1* ∗ *bv-to-nat w2* ≤ (2 ^ *length w1* − 1) ∗ (2 ^ *length w2* − 1)
apply (*cut-tac h*)
apply (*rule mult-mono*)
apply *auto*
done
also have ... < 2 ^ *length w1* ∗ 2 ^ *length w2*
by (*rule mult-strict-mono*,*auto*)
also have ... = 2 ^ (*length w1* + *length w2*)
by (*simp add*: *power-add*)
finally show *bv-to-nat w1* ∗ *bv-to-nat w2* ≤ 2 ^ (*length w1* + *length w2*) − 1
by *arith*
qed

54.5 Signed Vectors

consts
norm-signed :: *bit list* => *bit list*
primrec
norm-signed-Nil: *norm-signed* [] = []
norm-signed-Cons: *norm-signed* (*b*#*bs*) =
(*case b of*
0 => *if norm-unsigned bs* = [] *then* [] *else b*#*norm-unsigned bs*
| **1** => *b*#*rem-initial b bs*)

lemma *norm-signed0* [*simp*]: *norm-signed* [**0**] = []
by *simp*

lemma *norm-signed1* [*simp*]: *norm-signed* [**1**] = [**1**]
by *simp*

lemma *norm-signed01* [*simp*]: *norm-signed* (**0**#**1**#*xs*) = **0**#**1**#*xs*
by *simp*

lemma *norm-signed00* [*simp*]: *norm-signed* (**0**#**0**#*xs*) = *norm-signed* (**0**#*xs*)
by *simp*

lemma *norm-signed10* [*simp*]: *norm-signed* (**1**#**0**#*xs*) = **1**#**0**#*xs*

by *simp*

lemma *norm-signed11* [*simp*]: *norm-signed* (**1**#**1**#*xs*) = *norm-signed* (**1**#*xs*)
by *simp*

lemmas [*simp del*] = *norm-signed-Cons*

definition
int-to-bv :: *int* => *bit list* **where**
int-to-bv n = (*if 0* ≤ *n*
then norm-signed (**0**#*nat-to-bv* (*nat n*))
else norm-signed (*bv-not* (**0**#*nat-to-bv* (*nat* (−*n*− *1*)))))

lemma *int-to-bv-ge0* [*simp*]: *0* ≤ *n* ==> *int-to-bv n* = *norm-signed* (**0** # *nat-to-bv* (*nat n*))
by (*simp add*: *int-to-bv-def*)

lemma *int-to-bv-lt0* [*simp*]:
n < *0* ==> *int-to-bv n* = *norm-signed* (*bv-not* (**0**#*nat-to-bv* (*nat* (−*n*− *1*))))
by (*simp add*: *int-to-bv-def*)

lemma *norm-signed-idem* [*simp*]: *norm-signed* (*norm-signed w*) = *norm-signed w*
proof (*rule bit-list-induct* [*of* - *w*], *simp-all*)
fix *xs*
assume *eq*: *norm-signed* (*norm-signed xs*) = *norm-signed xs*
show *norm-signed* (*norm-signed* (**0**#*xs*)) = *norm-signed* (**0**#*xs*)
proof (*rule bit-list-cases* [*of xs*],*simp-all*)
fix *ys*
assume *xs* = **0**#*ys*
from *this* [*symmetric*] **and** *eq*
show *norm-signed* (*norm-signed* (**0**#*ys*)) = *norm-signed* (**0**#*ys*)
by *simp*
qed
next
fix *xs*
assume *eq*: *norm-signed* (*norm-signed xs*) = *norm-signed xs*
show *norm-signed* (*norm-signed* (**1**#*xs*)) = *norm-signed* (**1**#*xs*)
proof (*rule bit-list-cases* [*of xs*],*simp-all*)
fix *ys*
assume *xs* = **1**#*ys*
from *this* [*symmetric*] **and** *eq*
show *norm-signed* (*norm-signed* (**1**#*ys*)) = *norm-signed* (**1**#*ys*)
by *simp*
qed
qed

definition
bv-to-int :: *bit list* => *int* **where**
bv-to-int w =

(*case bv-msb w of* **0** => *int* (*bv-to-nat w*)
| **1** => − *int* (*bv-to-nat* (*bv-not w*) + *1*))

lemma *bv-to-int-Nil* [*simp*]: *bv-to-int* [] = *0*
by (*simp add*: *bv-to-int-def*)

lemma *bv-to-int-Cons0* [*simp*]: *bv-to-int* (**0**#*bs*) = *int* (*bv-to-nat bs*)
by (*simp add*: *bv-to-int-def*)

lemma *bv-to-int-Cons1* [*simp*]: *bv-to-int* (**1**#*bs*) = − *int* (*bv-to-nat* (*bv-not bs*) + *1*)
by (*simp add*: *bv-to-int-def*)

lemma *bv-to-int-type* [*simp*]: *bv-to-int* (*norm-signed w*) = *bv-to-int w*
proof (*rule bit-list-induct* [*of* - *w*], *simp-all*)
fix *xs*
assume *ind*: *bv-to-int* (*norm-signed xs*) = *bv-to-int xs*
show *bv-to-int* (*norm-signed* (**0**#*xs*)) = *int* (*bv-to-nat xs*)
proof (*rule bit-list-cases* [*of xs*], *simp-all*)
fix *ys*
assume [*simp*]: *xs* = **0**#*ys*
from *ind*
show *bv-to-int* (*norm-signed* (**0**#*ys*)) = *int* (*bv-to-nat ys*)
by *simp*
qed
next
fix *xs*
assume *ind*: *bv-to-int* (*norm-signed xs*) = *bv-to-int xs*
show *bv-to-int* (*norm-signed* (**1**#*xs*)) = −*1* − *int* (*bv-to-nat* (*bv-not xs*))
proof (*rule bit-list-cases* [*of xs*], *simp-all*)
fix *ys*
assume [*simp*]: *xs* = **1**#*ys*
from *ind*
show *bv-to-int* (*norm-signed* (**1**#*ys*)) = −*1* − *int* (*bv-to-nat* (*bv-not ys*))
by *simp*
qed
qed

lemma *bv-to-int-upper-range*: *bv-to-int w* < *2* ^ (*length w* − *1*)
proof (*rule bit-list-cases* [*of w*],*simp-all*)
fix *bs*
from *bv-to-nat-upper-range*
show *int* (*bv-to-nat bs*) < *2* ^ *length bs*
by (*simp add*: *int-nat-two-exp*)
next
fix *bs*
have −*1* − *int* (*bv-to-nat* (*bv-not bs*)) ≤ *0* **by** *simp*
also have ... < *2* ^ *length bs* **by** (*induct bs*) *simp-all*
finally show −*1* − *int* (*bv-to-nat* (*bv-not bs*)) < *2* ^ *length bs* **.**

qed

lemma *bv-to-int-lower-range*: $-(2\ \hat{}\ (length\ w - 1)) \leq bv\text{-}to\text{-}int\ w$
proof (*rule bit-list-cases* [*of w*],*simp-all*)
  fix *bs* :: *bit list*
  have $-(2\ \hat{}\ length\ bs) \leq (0::int)$ **by** (*induct bs*) *simp-all*
  also have ... $\leq int\ (bv\text{-}to\text{-}nat\ bs)$ **by** *simp*
  finally show $-(2\ \hat{}\ length\ bs) \leq int\ (bv\text{-}to\text{-}nat\ bs)$ **.**
next
  fix *bs*
  from *bv-to-nat-upper-range* [*of bv-not bs*]
  show $-(2\ \hat{}\ length\ bs) \leq -1 - int\ (bv\text{-}to\text{-}nat\ (bv\text{-}not\ bs))$
    by (*simp add*: *int-nat-two-exp*)
qed

lemma *int-bv-int* [*simp*]: $int\text{-}to\text{-}bv\ (bv\text{-}to\text{-}int\ w) = norm\text{-}signed\ w$
proof (*rule bit-list-cases* [*of w*],*simp*)
  fix *xs*
  assume [*simp*]: $w = \mathbf{0}\#xs$
  show *?thesis*
    apply *simp*
    apply (*subst norm-signed-Cons* [*of* **0** *xs*])
    apply *simp*
    using *norm-unsigned-result* [*of xs*]
    apply *safe*
    apply (*rule bit-list-cases* [*of norm-unsigned xs*])
    apply *simp-all*
    done
next
  fix *xs*
  assume [*simp*]: $w = \mathbf{1}\#xs$
  show *?thesis*
    apply (*simp del*: *int-to-bv-lt0*)
    apply (*rule bit-list-induct* [*of* - *xs*])
    apply *simp*
    apply (*subst int-to-bv-lt0*)
    apply (*subgoal-tac* $-\ int\ (bv\text{-}to\text{-}nat\ (bv\text{-}not\ (\mathbf{0}\ \#\ bs))) + -1 < 0 + 0$)
    apply *simp*
    apply (*rule add-le-less-mono*)
    apply *simp*
    apply *simp*
    apply (*simp del*: *bv-to-nat1 bv-to-nat-helper*)
    apply *simp*
    done
qed

lemma *bv-int-bv* [*simp*]: $bv\text{-}to\text{-}int\ (int\text{-}to\text{-}bv\ i) = i$
  by (*cases* $0 \leq i$) *simp-all*

lemma *bv-msb-norm* [*simp*]: *bv-msb* (*norm-signed w*) = *bv-msb w*
by (*rule bit-list-cases* [*of w*]) (*simp-all add*: *norm-signed-Cons*)

lemma *norm-signed-length*: *length* (*norm-signed w*) ≤ *length w*
apply (*cases w*, *simp-all*)
apply (*subst norm-signed-Cons*)
apply (*case-tac a*, *simp-all*)
apply (*rule rem-initial-length*)
done

lemma *norm-signed-equal*: *length* (*norm-signed w*) = *length w* ==> *norm-signed w* = *w*
proof (*rule bit-list-cases* [*of w*], *simp-all*)
fix *xs*
assume *length* (*norm-signed* (**0**#*xs*)) = *Suc* (*length xs*)
thus *norm-signed* (**0**#*xs*) = **0**#*xs*
apply (*simp add*: *norm-signed-Cons*)
apply *safe*
apply *simp-all*
apply (*rule norm-unsigned-equal*)
apply *assumption*
done
next
fix *xs*
assume *length* (*norm-signed* (**1**#*xs*)) = *Suc* (*length xs*)
thus *norm-signed* (**1**#*xs*) = **1**#*xs*
apply (*simp add*: *norm-signed-Cons*)
apply (*rule rem-initial-equal*)
apply *assumption*
done
qed

lemma *bv-extend-norm-signed*: *bv-msb w* = *b* ==> *bv-extend* (*length w*) *b* (*norm-signed w*) = *w*
proof (*rule bit-list-cases* [*of w*],*simp-all*)
fix *xs*
show *bv-extend* (*Suc* (*length xs*)) **0** (*norm-signed* (**0**#*xs*)) = **0**#*xs*
proof (*simp add*: *norm-signed-list-def*,*auto*)
assume *norm-unsigned xs* = []
hence *xx*: *rem-initial* **0** *xs* = []
by (*simp add*: *norm-unsigned-def*)
have *bv-extend* (*Suc* (*length xs*)) **0** (**0**#*rem-initial* **0** *xs*) = **0**#*xs*
apply (*simp add*: *bv-extend-def replicate-app-Cons-same*)
apply (*fold bv-extend-def*)
apply (*rule bv-extend-rem-initial*)
done
thus *bv-extend* (*Suc* (*length xs*)) **0** [**0**] = **0**#*xs*
by (*simp add*: *xx*)
next

```
    show bv-extend (Suc (length xs)) 0 (0#norm-unsigned xs) = 0#xs
      apply (simp add: norm-unsigned-def)
      apply (simp add: bv-extend-def replicate-app-Cons-same)
      apply (fold bv-extend-def)
      apply (rule bv-extend-rem-initial)
      done
  qed
next
  fix xs
  show bv-extend (Suc (length xs)) 1 (norm-signed (1#xs)) = 1#xs
    apply (simp add: norm-signed-Cons)
    apply (simp add: bv-extend-def replicate-app-Cons-same)
    apply (fold bv-extend-def)
    apply (rule bv-extend-rem-initial)
    done
qed

lemma bv-to-int-qinj:
  assumes one: bv-to-int xs = bv-to-int ys
  and     len: length xs = length ys
  shows        xs = ys
proof −
  from one
  have int-to-bv (bv-to-int xs) = int-to-bv (bv-to-int ys) by simp
  hence xsys: norm-signed xs = norm-signed ys by simp
  hence xsys': bv-msb xs = bv-msb ys
  proof −
    have bv-msb xs = bv-msb (norm-signed xs) by simp
    also have ... = bv-msb (norm-signed ys) by (simp add: xsys)
    also have ... = bv-msb ys by simp
    finally show ?thesis .
  qed
  have xs = bv-extend (length xs) (bv-msb xs) (norm-signed xs)
    by (simp add: bv-extend-norm-signed)
  also have ... = bv-extend (length ys) (bv-msb ys) (norm-signed ys)
    by (simp add: xsys xsys' len)
  also have ... = ys
    by (simp add: bv-extend-norm-signed)
  finally show ?thesis .
qed

lemma int-to-bv-returntype [simp]: norm-signed (int-to-bv w) = int-to-bv w
  by (simp add: int-to-bv-def)

lemma bv-to-int-msb0: 0 ≤ bv-to-int w1 ==> bv-msb w1 = 0
  by (rule bit-list-cases,simp-all)

lemma bv-to-int-msb1: bv-to-int w1 < 0 ==> bv-msb w1 = 1
  by (rule bit-list-cases,simp-all)
```

lemma *bv-to-int-lower-limit-gt0*:
assumes *w0*: *0 < bv-to-int w*
shows *2 ^ (length (norm-signed w) − 2) ≤ bv-to-int w*
proof −
from *w0*
have *0 ≤ bv-to-int w* **by** *simp*
hence [*simp*]: *bv-msb w* = **0 by** (*rule bv-to-int-msb0*)
have *2 ^ (length (norm-signed w) − 2) ≤ bv-to-int (norm-signed w)*
proof (*rule bit-list-cases* [*of w*])
assume *w* = []
with *w0* **show** *?thesis* **by** *simp*
next
fix *w′*
assume *weq*: *w* = **0** # *w′*
thus *?thesis*
proof (*simp add*: *norm-signed-Cons*,*safe*)
assume *norm-unsigned w′* = []
with *weq* **and** *w0* **show** *False*
by (*simp add*: *norm-empty-bv-to-nat-zero*)
next
assume *w′0*: *norm-unsigned w′* ≠ []
have *0 < bv-to-nat w′*
proof (*rule ccontr*)
assume ~ (*0 < bv-to-nat w′*)
hence *bv-to-nat w′* = *0*
by *arith*
hence *norm-unsigned w′* = []
by (*simp add*: *bv-to-nat-zero-imp-empty*)
with *w′0*
show *False* **by** *simp*
qed
with *bv-to-nat-lower-limit* [*of w′*]
show *2 ^ (length (norm-unsigned w′) − Suc 0) ≤ int (bv-to-nat w′)*
by (*simp add*: *int-nat-two-exp*)
qed
next
fix *w′*
assume *w* = **1** # *w′*
from *w0* **have** *bv-msb w* = **0 by** *simp*
with *prems* **show** *?thesis* **by** *simp*
qed
also have ... = *bv-to-int w* **by** *simp*
finally show *?thesis* **.**
qed

lemma *norm-signed-result*: *norm-signed w* = [] ∨ *norm-signed w* = [**1**] ∨ *bv-msb* (*norm-signed w*) ≠ *bv-msb* (*tl* (*norm-signed w*))
apply (*rule bit-list-cases* [*of w*],*simp-all*)

```
  apply (case-tac bs,simp-all)
  apply (case-tac a,simp-all)
  apply (simp add: norm-signed-Cons)
  apply safe
  apply simp
proof −
  fix l
  assume msb: 0 = bv-msb (norm-unsigned l)
  assume norm-unsigned l ≠ []
  with norm-unsigned-result [of l]
  have bv-msb (norm-unsigned l) = 1 by simp
  with msb show False by simp
next
  fix xs
  assume p: 1 = bv-msb (tl (norm-signed (1 # xs)))
  have 1 ≠ bv-msb (tl (norm-signed (1 # xs)))
    by (rule bit-list-induct [of - xs],simp-all)
  with p show False by simp
qed

lemma bv-to-int-upper-limit-lem1:
  assumes w0: bv-to-int w < −1
  shows       bv-to-int w < − (2 ^ (length (norm-signed w) − 2))
proof −
  from w0
  have bv-to-int w < 0 by simp
  hence msbw [simp]: bv-msb w = 1
    by (rule bv-to-int-msb1)
  have bv-to-int w = bv-to-int (norm-signed w) by simp
  also from norm-signed-result [of w]
  have ... < − (2 ^ (length (norm-signed w) − 2))
  proof safe
    assume norm-signed w = []
    hence bv-to-int (norm-signed w) = 0 by simp
    with w0 show ?thesis by simp
  next
    assume norm-signed w = [1]
    hence bv-to-int (norm-signed w) = −1 by simp
    with w0 show ?thesis by simp
  next
    assume bv-msb (norm-signed w) ≠ bv-msb (tl (norm-signed w))
    hence msb-tl: 1 ≠ bv-msb (tl (norm-signed w)) by simp
    show bv-to-int (norm-signed w) < − (2 ^ (length (norm-signed w) − 2))
    proof (rule bit-list-cases [of norm-signed w])
      assume norm-signed w = []
      hence bv-to-int (norm-signed w) = 0 by simp
      with w0 show ?thesis by simp
    next
      fix w'
```

```
    assume nw: norm-signed w = 0 # w'
    from msbw have bv-msb (norm-signed w) = 1 by simp
    with nw show ?thesis by simp
   next
    fix w'
    assume weq: norm-signed w = 1 # w'
    show ?thesis
    proof (rule bit-list-cases [of w'])
     assume w'eq: w' = []
     from w0 have bv-to-int (norm-signed w) < -1 by simp
     with w'eq and weq show ?thesis by simp
    next
     fix w''
     assume w'eq: w' = 0 # w''
     show ?thesis
      apply (simp add: weq w'eq)
      apply (subgoal-tac - int (bv-to-nat (bv-not w'')) + -1 < 0 + 0)
      apply (simp add: int-nat-two-exp)
      apply (rule add-le-less-mono)
      apply simp-all
      done
    next
     fix w''
     assume w'eq: w' = 1 # w''
     with weq and msb-tl show ?thesis by simp
    qed
   qed
  qed
  finally show ?thesis .
qed

lemma length-int-to-bv-upper-limit-gt0:
  assumes w0: 0 < i
  and     wk: i ≤ 2 ^ (k - 1) - 1
  shows       length (int-to-bv i) ≤ k
proof (rule ccontr)
  from w0 wk
  have k1: 1 < k
    by (cases k - 1,simp-all)
  assume ~ length (int-to-bv i) ≤ k
  hence k < length (int-to-bv i) by simp
  hence k ≤ length (int-to-bv i) - 1 by arith
  hence a: k - 1 ≤ length (int-to-bv i) - 2 by arith
  hence (2::int) ^ (k - 1) ≤ 2 ^ (length (int-to-bv i) - 2) by simp
  also have ... ≤ i
  proof -
    have 2 ^ (length (norm-signed (int-to-bv i)) - 2) ≤ bv-to-int (int-to-bv i)
    proof (rule bv-to-int-lower-limit-gt0)
      from w0 show 0 < bv-to-int (int-to-bv i) by simp
```

```
    qed
    thus ?thesis by simp
  qed
  finally have 2 ^ (k - 1) ≤ i .
  with wk show False by simp
qed

lemma pos-length-pos:
  assumes i0: 0 < bv-to-int w
  shows       0 < length w
proof -
  from norm-signed-result [of w]
  have 0 < length (norm-signed w)
  proof (auto)
    assume ii: norm-signed w = []
    have bv-to-int (norm-signed w) = 0 by (subst ii) simp
    hence bv-to-int w = 0 by simp
    with i0 show False by simp
  next
    assume ii: norm-signed w = []
    assume jj: bv-msb w ≠ 0
    have 0 = bv-msb (norm-signed w)
      by (subst ii) simp
    also have ... ≠ 0
      by (simp add: jj)
    finally show False by simp
  qed
  also have ... ≤ length w
    by (rule norm-signed-length)
  finally show ?thesis .
qed

lemma neg-length-pos:
  assumes i0: bv-to-int w < -1
  shows       0 < length w
proof -
  from norm-signed-result [of w]
  have 0 < length (norm-signed w)
  proof (auto)
    assume ii: norm-signed w = []
    have bv-to-int (norm-signed w) = 0
      by (subst ii) simp
    hence bv-to-int w = 0 by simp
    with i0 show False by simp
  next
    assume ii: norm-signed w = []
    assume jj: bv-msb w ≠ 0
    have 0 = bv-msb (norm-signed w) by (subst ii) simp
    also have ... ≠ 0 by (simp add: jj)
```

```
    finally show False by simp
  qed
  also have ... ≤ length w
    by (rule norm-signed-length)
  finally show ?thesis .
qed

lemma length-int-to-bv-lower-limit-gt0:
  assumes wk: 2 ^ (k − 1) ≤ i
  shows       k < length (int-to-bv i)
proof (rule ccontr)
  have 0 < (2::int) ^ (k − 1)
    by (rule zero-less-power) simp
  also have ... ≤ i by (rule wk)
  finally have i0: 0 < i .
  have lii0: 0 < length (int-to-bv i)
    apply (rule pos-length-pos)
    apply (simp,rule i0)
    done
  assume ~ k < length (int-to-bv i)
  hence length (int-to-bv i) ≤ k by simp
  with lii0
  have a: length (int-to-bv i) − 1 ≤ k − 1
    by arith
  have i < 2 ^ (length (int-to-bv i) − 1)
  proof −
    have i = bv-to-int (int-to-bv i)
      by simp
    also have ... < 2 ^ (length (int-to-bv i) − 1)
      by (rule bv-to-int-upper-range)
    finally show ?thesis .
  qed
  also have (2::int) ^ (length (int-to-bv i) − 1) ≤ 2 ^ (k − 1) using a
    by simp
  finally have i < 2 ^ (k − 1) .
  with wk show False by simp
qed

lemma length-int-to-bv-upper-limit-lem1:
  assumes w1: i < −1
  and       wk: − (2 ^ (k − 1)) ≤ i
  shows       length (int-to-bv i) ≤ k
proof (rule ccontr)
  from w1 wk
  have k1: 1 < k by (cases k − 1) simp-all
  assume ~ length (int-to-bv i) ≤ k
  hence k < length (int-to-bv i) by simp
  hence k ≤ length (int-to-bv i) − 1 by arith
  hence a: k − 1 ≤ length (int-to-bv i) − 2 by arith
```

```
  have i < - (2 ^ (length (int-to-bv i) - 2))
  proof -
    have i = bv-to-int (int-to-bv i)
      by simp
    also have ... < - (2 ^ (length (norm-signed (int-to-bv i)) - 2))
      by (rule bv-to-int-upper-limit-lem1,simp,rule w1)
    finally show ?thesis by simp
  qed
  also have ... ≤ -(2 ^ (k - 1))
  proof -
    have (2::int) ^ (k - 1) ≤ 2 ^ (length (int-to-bv i) - 2) using a by simp
    thus ?thesis by simp
  qed
  finally have i < -(2 ^ (k - 1)) .
  with wk show False by simp
qed

lemma length-int-to-bv-lower-limit-lem1:
  assumes wk: i < -(2 ^ (k - 1))
  shows       k < length (int-to-bv i)
proof (rule ccontr)
  from wk have i ≤ -(2 ^ (k - 1)) - 1 by simp
  also have ... < -1
  proof -
    have 0 < (2::int) ^ (k - 1)
      by (rule zero-less-power) simp
    hence -((2::int) ^ (k - 1)) < 0 by simp
    thus ?thesis by simp
  qed
  finally have i1: i < -1 .
  have lii0: 0 < length (int-to-bv i)
    apply (rule neg-length-pos)
    apply (simp, rule i1)
    done
  assume ~ k < length (int-to-bv i)
  hence length (int-to-bv i) ≤ k
    by simp
  with lii0 have a: length (int-to-bv i) - 1 ≤ k - 1 by arith
  hence (2::int) ^ (length (int-to-bv i) - 1) ≤ 2 ^ (k - 1) by simp
  hence -((2::int) ^ (k - 1)) ≤ - (2 ^ (length (int-to-bv i) - 1)) by simp
  also have ... ≤ i
  proof -
    have - (2 ^ (length (int-to-bv i) - 1)) ≤ bv-to-int (int-to-bv i)
      by (rule bv-to-int-lower-range)
    also have ... = i
      by simp
    finally show ?thesis .
  qed
  finally have -(2 ^ (k - 1)) ≤ i .
```

with *wk* **show** *False* **by** *simp*
qed

54.6 Signed Arithmetic Operations

54.6.1 Conversion from unsigned to signed

definition
utos :: *bit list* => *bit list* **where**
utos w = *norm-signed* (**0** # *w*)

lemma *utos-type* [*simp*]: *utos* (*norm-unsigned w*) = *utos w*
by (*simp add*: *utos-def norm-signed-Cons*)

lemma *utos-returntype* [*simp*]: *norm-signed* (*utos w*) = *utos w*
by (*simp add*: *utos-def*)

lemma *utos-length*: *length* (*utos w*) ≤ *Suc* (*length w*)
by (*simp add*: *utos-def norm-signed-Cons*)

lemma *bv-to-int-utos*: *bv-to-int* (*utos w*) = *int* (*bv-to-nat w*)
proof (*simp add*: *utos-def norm-signed-Cons*, *safe*)
assume *norm-unsigned w* = []
hence *bv-to-nat* (*norm-unsigned w*) = *0* **by** *simp*
thus *bv-to-nat w* = *0* **by** *simp*
qed

54.6.2 Unary minus

definition
bv-uminus :: *bit list* => *bit list* **where**
bv-uminus w = *int-to-bv* (− *bv-to-int w*)

lemma *bv-uminus-type* [*simp*]: *bv-uminus* (*norm-signed w*) = *bv-uminus w*
by (*simp add*: *bv-uminus-def*)

lemma *bv-uminus-returntype* [*simp*]: *norm-signed* (*bv-uminus w*) = *bv-uminus w*
by (*simp add*: *bv-uminus-def*)

lemma *bv-uminus-length*: *length* (*bv-uminus w*) ≤ *Suc* (*length w*)
proof −
have *1* < −*bv-to-int w* ∨ −*bv-to-int w* = *1* ∨ −*bv-to-int w* = *0* ∨ −*bv-to-int w* = −*1* ∨ −*bv-to-int w* < −*1*
by *arith*
thus *?thesis*
proof *safe*
assume *p*: *1* < − *bv-to-int w*
have *lw*: *0* < *length w*
apply (*rule neg-length-pos*)
using *p*

```
    apply simp
    done
  show ?thesis
  proof (simp add: bv-uminus-def,rule length-int-to-bv-upper-limit-gt0,simp-all)
    from prems show bv-to-int w < 0 by simp
  next
    have -(2^(length w - 1)) ≤ bv-to-int w
      by (rule bv-to-int-lower-range)
    hence - bv-to-int w ≤ 2^(length w - 1) by simp
    also from lw have ... < 2 ^ length w by simp
    finally show - bv-to-int w < 2 ^ length w by simp
  qed
next
  assume p: - bv-to-int w = 1
  hence lw: 0 < length w by (cases w) simp-all
  from p
  show ?thesis
    apply (simp add: bv-uminus-def)
    using lw
    apply (simp (no-asm) add: nat-to-bv-non0)
    done
next
  assume - bv-to-int w = 0
  thus ?thesis by (simp add: bv-uminus-def)
next
  assume p: - bv-to-int w = -1
  thus ?thesis by (simp add: bv-uminus-def)
next
  assume p: - bv-to-int w < -1
  show ?thesis
    apply (simp add: bv-uminus-def)
    apply (rule length-int-to-bv-upper-limit-lem1)
    apply (rule p)
    apply simp
  proof -
    have bv-to-int w < 2 ^ (length w - 1)
      by (rule bv-to-int-upper-range)
    also have ... ≤ 2 ^ length w by simp
    finally show bv-to-int w ≤ 2 ^ length w by simp
  qed
qed
qed

lemma bv-uminus-length-utos: length (bv-uminus (utos w)) ≤ Suc (length w)
proof -
  have -bv-to-int (utos w) = 0 ∨ -bv-to-int (utos w) = -1 ∨ -bv-to-int (utos
w) < -1
    by (simp add: bv-to-int-utos, arith)
  thus ?thesis
```

```
  proof safe
    assume −bv-to-int (utos w) = 0
    thus ?thesis by (simp add: bv-uminus-def)
  next
    assume −bv-to-int (utos w) = −1
    thus ?thesis by (simp add: bv-uminus-def)
  next
    assume p: −bv-to-int (utos w) < −1
    show ?thesis
      apply (simp add: bv-uminus-def)
      apply (rule length-int-to-bv-upper-limit-lem1)
      apply (rule p)
      apply (simp add: bv-to-int-utos)
      using bv-to-nat-upper-range [of w]
      apply (simp add: int-nat-two-exp)
      done
  qed
qed

definition
  bv-sadd :: [bit list, bit list ] => bit list where
  bv-sadd w1 w2 = int-to-bv (bv-to-int w1 + bv-to-int w2)

lemma bv-sadd-type1 [simp]: bv-sadd (norm-signed w1) w2 = bv-sadd w1 w2
  by (simp add: bv-sadd-def)

lemma bv-sadd-type2 [simp]: bv-sadd w1 (norm-signed w2) = bv-sadd w1 w2
  by (simp add: bv-sadd-def)

lemma bv-sadd-returntype [simp]: norm-signed (bv-sadd w1 w2) = bv-sadd w1 w2
  by (simp add: bv-sadd-def)

lemma adder-helper:
  assumes lw: 0 < max (length w1) (length w2)
  shows  ((2::int) ^ (length w1 − 1)) + (2 ^ (length w2 − 1)) ≤ 2 ^ max (length
w1) (length w2)
proof −
  have ((2::int) ^ (length w1 − 1)) + (2 ^ (length w2 − 1)) ≤
      2 ^ (max (length w1) (length w2) − 1) + 2 ^ (max (length w1) (length w2)
− 1)
    apply (cases length w1 ≤ length w2)
    apply (auto simp add: max-def)
    done
  also have ... = 2 ^ max (length w1) (length w2)
  proof −
    from lw
    show ?thesis
      apply simp
      apply (subst power-Suc [symmetric])
```

```
    apply (simp del: power-int.simps)
    done
 qed
 finally show ?thesis .
qed

lemma bv-sadd-length: length (bv-sadd w1 w2) ≤ Suc (max (length w1) (length
w2))
proof −
 let ?Q = bv-to-int w1 + bv-to-int w2

 have helper: ?Q ≠ 0 ==> 0 < max (length w1) (length w2)
 proof −
  assume p: ?Q ≠ 0
  show 0 < max (length w1) (length w2)
  proof (simp add: less-max-iff-disj,rule)
   assume [simp]: w1 = []
   show w2 ≠ []
   proof (rule ccontr,simp)
    assume [simp]: w2 = []
    from p show False by simp
   qed
  qed
 qed

 have 0 < ?Q ∨ ?Q = 0 ∨ ?Q = −1 ∨ ?Q < −1 by arith
 thus ?thesis
 proof safe
  assume ?Q = 0
  thus ?thesis
   by (simp add: bv-sadd-def)
 next
  assume ?Q = −1
  thus ?thesis
   by (simp add: bv-sadd-def)
 next
  assume p: 0 < ?Q
  show ?thesis
   apply (simp add: bv-sadd-def)
   apply (rule length-int-to-bv-upper-limit-gt0)
   apply (rule p)
  proof simp
   from bv-to-int-upper-range [of w2]
   have bv-to-int w2 ≤ 2 ^ (length w2 − 1)
    by simp
   with bv-to-int-upper-range [of w1]
   have bv-to-int w1 + bv-to-int w2 < (2 ^ (length w1 − 1)) + (2 ^ (length w2
− 1))
    by (rule zadd-zless-mono)
```

```
    also have ... ≤ 2 ^ max (length w1) (length w2)
      apply (rule adder-helper)
      apply (rule helper)
      using p
      apply simp
      done
    finally show ?Q < 2 ^ max (length w1) (length w2) .
  qed
 next
  assume p: ?Q < -1
  show ?thesis
    apply (simp add: bv-sadd-def)
    apply (rule length-int-to-bv-upper-limit-lem1,simp-all)
    apply (rule p)
  proof -
    have (2 ^ (length w1 - 1)) + 2 ^ (length w2 - 1) ≤ (2::int) ^ max (length
w1) (length w2)
      apply (rule adder-helper)
      apply (rule helper)
      using p
      apply simp
      done
    hence -((2::int) ^ max (length w1) (length w2)) ≤ - (2 ^ (length w1 - 1))
+ -(2 ^ (length w2 - 1))
      by simp
    also have - (2 ^ (length w1 - 1)) + -(2 ^ (length w2 - 1)) ≤ ?Q
      apply (rule add-mono)
      apply (rule bv-to-int-lower-range [of w1])
      apply (rule bv-to-int-lower-range [of w2])
      done
    finally show - (2^max (length w1) (length w2)) ≤ ?Q .
  qed
 qed
qed

definition
  bv-sub :: [bit list, bit list] => bit list where
  bv-sub w1 w2 = bv-sadd w1 (bv-uminus w2)

lemma bv-sub-type1 [simp]: bv-sub (norm-signed w1) w2 = bv-sub w1 w2
  by (simp add: bv-sub-def)

lemma bv-sub-type2 [simp]: bv-sub w1 (norm-signed w2) = bv-sub w1 w2
  by (simp add: bv-sub-def)

lemma bv-sub-returntype [simp]: norm-signed (bv-sub w1 w2) = bv-sub w1 w2
  by (simp add: bv-sub-def)

lemma bv-sub-length: length (bv-sub w1 w2) ≤ Suc (max (length w1) (length w2))
```

proof (*cases bv-to-int w2 = 0*)
assume *p*: *bv-to-int w2 = 0*
show *?thesis*
proof (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def p*)
have *length* (*norm-signed w1*) ≤ *length w1*
by (*rule norm-signed-length*)
also have ... ≤ *max* (*length w1*) (*length w2*)
by (*rule le-maxI1*)
also have ... ≤ *Suc* (*max* (*length w1*) (*length w2*))
by *arith*
finally show *length* (*norm-signed w1*) ≤ *Suc* (*max* (*length w1*) (*length w2*)) **.**
qed
next
assume *bv-to-int w2* ≠ *0*
hence *0* < *length w2* **by** (*cases w2*,*simp-all*)
hence *lmw*: *0* < *max* (*length w1*) (*length w2*) **by** *arith*

let *?Q* = *bv-to-int w1* − *bv-to-int w2*

have *0* < *?Q* ∨ *?Q* = *0* ∨ *?Q* = −*1* ∨ *?Q* < −*1* **by** *arith*
thus *?thesis*
proof *safe*
assume *?Q* = *0*
thus *?thesis*
by (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
next
assume *?Q* = −*1*
thus *?thesis*
by (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
next
assume *p*: *0* < *?Q*
show *?thesis*
apply (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
apply (*rule length-int-to-bv-upper-limit-gt0*)
apply (*rule p*)
proof *simp*
from *bv-to-int-lower-range* [*of w2*]
have *v2*: − *bv-to-int w2* ≤ *2* ^ (*length w2* − *1*) **by** *simp*
have *bv-to-int w1* + − *bv-to-int w2* < (*2* ^ (*length w1* − *1*)) + (*2* ^ (*length w2* − *1*))
apply (*rule zadd-zless-mono*)
apply (*rule bv-to-int-upper-range* [*of w1*])
apply (*rule v2*)
done
also have ... ≤ *2* ^ *max* (*length w1*) (*length w2*)
apply (*rule adder-helper*)
apply (*rule lmw*)
done
finally show *?Q* < *2* ^ *max* (*length w1*) (*length w2*) **by** *simp*

```
    qed
  next
    assume p: ?Q < −1
    show ?thesis
      apply (simp add: bv-sub-def bv-sadd-def bv-uminus-def)
      apply (rule length-int-to-bv-upper-limit-lem1)
      apply (rule p)
    proof simp
      have (2 ^ (length w1 − 1)) + 2 ^ (length w2 − 1) ≤ (2::int) ^ max (length
w1) (length w2)
        apply (rule adder-helper)
        apply (rule lmw)
        done
      hence −((2::int) ^ max (length w1) (length w2)) ≤ − (2 ^ (length w1 − 1))
+ −(2 ^ (length w2 − 1))
        by simp
      also have − (2 ^ (length w1 − 1)) + −(2 ^ (length w2 − 1)) ≤ bv-to-int w1
+ −bv-to-int w2
        apply (rule add-mono)
        apply (rule bv-to-int-lower-range [of w1])
        using bv-to-int-upper-range [of w2]
        apply simp
        done
      finally show − (2^max (length w1) (length w2)) ≤ ?Q by simp
    qed
  qed
qed

definition
  bv-smult :: [bit list, bit list] => bit list where
  bv-smult w1 w2 = int-to-bv (bv-to-int w1 * bv-to-int w2)

lemma bv-smult-type1 [simp]: bv-smult (norm-signed w1) w2 = bv-smult w1 w2
  by (simp add: bv-smult-def)

lemma bv-smult-type2 [simp]: bv-smult w1 (norm-signed w2) = bv-smult w1 w2
  by (simp add: bv-smult-def)

lemma bv-smult-returntype [simp]: norm-signed (bv-smult w1 w2) = bv-smult w1
w2
  by (simp add: bv-smult-def)

lemma bv-smult-length: length (bv-smult w1 w2) ≤ length w1 + length w2
proof −
  let ?Q = bv-to-int w1 * bv-to-int w2

  have lmw: ?Q ≠ 0 ==> 0 < length w1 ∧ 0 < length w2 by auto

  have 0 < ?Q ∨ ?Q = 0 ∨ ?Q = −1 ∨ ?Q < −1 by arith
```

thus *?thesis*
proof (*safe dest*!: *iffD1* [*OF mult-eq-0-iff*])
assume *bv-to-int w1 = 0*
thus *?thesis* **by** (*simp add*: *bv-smult-def*)
next
assume *bv-to-int w2 = 0*
thus *?thesis* **by** (*simp add*: *bv-smult-def*)
next
assume *p*: *?Q = −1*
show *?thesis*
apply (*simp add*: *bv-smult-def p*)
apply (*cut-tac lmw*)
apply *arith*
using *p*
apply *simp*
done
next
assume *p*: *0 < ?Q*
thus *?thesis*
proof (*simp add*: *zero-less-mult-iff*,*safe*)
assume *bi1*: *0 < bv-to-int w1*
assume *bi2*: *0 < bv-to-int w2*
show *?thesis*
apply (*simp add*: *bv-smult-def*)
apply (*rule length-int-to-bv-upper-limit-gt0*)
apply (*rule p*)
proof *simp*
have *?Q < 2 ^ (length w1 − 1) * 2 ^ (length w2 − 1)*
apply (*rule mult-strict-mono*)
apply (*rule bv-to-int-upper-range*)
apply (*rule bv-to-int-upper-range*)
apply (*rule zero-less-power*)
apply *simp*
using *bi2*
apply *simp*
done
also have *... ≤ 2 ^ (length w1 + length w2 − Suc 0)*
apply *simp*
apply (*subst zpower-zadd-distrib* [*symmetric*])
apply *simp*
done
finally show *?Q < 2 ^ (length w1 + length w2 − Suc 0)* **.**
qed
next
assume *bi1*: *bv-to-int w1 < 0*
assume *bi2*: *bv-to-int w2 < 0*
show *?thesis*
apply (*simp add*: *bv-smult-def*)
apply (*rule length-int-to-bv-upper-limit-gt0*)

```
      apply (rule p)
     proof simp
      have −bv-to-int w1 ∗ −bv-to-int w2 ≤ 2 ^ (length w1 − 1) ∗ 2 ^(length w2
− 1)
        apply (rule mult-mono)
        using bv-to-int-lower-range [of w1]
        apply simp
        using bv-to-int-lower-range [of w2]
        apply simp
        apply (rule zero-le-power,simp)
        using bi2
        apply simp
        done
       hence ?Q ≤ 2 ^ (length w1 − 1) ∗ 2 ^(length w2 − 1)
        by simp
       also have ... < 2 ^ (length w1 + length w2 − Suc 0)
        apply simp
        apply (subst zpower-zadd-distrib [symmetric])
        apply simp
        apply (cut-tac lmw)
        apply arith
        apply (cut-tac p)
        apply arith
        done
       finally show ?Q < 2 ^ (length w1 + length w2 − Suc 0) .
     qed
   qed
  next
   assume p: ?Q < −1
   show ?thesis
    apply (subst bv-smult-def)
    apply (rule length-int-to-bv-upper-limit-lem1)
    apply (rule p)
   proof simp
     have (2::int) ^ (length w1 − 1) ∗ 2 ^(length w2 − 1) ≤ 2 ^ (length w1 +
length w2 − Suc 0)
      apply simp
      apply (subst zpower-zadd-distrib [symmetric])
      apply simp
      done
     hence −((2::int) ^ (length w1 + length w2 − Suc 0)) ≤ −(2^(length w1 −
1) ∗ 2 ^ (length w2 − 1))
      by simp
     also have ... ≤ ?Q
     proof −
      from p
      have q: bv-to-int w1 ∗ bv-to-int w2 < 0
       by simp
      thus ?thesis
```

```
      proof (simp add: mult-less-0-iff,safe)
        assume bi1: 0 < bv-to-int w1
        assume bi2: bv-to-int w2 < 0
         have -bv-to-int w2 * bv-to-int w1 ≤ ((2::int) ^(length w2 - 1)) * (2 ^
(length w1 - 1))
          apply (rule mult-mono)
          using bv-to-int-lower-range [of w2]
          apply simp
          using bv-to-int-upper-range [of w1]
          apply simp
          apply (rule zero-le-power,simp)
          using bi1
          apply simp
          done
        hence -?Q ≤ ((2::int) ^(length w1 - 1)) * (2 ^ (length w2 - 1))
          by (simp add: zmult-ac)
         thus -(((2::int) ^(length w1 - Suc 0)) * (2 ^ (length w2 - Suc 0))) ≤
?Q
          by simp
      next
        assume bi1: bv-to-int w1 < 0
        assume bi2: 0 < bv-to-int w2
         have -bv-to-int w1 * bv-to-int w2 ≤ ((2::int) ^(length w1 - 1)) * (2 ^
(length w2 - 1))
          apply (rule mult-mono)
          using bv-to-int-lower-range [of w1]
          apply simp
          using bv-to-int-upper-range [of w2]
          apply simp
          apply (rule zero-le-power,simp)
          using bi2
          apply simp
          done
        hence -?Q ≤ ((2::int) ^(length w1 - 1)) * (2 ^ (length w2 - 1))
          by (simp add: zmult-ac)
         thus -(((2::int) ^(length w1 - Suc 0)) * (2 ^ (length w2 - Suc 0))) ≤
?Q
          by simp
      qed
    qed
    finally show -(2 ^ (length w1 + length w2 - Suc 0)) ≤ ?Q .
  qed
 qed
qed

lemma bv-msb-one: bv-msb w = 1 ==> bv-to-nat w ≠ 0
by (cases w) simp-all

lemma bv-smult-length-utos: length (bv-smult (utos w1) w2) ≤ length w1 + length
```

```
w2
proof −
  let ?Q = bv-to-int (utos w1) ∗ bv-to-int w2

  have lmw: ?Q ≠ 0 ==> 0 < length (utos w1) ∧ 0 < length w2 by auto

  have 0 < ?Q ∨ ?Q = 0 ∨ ?Q = −1 ∨ ?Q < −1 by arith
  thus ?thesis
  proof (safe dest!: iffD1 [OF mult-eq-0-iff])
    assume bv-to-int (utos w1) = 0
    thus ?thesis by (simp add: bv-smult-def)
  next
    assume bv-to-int w2 = 0
    thus ?thesis by (simp add: bv-smult-def)
  next
    assume p: 0 < ?Q
    thus ?thesis
    proof (simp add: zero-less-mult-iff,safe)
      assume biw2: 0 < bv-to-int w2
      show ?thesis
        apply (simp add: bv-smult-def)
        apply (rule length-int-to-bv-upper-limit-gt0)
        apply (rule p)
      proof simp
        have ?Q < 2 ^ length w1 ∗ 2 ^ (length w2 − 1)
          apply (rule mult-strict-mono)
          apply (simp add: bv-to-int-utos int-nat-two-exp)
          apply (rule bv-to-nat-upper-range)
          apply (rule bv-to-int-upper-range)
          apply (rule zero-less-power,simp)
          using biw2
          apply simp
          done
        also have ... ≤ 2 ^ (length w1 + length w2 − Suc 0)
          apply simp
          apply (subst zpower-zadd-distrib [symmetric])
          apply simp
          apply (cut-tac lmw)
          apply arith
          using p
          apply auto
          done
        finally show ?Q < 2 ^ (length w1 + length w2 − Suc 0) .
      qed
    next
      assume bv-to-int (utos w1) < 0
      thus ?thesis by (simp add: bv-to-int-utos)
    qed
  next
```

assume *p*: *?Q = −1*
thus *?thesis*
apply (*simp add*: *bv-smult-def*)
apply (*cut-tac lmw*)
apply *arith*
apply *simp*
done
next
assume *p*: *?Q < −1*
show *?thesis*
apply (*subst bv-smult-def*)
apply (*rule length-int-to-bv-upper-limit-lem1*)
apply (*rule p*)
proof *simp*
have (*2*::*int*) ^ *length w1 * 2* ^(*length w2 − 1*) ≤ *2* ^ (*length w1 + length w2 − Suc 0*)
apply *simp*
apply (*subst zpower-zadd-distrib* [*symmetric*])
apply *simp*
apply (*cut-tac lmw*)
apply *arith*
apply (*cut-tac p*)
apply *arith*
done
hence −((*2*::*int*) ^ (*length w1 + length w2 − Suc 0*)) ≤ −(*2* ^ *length w1 * 2* ^ (*length w2 − 1*))
by *simp*
also have ... ≤ *?Q*
proof −
from *p*
have *q*: *bv-to-int* (*utos w1*) * *bv-to-int w2 < 0*
by *simp*
thus *?thesis*
proof (*simp add*: *mult-less-0-iff*,*safe*)
assume *bi1*: *0 < bv-to-int* (*utos w1*)
assume *bi2*: *bv-to-int w2 < 0*
have −*bv-to-int w2 * bv-to-int* (*utos w1*) ≤ ((*2*::*int*) ^(*length w2 − 1*)) * (*2* ^ *length w1*)
apply (*rule mult-mono*)
using *bv-to-int-lower-range* [*of w2*]
apply *simp*
apply (*simp add*: *bv-to-int-utos*)
using *bv-to-nat-upper-range* [*of w1*]
apply (*simp add*: *int-nat-two-exp*)
apply (*rule zero-le-power*,*simp*)
using *bi1*
apply *simp*
done
hence −*?Q* ≤ ((*2*::*int*) ^*length w1*) * (*2* ^ (*length w2 − 1*))

```
        by (simp add: zmult-ac)
      thus -(((2::int) ^length w1) * (2 ^ (length w2 - Suc 0))) ≤ ?Q
        by simp
    next
      assume bi1: bv-to-int (utos w1) < 0
      thus -(((2::int) ^length w1) * (2 ^ (length w2 - Suc 0))) ≤ ?Q
        by (simp add: bv-to-int-utos)
    qed
  qed
  finally show -(2 ^ (length w1 + length w2 - Suc 0)) ≤ ?Q .
 qed
qed
qed

lemma bv-smult-sym: bv-smult w1 w2 = bv-smult w2 w1
 by (simp add: bv-smult-def zmult-ac)
```

54.7 Structural operations

```
definition
 bv-select :: [bit list,nat] => bit where
 bv-select w i = w ! (length w - 1 - i)

definition
 bv-chop :: [bit list,nat] => bit list * bit list where
 bv-chop w i = (let len = length w in (take (len - i) w,drop (len - i) w))

definition
 bv-slice :: [bit list,nat*nat] => bit list where
 bv-slice w = (λ(b,e). fst (bv-chop (snd (bv-chop w (b+1))) e))

lemma bv-select-rev:
 assumes notnull: n < length w
 shows          bv-select w n = rev w ! n
proof -
 have ∀n. n < length w --> bv-select w n = rev w ! n
 proof (rule length-induct [of - w],auto simp add: bv-select-def)
   fix xs :: bit list
   fix n
  assume ind: ∀ys::bit list. length ys < length xs --> (∀n. n < length ys -->
ys ! (length ys - Suc n) = rev ys ! n)
   assume notx: n < length xs
   show xs ! (length xs - Suc n) = rev xs ! n
   proof (cases xs)
     assume xs = []
     with notx show ?thesis by simp
   next
     fix y ys
     assume [simp]: xs = y # ys
```

```
      show ?thesis
      proof (auto simp add: nth-append)
        assume noty: n < length ys
        from spec [OF ind,of ys]
        have ∀ n. n < length ys −−> ys ! (length ys − Suc n) = rev ys ! n
          by simp
        hence n < length ys −−> ys ! (length ys − Suc n) = rev ys ! n ..
        from this and noty
        have ys ! (length ys − Suc n) = rev ys ! n ..
        thus (y # ys) ! (length ys − n) = rev ys ! n
          by (simp add: nth-Cons' noty linorder-not-less [symmetric])
      next
        assume ~ n < length ys
        hence x: length ys ≤ n by simp
        from notx have n < Suc (length ys) by simp
        hence n ≤ length ys by simp
        with x have length ys = n by simp
        thus y = [y] ! (n − length ys) by simp
      qed
    qed
  qed
  then have n < length w −−> bv-select w n = rev w ! n ..
  from this and notnull show ?thesis ..
qed

lemma bv-chop-append: bv-chop (w1 @ w2) (length w2) = (w1,w2)
  by (simp add: bv-chop-def Let-def)

lemma append-bv-chop-id: fst (bv-chop w l) @ snd (bv-chop w l) = w
  by (simp add: bv-chop-def Let-def)

lemma bv-chop-length-fst [simp]: length (fst (bv-chop w i)) = length w − i
  by (simp add: bv-chop-def Let-def)

lemma bv-chop-length-snd [simp]: length (snd (bv-chop w i)) = min i (length w)
  by (simp add: bv-chop-def Let-def)

lemma bv-slice-length [simp]: [| j ≤ i; i < length w |] ==> length (bv-slice w
(i,j)) = i − j + 1
  by (auto simp add: bv-slice-def)

definition
  length-nat :: nat => nat where
  length-nat x = (LEAST n. x < 2 ^ n)

lemma length-nat: length (nat-to-bv n) = length-nat n
  apply (simp add: length-nat-def)
  apply (rule Least-equality [symmetric])
  prefer 2
```

```
  apply (rule length-nat-to-bv-upper-limit)
  apply arith
  apply (rule ccontr)
proof −
  assume ~ n < 2 ^ length (nat-to-bv n)
  hence 2 ^ length (nat-to-bv n) ≤ n by simp
  hence length (nat-to-bv n) < length (nat-to-bv n)
    by (rule length-nat-to-bv-lower-limit)
  thus False by simp
qed

lemma length-nat-0 [simp]: length-nat 0 = 0
  by (simp add: length-nat-def Least-equality)

lemma length-nat-non0:
  assumes n0: n ≠ 0
  shows       length-nat n = Suc (length-nat (n div 2))
  apply (simp add: length-nat [symmetric])
  apply (subst nat-to-bv-non0 [of n])
  apply (simp-all add: n0)
  done

definition
  length-int :: int => nat where
  length-int x =
    (if 0 < x then Suc (length-nat (nat x))
    else if x = 0 then 0
    else Suc (length-nat (nat (−x − 1))))

lemma length-int: length (int-to-bv i) = length-int i
proof (cases 0 < i)
  assume i0: 0 < i
  hence length (int-to-bv i) =
      length (norm-signed (0 # norm-unsigned (nat-to-bv (nat i)))) by simp
  also from norm-unsigned-result [of nat-to-bv (nat i)]
  have ... = Suc (length-nat (nat i))
    apply safe
    apply (simp del: norm-unsigned-nat-to-bv)
    apply (drule norm-empty-bv-to-nat-zero)
    using prems
    apply simp
    apply (cases norm-unsigned (nat-to-bv (nat i)))
    apply (drule norm-empty-bv-to-nat-zero [of nat-to-bv (nat i)])
    using prems
    apply simp
    apply simp
    using prems
    apply (simp add: length-nat [symmetric])
    done
```

```
  finally show ?thesis
    using i0
    by (simp add: length-int-def)
next
  assume ~ 0 < i
  hence i0: i ≤ 0 by simp
  show ?thesis
  proof (cases i = 0)
    assume i = 0
    thus ?thesis by (simp add: length-int-def)
  next
    assume i ≠ 0
    with i0 have i0: i < 0 by simp
    hence length (int-to-bv i) =
        length (norm-signed (1 # bv-not (norm-unsigned (nat-to-bv (nat (− i) −
1)))))
      by (simp add: int-to-bv-def nat-diff-distrib)
    also from norm-unsigned-result [of nat-to-bv (nat (− i) − 1)]
    have ... = Suc (length-nat (nat (− i) − 1))
      apply safe
      apply (simp del: norm-unsigned-nat-to-bv)
      apply (drule norm-empty-bv-to-nat-zero [of nat-to-bv (nat (−i) − Suc 0)])
      using prems
      apply simp
      apply (cases − i − 1 = 0)
      apply simp
      apply (simp add: length-nat [symmetric])
      apply (cases norm-unsigned (nat-to-bv (nat (− i) − 1)))
      apply simp
      apply simp
      done
    finally
    show ?thesis
      using i0 by (simp add: length-int-def nat-diff-distrib del: int-to-bv-lt0)
  qed
qed

lemma length-int-0 [simp]: length-int 0 = 0
  by (simp add: length-int-def)

lemma length-int-gt0: 0 < i ==> length-int i = Suc (length-nat (nat i))
  by (simp add: length-int-def)

lemma length-int-lt0: i < 0 ==> length-int i = Suc (length-nat (nat (− i) − 1))
  by (simp add: length-int-def nat-diff-distrib)

lemma bv-chopI: [| w = w1 @ w2 ; i = length w2 |] ==> bv-chop w i = (w1,w2)
  by (simp add: bv-chop-def Let-def)
```

lemma *bv-sliceI*: [| $j \leq i$; $i < length\ w$; $w = w1\ @\ w2\ @\ w3$; $Suc\ i = length\ w2 + j$; $j = length\ w3$ |] ==> *bv-slice w* $(i,j) = w2$
apply (*simp add*: *bv-slice-def*)
apply (*subst bv-chopI* [*of w1 @ w2 @ w3 w1 w2 @ w3*])
apply *simp*
apply *simp*
apply *simp*
apply (*subst bv-chopI* [*of w2 @ w3 w2 w3*],*simp-all*)
done

lemma *bv-slice-bv-slice*:
assumes *ki*: $k \leq i$
and *ij*: $i \leq j$
and *jl*: $j \leq l$
and *lw*: $l < length\ w$
shows *bv-slice w* $(j,i) =$ *bv-slice* (*bv-slice w* (l,k)) $(j-k,i-k)$
proof −
def *w1* == *fst* (*bv-chop w* (*Suc l*))
and *w2* == *fst* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))
and *w3* == *fst* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))) *i*)
and *w4* == *fst* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))) *i*)) *k*)
and *w5* == *snd* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))) *i*)) *k*)
note *w-defs* = *this*

have *w-def*: $w = w1\ @\ w2\ @\ w3\ @\ w4\ @\ w5$
by (*simp add*: *w-defs append-bv-chop-id*)

from *ki ij jl lw*
show *?thesis*
apply (*subst bv-sliceI* [**where** *?j* = *i* **and** *?i* = *j* **and** *?w* = *w* **and** *?w1.0* = *w1 @ w2* **and** *?w2.0* = *w3* **and** *?w3.0* = *w4 @ w5*])
apply *simp-all*
apply (*rule w-def*)
apply (*simp add*: *w-defs min-def*)
apply (*simp add*: *w-defs min-def*)
apply (*subst bv-sliceI* [**where** *?j* = *k* **and** *?i* = *l* **and** *?w* = *w* **and** *?w1.0* = *w1* **and** *?w2.0* = *w2 @ w3 @ w4* **and** *?w3.0* = *w5*])
apply *simp-all*
apply (*rule w-def*)
apply (*simp add*: *w-defs min-def*)
apply (*simp add*: *w-defs min-def*)
apply (*subst bv-sliceI* [**where** *?j* = $i-k$ **and** *?i* = $j-k$ **and** *?w* = *w2 @ w3 @ w4* **and** *?w1.0* = *w2* **and** *?w2.0* = *w3* **and** *?w3.0* = *w4*])
apply *simp-all*
apply (*simp-all add*: *w-defs min-def*)
done
qed

lemma *bv-to-nat-extend* [*simp*]: *bv-to-nat* (*bv-extend n* **0** *w*) = *bv-to-nat w*
apply (*simp add*: *bv-extend-def*)
apply (*subst bv-to-nat-dist-append*)
apply *simp*
apply (*induct n* − *length w*)
apply *simp-all*
done

lemma *bv-msb-extend-same* [*simp*]: *bv-msb w* = *b* ==> *bv-msb* (*bv-extend n b w*) = *b*
apply (*simp add*: *bv-extend-def*)
apply (*induct n* − *length w*)
apply *simp-all*
done

lemma *bv-to-int-extend* [*simp*]:
assumes *a*: *bv-msb w* = *b*
shows *bv-to-int* (*bv-extend n b w*) = *bv-to-int w*
proof (*cases bv-msb w*)
assume [*simp*]: *bv-msb w* = **0**
with *a* **have** [*simp*]: *b* = **0** **by** *simp*
show *?thesis* **by** (*simp add*: *bv-to-int-def*)
next
assume [*simp*]: *bv-msb w* = **1**
with *a* **have** [*simp*]: *b* = **1** **by** *simp*
show *?thesis*
apply (*simp add*: *bv-to-int-def*)
apply (*simp add*: *bv-extend-def*)
apply (*induct n* − *length w*,*simp-all*)
done
qed

lemma *length-nat-mono* [*simp*]: *x* ≤ *y* ==> *length-nat x* ≤ *length-nat y*
proof (*rule ccontr*)
assume *xy*: *x* ≤ *y*
assume ~ *length-nat x* ≤ *length-nat y*
hence *lxly*: *length-nat y* < *length-nat x*
by *simp*
hence *length-nat y* < (*LEAST n*. *x* < *2* ^ *n*)
by (*simp add*: *length-nat-def*)
hence ~ *x* < *2* ^ *length-nat y*
by (*rule not-less-Least*)
hence *xx*: *2* ^ *length-nat y* ≤ *x*
by *simp*
have *yy*: *y* < *2* ^ *length-nat y*
apply (*simp add*: *length-nat-def*)
apply (*rule LeastI*)
apply (*subgoal-tac y* < *2* ^ *y*,*assumption*)

```
    apply (cases 0 ≤ y)
    apply (induct y,simp-all)
    done
  with xx have y < x by simp
  with xy show False by simp
qed

lemma length-nat-mono-int [simp]: x ≤ y ==> length-nat x ≤ length-nat y
  by (rule length-nat-mono) arith

lemma length-nat-pos [simp,intro!]: 0 < x ==> 0 < length-nat x
  by (simp add: length-nat-non0)

lemma length-int-mono-gt0: [| 0 ≤ x ; x ≤ y |] ==> length-int x ≤ length-int y
  by (cases x = 0) (simp-all add: length-int-gt0 nat-le-eq-zle)

lemma length-int-mono-lt0: [| x ≤ y ; y ≤ 0 |] ==> length-int y ≤ length-int x
  by (cases y = 0) (simp-all add: length-int-lt0)

lemmas [simp] = length-nat-non0

lemma nat-to-bv (number-of Int.Pls) = []
  by simp

consts
  fast-bv-to-nat-helper :: [bit list, int] => int
primrec
  fast-bv-to-nat-Nil: fast-bv-to-nat-helper [] k = k
  fast-bv-to-nat-Cons: fast-bv-to-nat-helper (b#bs) k =
    fast-bv-to-nat-helper bs ((bit-case Int.Bit0 Int.Bit1 b) k)

lemma fast-bv-to-nat-Cons0: fast-bv-to-nat-helper (0#bs) bin =
    fast-bv-to-nat-helper bs (Int.Bit0 bin)
  by simp

lemma fast-bv-to-nat-Cons1: fast-bv-to-nat-helper (1#bs) bin =
    fast-bv-to-nat-helper bs (Int.Bit1 bin)
  by simp

lemma fast-bv-to-nat-def:
  bv-to-nat bs == number-of (fast-bv-to-nat-helper bs Int.Pls)
proof (simp add: bv-to-nat-def)
  have ∀ bin. ¬ (neg (number-of bin :: int)) ⟶ (foldl (%bn b. 2 * bn + bitval b)
(number-of bin) bs) = number-of (fast-bv-to-nat-helper bs bin)
    apply (induct bs,simp)
    apply (case-tac a,simp-all)
    done
  thus foldl (λbn b. 2 * bn + bitval b) 0 bs ≡ number-of (fast-bv-to-nat-helper bs
Int.Pls)
```

by (*simp del*: *nat-numeral-0-eq-0 add*: *nat-numeral-0-eq-0* [*symmetric*])
qed

declare *fast-bv-to-nat-Cons* [*simp del*]
declare *fast-bv-to-nat-Cons0* [*simp*]
declare *fast-bv-to-nat-Cons1* [*simp*]

setup ⟪
(∗*comments containing lcp are the removal of fast-bv-of-nat*∗)
let
fun is-const-bool (*Const*(*True*,-)) = *true*
| *is-const-bool* (*Const*(*False*,-)) = *true*
| *is-const-bool* - = *false*
fun is-const-bit (*Const*(*Word.bit.Zero*,-)) = *true*
| *is-const-bit* (*Const*(*Word.bit.One*,-)) = *true*
| *is-const-bit* - = *false*
fun num-is-usable (*Const*(@{*const-name Int.Pls*},-)) = *true*
| *num-is-usable* (*Const*(@{*const-name Int.Min*},-)) = *false*
| *num-is-usable* (*Const*(@{*const-name Int.Bit0*},-) $ *w*) =
num-is-usable w
| *num-is-usable* (*Const*(@{*const-name Int.Bit1*},-) $ *w*) =
num-is-usable w
| *num-is-usable* - = *false*
fun vec-is-usable (*Const*(*List.list.Nil*,-)) = *true*
| *vec-is-usable* (*Const*(*List.list.Cons*,-) $ *b* $ *bs*) =
vec-is-usable bs andalso is-const-bit b
| *vec-is-usable* - = *false*
(∗*lcp*∗ *val fast1-th* = *PureThy.get-thm thy Word.fast-nat-to-bv-def*∗)
val fast2-th = @{*thm Word.fast-bv-to-nat-def*};
(∗*lcp*∗ *fun f sg thms* (*Const*(*Word.nat-to-bv*,-) $ (*Const*(@{*const-name Int.number-of*},-)
$ *t*)) =
if num-is-usable t
then SOME (*Drule.cterm-instantiate* [(*cterm-of sg* (*Var* ((*w*, *0*), @{*typ int*})),
cterm-of sg t)] *fast1-th*)
else NONE
| *f* - - - = *NONE* ∗)
fun g sg thms (*Const*(*Word.bv-to-nat*,-) $ (*t as* (*Const*(*List.list.Cons*,-) $ - $ -)))
=
if vec-is-usable t then
SOME (*Drule.cterm-instantiate* [(*cterm-of sg* (*Var*((*bs*,*0*),*Type*(*List.list*,[*Type*(*Word.bit*,[])]))),*cterm-of
sg t*)] *fast2-th*)
else NONE
| *g* - - - = *NONE*
(∗*lcp*∗ *val simproc1* = *Simplifier.simproc thy nat-to-bv* [*Word.nat-to-bv* (*number-of
w*)] *f* ∗)
val simproc2 = *Simplifier.simproc* @{*theory*} *bv-to-nat* [*Word.bv-to-nat* (*x* #
xs)] *g*
in
Simplifier.map-simpset (*fn ss* => *ss addsimprocs* [(∗*lcp*∗*simproc1*,∗)*simproc2*])

end⟩⟩

declare *bv-to-nat1* [*simp del*]
declare *bv-to-nat-helper* [*simp del*]

definition
bv-mapzip :: [*bit* => *bit* => *bit*,*bit list*, *bit list*] => *bit list* **where**
bv-mapzip f w1 w2 =
(*let g* = *bv-extend* (*max* (*length w1*) (*length w2*)) **0**
in map (*split f*) (*zip* (*g w1*) (*g w2*)))

lemma *bv-length-bv-mapzip* [*simp*]:
length (*bv-mapzip f w1 w2*) = *max* (*length w1*) (*length w2*)
by (*simp add*: *bv-mapzip-def Let-def split*: *split-max*)

lemma *bv-mapzip-Nil* [*simp*]: *bv-mapzip f* [] [] = []
by (*simp add*: *bv-mapzip-def Let-def*)

lemma *bv-mapzip-Cons* [*simp*]: *length w1* = *length w2* ==>
bv-mapzip f (*x*#*w1*) (*y*#*w2*) = *f x y* # *bv-mapzip f w1 w2*
by (*simp add*: *bv-mapzip-def Let-def*)

end

55 Zorn: Zorn's Lemma

theory *Zorn*
imports *Order-Relation*
begin

definition *chain-subset* :: *'a set set* ⇒ *bool* (*chain*$_{\subseteq}$) **where**
chain$_{\subseteq}$ *C* ≡ ∀ *A*∈*C*.∀ *B*∈*C*. *A* ⊆ *B* ∨ *B* ⊆ *A*

The lemma and section numbers refer to an unpublished article [1].

definition
chain :: *'a set set* => *'a set set set* **where**
chain S = {*F*. *F* ⊆ *S* & *chain*$_{\subseteq}$ *F*}

definition
super :: [*'a set set*,*'a set set*] => *'a set set set* **where**
super S c = {*d*. *d* ∈ *chain S* & *c* ⊂ *d*}

definition
maxchain :: *'a set set* => *'a set set set* **where**
maxchain S = {*c*. *c* ∈ *chain S* & *super S c* = {}}

definition

```
  succ      :: ['a set set,'a set set] => 'a set set where
  succ S c =
    (if c ∉ chain S | c ∈ maxchain S
    then c else SOME c'. c' ∈ super S c)

inductive-set
  TFin :: 'a set set => 'a set set set
  for S :: 'a set set
  where
    succI:         x ∈ TFin S ==> succ S x ∈ TFin S
  | Pow-UnionI:    Y ∈ Pow(TFin S) ==> Union(Y) ∈ TFin S
```

55.1 Mathematical Preamble

```
lemma Union-lemma0:
   (∀x ∈ C. x ⊆ A | B ⊆ x) ==> Union(C) ⊆ A | B ⊆ Union(C)
  by blast
```

This is theorem *increasingD2* of ZF/Zorn.thy

```
lemma Abrial-axiom1: x ⊆ succ S x
  apply (auto simp add: succ-def super-def maxchain-def)
  apply (rule contrapos-np, assumption)
  apply (rule-tac Q=λS. xa ∈ S in someI2, blast+)
  done

lemmas TFin-UnionI = TFin.Pow-UnionI [OF PowI]

lemma TFin-induct:
  assumes H: n ∈ TFin S
  and I: !!x. x ∈ TFin S ==> P x ==> P (succ S x)
    !!Y. Y ⊆ TFin S ==> Ball Y P ==> P(Union Y)
  shows P n using H
  apply (induct rule: TFin.induct [where P=P])
   apply (blast intro: I)+
  done

lemma succ-trans: x ⊆ y ==> x ⊆ succ S y
  apply (erule subset-trans)
  apply (rule Abrial-axiom1)
  done
```

Lemma 1 of section 3.1

```
lemma TFin-linear-lemma1:
     [| n ∈ TFin S;  m ∈ TFin S;
        ∀x ∈ TFin S. x ⊆ m --> x = m | succ S x ⊆ m
      |] ==> n ⊆ m | succ S m ⊆ n
  apply (erule TFin-induct)
   apply (erule-tac [2] Union-lemma0)
  apply (blast del: subsetI intro: succ-trans)
  done
```

Lemma 2 of section 3.2

lemma *TFin-linear-lemma2*:
m ∈ *TFin S* ==> ∀ *n* ∈ *TFin S*. *n* ⊆ *m* −−> *n*=*m* | *succ S n* ⊆ *m*
apply (*erule TFin-induct*)
apply (*rule impI* [*THEN ballI*])

case split using *TFin-linear-lemma1*

apply (*rule-tac n1* = *n* **and** *m1* = *x* **in** *TFin-linear-lemma1* [*THEN disjE*], *assumption*+)
apply (*drule-tac x* = *n* **in** *bspec*, *assumption*)
apply (*blast del*: *subsetI intro*: *succ-trans*, *blast*)

second induction step

apply (*rule impI* [*THEN ballI*])
apply (*rule Union-lemma0* [*THEN disjE*])
apply (*rule-tac* [*3*] *disjI2*)
prefer *2* **apply** *blast*
apply (*rule ballI*)
apply (*rule-tac n1* = *n* **and** *m1* = *x* **in** *TFin-linear-lemma1* [*THEN disjE*], *assumption*+, *auto*)
apply (*blast intro*!: *Abrial-axiom1* [*THEN subsetD*])
done

Re-ordering the premises of Lemma 2

lemma *TFin-subsetD*:
[| *n* ⊆ *m*; *m* ∈ *TFin S*; *n* ∈ *TFin S* |] ==> *n*=*m* | *succ S n* ⊆ *m*
by (*rule TFin-linear-lemma2* [*rule-format*])

Consequences from section 3.3 – Property 3.2, the ordering is total

lemma *TFin-subset-linear*: [| *m* ∈ *TFin S*; *n* ∈ *TFin S*|] ==> *n* ⊆ *m* | *m* ⊆ *n*
apply (*rule disjE*)
apply (*rule TFin-linear-lemma1* [*OF - - TFin-linear-lemma2*])
apply (*assumption*+, *erule disjI2*)
apply (*blast del*: *subsetI*
intro: *subsetI Abrial-axiom1* [*THEN subset-trans*])
done

Lemma 3 of section 3.3

lemma *eq-succ-upper*: [| *n* ∈ *TFin S*; *m* ∈ *TFin S*; *m* = *succ S m* |] ==> *n* ⊆ *m*
apply (*erule TFin-induct*)
apply (*drule TFin-subsetD*)
apply (*assumption*+, *force*, *blast*)
done

Property 3.3 of section 3.3

lemma *equal-succ-Union*: *m* ∈ *TFin S* ==> (*m* = *succ S m*) = (*m* = *Union*(*TFin S*))
apply (*rule iffI*)

apply (*rule Union-upper* [*THEN equalityI*])
apply *assumption*
apply (*rule eq-succ-upper* [*THEN Union-least*], *assumption*+)
apply (*erule ssubst*)
apply (*rule Abrial-axiom1* [*THEN equalityI*])
apply (*blast del*: *subsetI intro*: *subsetI TFin-UnionI TFin.succI*)
done

55.2 Hausdorff's Theorem: Every Set Contains a Maximal Chain.

NB: We assume the partial ordering is $\subseteq$, the subset relation!

lemma *empty-set-mem-chain*: $(\{\} :: 'a\ set\ set) \in chain\ S$
by (*unfold chain-def chain-subset-def*) *auto*

lemma *super-subset-chain*: $super\ S\ c \subseteq chain\ S$
by (*unfold super-def*) *blast*

lemma *maxchain-subset-chain*: $maxchain\ S \subseteq chain\ S$
by (*unfold maxchain-def*) *blast*

lemma *mem-super-Ex*: $c \in chain\ S - maxchain\ S ==> EX\ d.\ d \in super\ S\ c$
by (*unfold super-def maxchain-def*) *auto*

lemma *select-super*:
$c \in chain\ S - maxchain\ S ==> (\epsilon\ c'.\ c' : super\ S\ c) : super\ S\ c$
apply (*erule mem-super-Ex* [*THEN exE*])
apply (*rule someI2* [**where** $Q = \%X.\ X : super\ S\ c$], *auto*)
done

lemma *select-not-equals*:
$c \in chain\ S - maxchain\ S ==> (\epsilon\ c'.\ c' : super\ S\ c) \neq c$
apply (*rule notI*)
apply (*drule select-super*)
apply (*simp add*: *super-def less-le*)
done

lemma *succI3*: $c \in chain\ S - maxchain\ S ==> succ\ S\ c = (\epsilon\ c'.\ c' : super\ S\ c)$
by (*unfold succ-def*) (*blast intro*!: *if-not-P*)

lemma *succ-not-equals*: $c \in chain\ S - maxchain\ S ==> succ\ S\ c \neq c$
apply (*frule succI3*)
apply (*simp* (*no-asm-simp*))
apply (*rule select-not-equals*, *assumption*)
done

lemma *TFin-chain-lemma4*: $c \in TFin\ S ==> (c :: 'a\ set\ set) : chain\ S$
apply (*erule TFin-induct*)
apply (*simp add*: *succ-def select-super* [*THEN super-subset-chain*[*THEN sub-*

setD]])
apply (*unfold chain-def chain-subset-def*)
apply (*rule CollectI*, *safe*)
apply (*drule bspec*, *assumption*)
apply (*rule-tac* [*2*] *m1* = *Xa* **and** *n1* = *X* **in** *TFin-subset-linear* [*THEN disjE*],
best+)
done

theorem *Hausdorff*: ∃ *c*. (*c* :: *'a set set*): *maxchain S*
apply (*rule-tac x* = *Union* (*TFin S*) **in** *exI*)
apply (*rule classical*)
apply (*subgoal-tac succ S* (*Union* (*TFin S*)) = *Union* (*TFin S*))
prefer *2*
apply (*blast intro*!: *TFin-UnionI equal-succ-Union* [*THEN iffD2*, *symmetric*])
apply (*cut-tac subset-refl* [*THEN TFin-UnionI*, *THEN TFin-chain-lemma4*])
apply (*drule DiffI* [*THEN succ-not-equals*], *blast*+)
done

55.3 Zorn's Lemma: If All Chains Have Upper Bounds Then There Is a Maximal Element

lemma *chain-extend*:
[| *c* ∈ *chain S*; *z* ∈ *S*; ∀ *x* ∈ *c*. *x* ⊆ (*z*:: *'a set*) |] ==> {*z*} *Un c* ∈ *chain S*
by (*unfold chain-def chain-subset-def*) *blast*

lemma *chain-Union-upper*: [| *c* ∈ *chain S*; *x* ∈ *c* |] ==> *x* ⊆ *Union*(*c*)
by *auto*

lemma *chain-ball-Union-upper*: *c* ∈ *chain S* ==> ∀ *x* ∈ *c*. *x* ⊆ *Union*(*c*)
by *auto*

lemma *maxchain-Zorn*:
[| *c* ∈ *maxchain S*; *u* ∈ *S*; *Union*(*c*) ⊆ *u* |] ==> *Union*(*c*) = *u*
apply (*rule ccontr*)
apply (*simp add*: *maxchain-def*)
apply (*erule conjE*)
apply (*subgoal-tac* ({*u*} *Un c*) ∈ *super S c*)
apply *simp*
apply (*unfold super-def less-le*)
apply (*blast intro*: *chain-extend dest*: *chain-Union-upper*)
done

theorem *Zorn-Lemma*:
∀ *c* ∈ *chain S*. *Union*(*c*): *S* ==> ∃ *y* ∈ *S*. ∀ *z* ∈ *S*. *y* ⊆ *z* --> *y* = *z*
apply (*cut-tac Hausdorff maxchain-subset-chain*)
apply (*erule exE*)
apply (*drule subsetD*, *assumption*)
apply (*drule bspec*, *assumption*)
apply (*rule-tac x* = *Union*(*c*) **in** *bexI*)

apply (*rule ballI*, *rule impI*)
apply (*blast dest*!: *maxchain-Zorn*, *assumption*)
done

55.4 Alternative version of Zorn's Lemma

lemma *Zorn-Lemma2*:
$\forall c \in chain\ S.\ \exists y \in S.\ \forall x \in c.\ x \subseteq y$
$\Longrightarrow \exists y \in S.\ \forall x \in S.\ (y :: 'a\ set) \subseteq x \longrightarrow y = x$
apply (*cut-tac Hausdorff maxchain-subset-chain*)
apply (*erule exE*)
apply (*drule subsetD*, *assumption*)
apply (*drule bspec*, *assumption*, *erule bexE*)
apply (*rule-tac x = y* **in** *bexI*)
prefer *2* **apply** *assumption*
apply *clarify*
apply (*rule ccontr*)
apply (*frule-tac z = x* **in** *chain-extend*)
apply (*assumption*, *blast*)
apply (*unfold maxchain-def super-def less-le*)
apply (*blast elim*!: *equalityCE*)
done

Various other lemmas

lemma *chainD*: $[|\ c \in chain\ S;\ x \in c;\ y \in c\ |] \Longrightarrow x \subseteq y \mid y \subseteq x$
by (*unfold chain-def chain-subset-def*) *blast*

lemma *chainD2*: $!!(c :: 'a\ set\ set).\ c \in chain\ S \Longrightarrow c \subseteq S$
by (*unfold chain-def*) *blast*

definition *Chain* :: $('a*'a)set \Rightarrow 'a\ set\ set$ **where**
$Chain\ r \equiv \{A.\ \forall a{\in}A.\forall b{\in}A.\ (a,b) : r \vee (b,a) \in r\}$

lemma *mono-Chain*: $r \subseteq s \Longrightarrow Chain\ r \subseteq Chain\ s$
unfolding *Chain-def* **by** *blast*

Zorn's lemma for partial orders:

lemma *Zorns-po-lemma*:
assumes *po*: *Partial-order r* **and** *u*: $\forall C{\in}Chain\ r.\ \exists u{\in}Field\ r.\ \forall a{\in}C.\ (a,u):r$
shows $\exists m{\in}Field\ r.\ \forall a{\in}Field\ r.\ (m,a):r \longrightarrow a{=}m$
proof−
have *Preorder r* **using** *po* **by**(*simp add*:*partial-order-on-def*)
— Mirror r in the set of subsets below (wrt r) elements of A
let *?B* = %*x*. *r*^−*1* `` {*x*} **let** *?S* = *?B* ` *Field r*
have $\forall C \in chain\ ?S.\ EX\ U{:}?S.\ ALL\ A{:}C.\ A{\subseteq}U$
proof (*auto simp*:*chain-def chain-subset-def*)
fix *C* **assume** *1*: $C \subseteq ?S$ **and** *2*: $\forall A{\in}C.\forall B{\in}C.\ A{\subseteq}B \mid B{\subseteq}A$
let *?A* = $\{x{\in}Field\ r.\ \exists M{\in}C.\ M = ?B\ x\}$

```
    have C = ?B ‘ ?A using 1 by(auto simp: image-def)
    have ?A∈Chain r
    proof (simp add:Chain-def, intro allI impI, elim conjE)
      fix a b
      assume a ∈ Field r ?B a ∈ C b ∈ Field r ?B b ∈ C
      hence ?B a ⊆ ?B b ∨ ?B b ⊆ ?B a using 2 by auto
      thus (a, b) ∈ r ∨ (b, a) ∈ r using ‹Preorder r› ‹a:Field r› ‹b:Field r›
        by(simp add:subset-Image1-Image1-iff)
    qed
    then obtain u where uA: u:Field r ∀ a∈?A. (a,u) : r using u by auto
    have ∀ A∈C. A ⊆ r^−1 “ {u} (is ?P u)
    proof auto
      fix a B assume aB: B:C a:B
      with 1 obtain x where x:Field r B = r^−1 “ {x} by auto
      thus (a,u) : r using uA aB ‹Preorder r›
        by (auto simp add: preorder-on-def refl-def) (metis transD)
    qed
    thus EX u:Field r. ?P u using ‹u:Field r› by blast
  qed
  from Zorn-Lemma2[OF this]
  obtain m B where m:Field r B = r^−1 “ {m}
    ∀ x∈Field r. B ⊆ r^−1 “ {x} ⟶ B = r^−1 “ {x}
    by auto
  hence ∀ a∈Field r. (m, a) ∈ r ⟶ a = m using po ‹Preorder r› ‹m:Field r›
    by(auto simp:subset-Image1-Image1-iff Partial-order-eq-Image1-Image1-iff)
  thus ?thesis using ‹m:Field r› by blast
qed


definition init-seg-of :: (('a*'a)set * ('a*'a)set)set where
init-seg-of == {(r,s). r ⊆ s ∧ (∀ a b c. (a,b):s ∧ (b,c):r ⟶ (a,b):r)}

abbreviation initialSegmentOf :: ('a*'a)set ⇒ ('a*'a)set ⇒ bool
          (infix initial'-segment'-of 55) where
r initial-segment-of s == (r,s):init-seg-of

lemma refl-init-seg-of[simp]: r initial-segment-of r
by(simp add:init-seg-of-def)

lemma trans-init-seg-of:
  r initial-segment-of s ⟹ s initial-segment-of t ⟹ r initial-segment-of t
by(simp (no-asm-use) add: init-seg-of-def)
  (metis Domain-iff UnCI Un-absorb2 subset-trans)

lemma antisym-init-seg-of:
  r initial-segment-of s ⟹ s initial-segment-of r ⟹ r=s
by(auto simp:init-seg-of-def)

lemma Chain-init-seg-of-Union:
```

$R \in \mathit{Chain}\ \mathit{init\text{-}seg\text{-}of} \Longrightarrow r{\in}R \Longrightarrow r\ \mathit{initial\text{-}segment\text{-}of}\ \bigcup R$
by(*auto simp add*:*init-seg-of-def Chain-def Ball-def*) *blast*

lemma *chain-subset-trans-Union*:
$\mathit{chain}_{\subseteq}\ R \Longrightarrow \forall r{\in}R.\ \mathit{trans}\ r \Longrightarrow \mathit{trans}(\bigcup R)$
apply(*auto simp add*:*chain-subset-def*)
apply(*simp* (*no-asm-use*) *add*:*trans-def*)
apply (*metis subsetD*)
done

lemma *chain-subset-antisym-Union*:
$\mathit{chain}_{\subseteq}\ R \Longrightarrow \forall r{\in}R.\ \mathit{antisym}\ r \Longrightarrow \mathit{antisym}(\bigcup R)$
apply(*auto simp add*:*chain-subset-def antisym-def*)
apply (*metis subsetD*)
done

lemma *chain-subset-Total-Union*:
assumes $\mathit{chain}_{\subseteq}\ R\ \forall r{\in}R.\ \mathit{Total}\ r$
shows $\mathit{Total}\ (\bigcup R)$
proof (*simp add*: *total-on-def Ball-def*, *auto del*:*disjCI*)
fix *r s a b* **assume** *A*: *r*:*R s*:*R a*:*Field r b*:*Field s* $a{\neq}b$
from ‹$\mathit{chain}_{\subseteq}\ R$› ‹*r*:*R*› ‹*s*:*R*› **have** $r{\subseteq}s \vee s{\subseteq}r$
by(*simp add*:*chain-subset-def*)
thus $(\exists r{\in}R.\ (a,b) \in r) \vee (\exists r{\in}R.\ (b,a) \in r)$
proof
assume $r{\subseteq}s$ **hence** (*a*,*b*):*s* $\vee$ (*b*,*a*):*s* **using** *assms*(*2*) *A*
by(*simp add*:*total-on-def*)(*metis mono-Field subsetD*)
thus *?thesis* **using** ‹*s*:*R*› **by** *blast*
next
assume $s{\subseteq}r$ **hence** (*a*,*b*):*r* $\vee$ (*b*,*a*):*r* **using** *assms*(*2*) *A*
by(*simp add*:*total-on-def*)(*metis mono-Field subsetD*)
thus *?thesis* **using** ‹*r*:*R*› **by** *blast*
qed
qed

lemma *wf-Union-wf-init-segs*:
assumes $R \in \mathit{Chain}\ \mathit{init\text{-}seg\text{-}of}$ **and** $\forall r{\in}R.\ \mathit{wf}\ r$ **shows** $\mathit{wf}(\bigcup R)$
proof(*simp add*:*wf-iff-no-infinite-down-chain*, *rule ccontr*, *auto*)
fix *f* **assume** *1*: $\forall i.\ \exists r{\in}R.\ (f(\mathit{Suc}\ i), f\ i) \in r$
then obtain *r* **where** *r*:*R* **and** $(f(\mathit{Suc}\ 0), f\ 0) : r$ **by** *auto*
{ fix *i* **have** $(f(\mathit{Suc}\ i), f\ i) \in r$
proof(*induct i*)
case *0* **show** *?case* **by** *fact*
next
case (*Suc i*)
moreover obtain *s* **where** $s{\in}R$ **and** $(f(\mathit{Suc}(\mathit{Suc}\ i)), f(\mathit{Suc}\ i)) \in s$
using *1* **by** *auto*
moreover hence *s initial-segment-of r* $\vee$ *r initial-segment-of s*
using *assms*(*1*) ‹*r*:*R*› **by**(*simp add*: *Chain-def*)

ultimately show *?case* **by**(*simp add*:*init-seg-of-def*) *blast*
qed
}
thus *False* **using** *assms*(*2*) ‹*r*:*R*›
by(*simp add*:*wf-iff-no-infinite-down-chain*) *blast*
qed

lemma *Chain-inits-DiffI*:
$R \in$ *Chain init-seg-of* $\Longrightarrow \{r - s \mid r.\ r \in R\} \in$ *Chain init-seg-of*
apply(*auto simp*:*Chain-def init-seg-of-def*)
apply (*metis subsetD*)
apply (*metis subsetD*)
done

theorem *well-ordering*: $\exists r::('a*'a)set.$ *Well-order* $r \wedge$ *Field* $r =$ *UNIV*
proof−
— The initial segment relation on well-orders:
let *?WO* $= \{r::('a*'a)set.$ *Well-order* $r\}$
def $I \equiv$ *init-seg-of* $\cap$ *?WO* $\times$ *?WO*
have *I-init*: $I \subseteq$ *init-seg-of* **by**(*auto simp*:*I-def*)
hence *subch*: $!!R.\ R :$ *Chain* $I \Longrightarrow$ *chain*$_{\subseteq}$ R
by(*auto simp*:*init-seg-of-def chain-subset-def Chain-def*)
have *Chain-wo*: $!!R\ r.\ R \in$ *Chain* $I \Longrightarrow r \in R \Longrightarrow$ *Well-order* r
by(*simp add*:*Chain-def I-def*) *blast*
have *FI*: *Field* $I =$ *?WO* **by**(*auto simp add*:*I-def init-seg-of-def Field-def*)
hence *0*: *Partial-order* I
by(*auto simp*: *partial-order-on-def preorder-on-def antisym-def antisym-init-seg-of refl-def trans-def I-def elim*!: *trans-init-seg-of*)
— I-chains have upper bounds in ?WO wrt I: their Union
{ fix R **assume** $R \in$ *Chain* I
hence *Ris*: $R \in$ *Chain init-seg-of* **using** *mono-Chain*[*OF I-init*] **by** *blast*
have *subch*: *chain*$_{\subseteq}$ R **using** ‹R : *Chain* I› *I-init*
by(*auto simp*:*init-seg-of-def chain-subset-def Chain-def*)
have $\forall\, r{\in}R.$ *Refl* $r\ \forall\, r{\in}R.$ *trans* $r\ \forall\, r{\in}R.$ *antisym* $r\ \forall\, r{\in}R.$ *Total* r
$\forall\, r{\in}R.$ *wf*$(r{-}Id)$
using *Chain-wo*[*OF* ‹$R \in$ *Chain* I›] **by**(*simp-all add*:*order-on-defs*)
have *Refl* $(\bigcup R)$ **using** ‹$\forall\, r{\in}R.$ *Refl* r› **by**(*auto simp*:*refl-def*)
moreover have *trans* $(\bigcup R)$
by(*rule chain-subset-trans-Union*[*OF subch* ‹$\forall\, r{\in}R.$ *trans* r›])
moreover have *antisym*$(\bigcup R)$
by(*rule chain-subset-antisym-Union*[*OF subch* ‹$\forall\, r{\in}R.$ *antisym* r›])
moreover have *Total* $(\bigcup R)$
by(*rule chain-subset-Total-Union*[*OF subch* ‹$\forall\, r{\in}R.$ *Total* r›])
moreover have *wf*$((\bigcup R){-}Id)$
proof−
have $(\bigcup R){-}Id = \bigcup\{r{-}Id | r.\ r \in R\}$ **by** *blast*
with ‹$\forall\, r{\in}R.$ *wf*$(r{-}Id)$› *wf-Union-wf-init-segs*[*OF Chain-inits-DiffI*[*OF Ris*]]
show *?thesis* **by** (*simp* (*no-asm-simp*)) *blast*
qed

```
    ultimately have Well-order (⋃ R) by(simp add:order-on-defs)
    moreover have ∀ r ∈ R. r initial-segment-of ⋃ R using Ris
      by(simp add: Chain-init-seg-of-Union)
    ultimately have ⋃ R : ?WO ∧ (∀ r∈R. (r,⋃ R) : I)
      using mono-Chain[OF I-init] ‹R ∈ Chain I›
      by(simp (no-asm) add:I-def del:Field-Union)(metis Chain-wo subsetD)
  }
  hence 1: ∀ R ∈ Chain I. ∃ u∈Field I. ∀ r∈R. (r,u) : I by (subst FI) blast
— Zorn's Lemma yields a maximal well-order m:
  then obtain m::('a*'a)set where Well-order m and
    max: ∀ r. Well-order r ∧ (m,r):I ⟶ r=m
    using Zorns-po-lemma[OF 0 1] by (auto simp:FI)
— Now show by contradiction that m covers the whole type:
  { fix x::'a assume x ∉ Field m
— We assume that x is not covered and extend m at the top with x
    have m ≠ {}
    proof
      assume m={}
      moreover have Well-order {(x,x)}
      by(simp add:order-on-defs refl-def trans-def antisym-def total-on-def Field-def
Domain-def Range-def)
      ultimately show False using max
        by (auto simp:I-def init-seg-of-def simp del:Field-insert)
    qed
    hence Field m ≠ {} by(auto simp:Field-def)
    moreover have wf(m−Id) using ‹Well-order m›
      by(simp add:well-order-on-def)
— The extension of m by x:
    let ?s = {(a,x)|a. a : Field m} let ?m = insert (x,x) m Un ?s
    have Fm: Field ?m = insert x (Field m)
      apply(simp add:Field-insert Field-Un)
      unfolding Field-def by auto
    have Refl m trans m antisym m Total m wf(m−Id)
      using ‹Well-order m› by(simp-all add:order-on-defs)
— We show that the extension is a well-order
    have Refl ?m using ‹Refl m› Fm by(auto simp:refl-def)
    moreover have trans ?m using ‹trans m› ‹x ∉ Field m›
      unfolding trans-def Field-def Domain-def Range-def by blast
    moreover have antisym ?m using ‹antisym m› ‹x ∉ Field m›
      unfolding antisym-def Field-def Domain-def Range-def by blast
    moreover have Total ?m using ‹Total m› Fm by(auto simp: total-on-def)
    moreover have wf(?m−Id)
    proof−
      have wf ?s using ‹x ∉ Field m›
        by(auto simp add:wf-eq-minimal Field-def Domain-def Range-def) metis
      thus ?thesis using ‹wf(m−Id)› ‹x ∉ Field m›
        wf-subset[OF ‹wf ?s› Diff-subset]
        by (fastsimp intro!: wf-Un simp add: Un-Diff Field-def)
    qed
```

ultimately have *Well-order ?m* **by**(*simp add*:*order-on-defs*)
— We show that the extension is above m
moreover hence (*m*,*?m*) : *I* **using** ‹*Well-order m*› ‹*x* ∉ *Field m*›
by(*fastsimp simp*:*I-def init-seg-of-def Field-def Domain-def Range-def*)
ultimately
— This contradicts maximality of m:
have *False* **using** *max* ‹*x* ∉ *Field m*› **unfolding** *Field-def* **by** *blast*
}
hence *Field m* = *UNIV* **by** *auto*
moreover with ‹*Well-order m*› **have** *Well-order m* **by** *simp*
ultimately show *?thesis* **by** *blast*
qed

corollary *well-order-on*: ∃ *r*::('*a*∗'*a*)*set*. *well-order-on A r*
proof −
obtain *r*::('*a*∗'*a*)*set* **where** *wo*: *Well-order r* **and** *univ*: *Field r* = *UNIV*
using *well-ordering*[**where** '*a* = '*a*] **by** *blast*
let *?r* = {(*x*,*y*). *x*:*A* & *y*:*A* & (*x*,*y*):*r*}
have *1*: *Field ?r* = *A* **using** *wo univ*
by(*fastsimp simp*: *Field-def Domain-def Range-def order-on-defs refl-def*)
have *Refl r trans r antisym r Total r wf*(*r*−*Id*)
using ‹*Well-order r*› **by**(*simp-all add*:*order-on-defs*)
have *Refl ?r* **using** ‹*Refl r*› **by**(*auto simp*:*refl-def 1 univ*)
moreover have *trans ?r* **using** ‹*trans r*›
unfolding *trans-def* **by** *blast*
moreover have *antisym ?r* **using** ‹*antisym r*›
unfolding *antisym-def* **by** *blast*
moreover have *Total ?r* **using** ‹*Total r*› **by**(*simp add*:*total-on-def 1 univ*)
moreover have *wf*(*?r* − *Id*) **by**(*rule wf-subset*[*OF* ‹*wf*(*r*−*Id*)›]) *blast*
ultimately have *Well-order ?r* **by**(*simp add*:*order-on-defs*)
with *1* **show** *?thesis* **by** *metis*
qed

end

56 List-Prefix: List prefixes and postfixes

theory *List-Prefix*
imports *List*
begin

56.1 Prefix order on lists

instantiation *list* :: (*type*) *order*
begin

definition
prefix-def [*code func del*]: *xs* ≤ *ys* = (∃ *zs*. *ys* = *xs* @ *zs*)

definition
strict-prefix-def [*code func del*]: $xs < ys = (xs \leq ys \wedge xs \neq (ys::'a\ list))$

instance
by *intro-classes* (*auto simp add*: *prefix-def strict-prefix-def*)

end

lemma *prefixI* [*intro?*]: $ys = xs\ @\ zs \Longrightarrow xs \leq ys$
unfolding *prefix-def* **by** *blast*

lemma *prefixE* [*elim?*]:
assumes $xs \leq ys$
obtains *zs* **where** $ys = xs\ @\ zs$
using *assms* **unfolding** *prefix-def* **by** *blast*

lemma *strict-prefixI'* [*intro?*]: $ys = xs\ @\ z\ \#\ zs \Longrightarrow xs < ys$
unfolding *strict-prefix-def prefix-def* **by** *blast*

lemma *strict-prefixE'* [*elim?*]:
assumes $xs < ys$
obtains *z zs* **where** $ys = xs\ @\ z\ \#\ zs$
proof −
from ‹$xs < ys$› **obtain** *us* **where** $ys = xs\ @\ us$ **and** $xs \neq ys$
unfolding *strict-prefix-def prefix-def* **by** *blast*
with *that* **show** *?thesis* **by** (*auto simp add*: *neq-Nil-conv*)
qed

lemma *strict-prefixI* [*intro?*]: $xs \leq ys \Longrightarrow xs \neq ys \Longrightarrow xs < (ys::'a\ list)$
unfolding *strict-prefix-def* **by** *blast*

lemma *strict-prefixE* [*elim?*]:
fixes *xs ys* :: *'a list*
assumes $xs < ys$
obtains $xs \leq ys$ **and** $xs \neq ys$
using *assms* **unfolding** *strict-prefix-def* **by** *blast*

56.2 Basic properties of prefixes

theorem *Nil-prefix* [*iff*]: $[] \leq xs$
by (*simp add*: *prefix-def*)

theorem *prefix-Nil* [*simp*]: $(xs \leq []) = (xs = [])$
by (*induct xs*) (*simp-all add*: *prefix-def*)

lemma *prefix-snoc* [*simp*]: $(xs \leq ys\ @\ [y]) = (xs = ys\ @\ [y] \vee xs \leq ys)$
proof
assume $xs \leq ys\ @\ [y]$

then obtain *zs* **where** *zs*: $ys \mathbin{@} [y] = xs \mathbin{@} zs$ **..**
show $xs = ys \mathbin{@} [y] \vee xs \leq ys$
by (*metis append-Nil2 butlast-append butlast-snoc prefixI zs*)
next
assume $xs = ys \mathbin{@} [y] \vee xs \leq ys$
then show $xs \leq ys \mathbin{@} [y]$
by (*metis order-eq-iff strict-prefixE strict-prefixI' xt1*(7))
qed

lemma *Cons-prefix-Cons* [*simp*]: $(x \# xs \leq y \# ys) = (x = y \wedge xs \leq ys)$
by (*auto simp add*: *prefix-def*)

lemma *same-prefix-prefix* [*simp*]: $(xs \mathbin{@} ys \leq xs \mathbin{@} zs) = (ys \leq zs)$
by (*induct xs*) *simp-all*

lemma *same-prefix-nil* [*iff*]: $(xs \mathbin{@} ys \leq xs) = (ys = [])$
by (*metis append-Nil2 append-self-conv order-eq-iff prefixI*)

lemma *prefix-prefix* [*simp*]: $xs \leq ys \mathrel{==>} xs \leq ys \mathbin{@} zs$
by (*metis order-le-less-trans prefixI strict-prefixE strict-prefixI*)

lemma *append-prefixD*: $xs \mathbin{@} ys \leq zs \Longrightarrow xs \leq zs$
by (*auto simp add*: *prefix-def*)

theorem *prefix-Cons*: $(xs \leq y \# ys) = (xs = [] \vee (\exists zs.\ xs = y \# zs \wedge zs \leq ys))$
by (*cases xs*) (*auto simp add*: *prefix-def*)

theorem *prefix-append*:
$(xs \leq ys \mathbin{@} zs) = (xs \leq ys \vee (\exists us.\ xs = ys \mathbin{@} us \wedge us \leq zs))$
apply (*induct zs rule*: *rev-induct*)
apply *force*
apply (*simp del*: *append-assoc add*: *append-assoc* [*symmetric*])
apply (*metis append-eq-appendI*)
done

lemma *append-one-prefix*:
$xs \leq ys \mathrel{==>} length\ xs < length\ ys \mathrel{==>} xs \mathbin{@} [ys\ !\ length\ xs] \leq ys$
unfolding *prefix-def*
by (*metis Cons-eq-appendI append-eq-appendI append-eq-conv-conj*
eq-Nil-appendI nth-drop')

theorem *prefix-length-le*: $xs \leq ys \mathrel{==>} length\ xs \leq length\ ys$
by (*auto simp add*: *prefix-def*)

lemma *prefix-same-cases*:
$(xs_1 :: 'a\ list) \leq ys \Longrightarrow xs_2 \leq ys \Longrightarrow xs_1 \leq xs_2 \vee xs_2 \leq xs_1$
unfolding *prefix-def* **by** (*metis append-eq-append-conv2*)

lemma *set-mono-prefix*: $xs \leq ys \Longrightarrow set\ xs \subseteq set\ ys$

by (*auto simp add*: *prefix-def*)

lemma *take-is-prefix*: *take n xs* ≤ *xs*
unfolding *prefix-def* **by** (*metis append-take-drop-id*)

lemma *map-prefixI*: *xs* ≤ *ys* ⟹ *map f xs* ≤ *map f ys*
by (*auto simp*: *prefix-def*)

lemma *prefix-length-less*: *xs* < *ys* ⟹ *length xs* < *length ys*
by (*auto simp*: *strict-prefix-def prefix-def*)

lemma *strict-prefix-simps* [*simp*]:
xs < [] = *False*
[] < (*x* # *xs*) = *True*
(*x* # *xs*) < (*y* # *ys*) = (*x* = *y* ∧ *xs* < *ys*)
by (*simp-all add*: *strict-prefix-def cong*: *conj-cong*)

lemma *take-strict-prefix*: *xs* < *ys* ⟹ *take n xs* < *ys*
apply (*induct n arbitrary*: *xs ys*)
apply (*case-tac ys*, *simp-all*)[*1*]
apply (*metis order-less-trans strict-prefixI take-is-prefix*)
done

lemma *not-prefix-cases*:
assumes *pfx*: ¬ *ps* ≤ *ls*
obtains
(*c1*) *ps* ≠ [] **and** *ls* = []
| (*c2*) *a as x xs* **where** *ps* = *a*#*as* **and** *ls* = *x*#*xs* **and** *x* = *a* **and** ¬ *as* ≤ *xs*
| (*c3*) *a as x xs* **where** *ps* = *a*#*as* **and** *ls* = *x*#*xs* **and** *x* ≠ *a*
proof (*cases ps*)
case *Nil* **then show** *?thesis* **using** *pfx* **by** *simp*
next
case (*Cons a as*)
note *c* = ‹*ps* = *a*#*as*›
show *?thesis*
proof (*cases ls*)
case *Nil* **then show** *?thesis* **by** (*metis append-Nil2 pfx c1 same-prefix-nil*)
next
case (*Cons x xs*)
show *?thesis*
proof (*cases x* = *a*)
case *True*
have ¬ *as* ≤ *xs* **using** *pfx c Cons True* **by** *simp*
with *c Cons True* **show** *?thesis* **by** (*rule c2*)
next
case *False*
with *c Cons* **show** *?thesis* **by** (*rule c3*)
qed
qed

qed

lemma *not-prefix-induct* [*consumes 1*, *case-names Nil Neq Eq*]:
  assumes *np*: $\neg$ *ps* $\leq$ *ls*
    and *base*: $\bigwedge x\ xs.\ P\ (x\#xs)\ []$
    and *r1*: $\bigwedge x\ xs\ y\ ys.\ x \neq y \Longrightarrow P\ (x\#xs)\ (y\#ys)$
    and *r2*: $\bigwedge x\ xs\ y\ ys.\ [\![\ x = y;\ \neg\ xs \leq ys;\ P\ xs\ ys\ ]\!] \Longrightarrow P\ (x\#xs)\ (y\#ys)$
  shows *P ps ls* **using** *np*
proof (*induct ls arbitrary*: *ps*)
  case *Nil* **then show** *?case*
    by (*auto simp*: *neq-Nil-conv elim*!: *not-prefix-cases intro*!: *base*)
next
  case (*Cons y ys*)
  then have *npfx*: $\neg$ *ps* $\leq$ (*y* # *ys*) **by** *simp*
  then obtain *x xs* **where** *pv*: *ps* = *x* # *xs*
    by (*rule not-prefix-cases*) *auto*
  show *?case* **by** (*metis Cons.hyps Cons-prefix-Cons npfx pv r1 r2*)
qed

56.3 Parallel lists

definition
  parallel :: *'a list* => *'a list* => *bool* (**infixl** $\|$ *50*) **where**
  (*xs* $\|$ *ys*) = ($\neg$ *xs* $\leq$ *ys* $\wedge$ $\neg$ *ys* $\leq$ *xs*)

lemma *parallelI* [*intro*]: $\neg$ *xs* $\leq$ *ys* ==> $\neg$ *ys* $\leq$ *xs* ==> *xs* $\|$ *ys*
  unfolding *parallel-def* **by** *blast*

lemma *parallelE* [*elim*]:
  assumes *xs* $\|$ *ys*
  obtains $\neg$ *xs* $\leq$ *ys* $\wedge$ $\neg$ *ys* $\leq$ *xs*
  using *assms* **unfolding** *parallel-def* **by** *blast*

theorem *prefix-cases*:
  obtains *xs* $\leq$ *ys* | *ys* < *xs* | *xs* $\|$ *ys*
  unfolding *parallel-def strict-prefix-def* **by** *blast*

theorem *parallel-decomp*:
  xs $\|$ *ys* ==> $\exists$ *as b bs c cs. b* $\neq$ *c* $\wedge$ *xs* = *as* @ *b* # *bs* $\wedge$ *ys* = *as* @ *c* # *cs*
proof (*induct xs rule*: *rev-induct*)
  case *Nil*
  then have *False* **by** *auto*
  then show *?case* **..**
next
  case (*snoc x xs*)
  show *?case*
  proof (*rule prefix-cases*)
    assume *le*: *xs* $\leq$ *ys*
    then obtain *ys'* **where** *ys*: *ys* = *xs* @ *ys'* **..**

```
    show ?thesis
    proof (cases ys')
      assume ys' = []
      then show ?thesis by (metis append-Nil2 parallelE prefixI snoc.prems ys)
    next
      fix c cs assume ys': ys' = c # cs
      then show ?thesis
        by (metis Cons-eq-appendI eq-Nil-appendI parallelE prefixI
          same-prefix-prefix snoc.prems ys)
    qed
  next
    assume ys < xs then have ys ≤ xs @ [x] by (simp add: strict-prefix-def)
    with snoc have False by blast
    then show ?thesis ..
  next
    assume xs ∥ ys
    with snoc obtain as b bs c cs where neq: (b::'a) ≠ c
      and xs: xs = as @ b # bs and ys: ys = as @ c # cs
      by blast
    from xs have xs @ [x] = as @ b # (bs @ [x]) by simp
    with neq ys show ?thesis by blast
  qed
qed

lemma parallel-append: a ∥ b ⟹ a @ c ∥ b @ d
  apply (rule parallelI)
    apply (erule parallelE, erule conjE,
      induct rule: not-prefix-induct, simp+)+
  done

lemma parallel-appendI: xs ∥ ys ⟹ x = xs @ xs' ⟹ y = ys @ ys' ⟹ x ∥ y
  by (simp add: parallel-append)

lemma parallel-commute: a ∥ b ⟷ b ∥ a
  unfolding parallel-def by auto
```

56.4 Postfix order on lists

```
definition
  postfix :: 'a list => 'a list => bool  ((-/ >>= -) [51, 50] 50) where
  (xs >>= ys) = (∃zs. xs = zs @ ys)

lemma postfixI [intro?]: xs = zs @ ys ==> xs >>= ys
  unfolding postfix-def by blast

lemma postfixE [elim?]:
  assumes xs >>= ys
  obtains zs where xs = zs @ ys
  using assms unfolding postfix-def by blast
```

lemma *postfix-refl* [*iff*]: *xs >>= xs*
  by (*auto simp add*: *postfix-def*)
lemma *postfix-trans*: ⟦*xs >>= ys*; *ys >>= zs*⟧ ⟹ *xs >>= zs*
  by (*auto simp add*: *postfix-def*)
lemma *postfix-antisym*: ⟦*xs >>= ys*; *ys >>= xs*⟧ ⟹ *xs = ys*
  by (*auto simp add*: *postfix-def*)

lemma *Nil-postfix* [*iff*]: *xs >>= []*
  by (*simp add*: *postfix-def*)
lemma *postfix-Nil* [*simp*]: (*[] >>= xs*) = (*xs = []*)
  by (*auto simp add*: *postfix-def*)

lemma *postfix-ConsI*: *xs >>= ys* ⟹ *x#xs >>= ys*
  by (*auto simp add*: *postfix-def*)
lemma *postfix-ConsD*: *xs >>= y#ys* ⟹ *xs >>= ys*
  by (*auto simp add*: *postfix-def*)

lemma *postfix-appendI*: *xs >>= ys* ⟹ *zs @ xs >>= ys*
  by (*auto simp add*: *postfix-def*)
lemma *postfix-appendD*: *xs >>= zs @ ys* ⟹ *xs >>= ys*
  by (*auto simp add*: *postfix-def*)

lemma *postfix-is-subset*: *xs >>= ys ==> set ys* ⊆ *set xs*
proof −
  assume *xs >>= ys*
  then obtain *zs* **where** *xs = zs @ ys* **..**
  then show *?thesis* **by** (*induct zs*) *auto*
qed

lemma *postfix-ConsD2*: *x#xs >>= y#ys ==> xs >>= ys*
proof −
  assume *x#xs >>= y#ys*
  then obtain *zs* **where** *x#xs = zs @ y#ys* **..**
  then show *?thesis*
    by (*induct zs*) (*auto intro*!: *postfix-appendI postfix-ConsI*)
qed

lemma *postfix-to-prefix*: *xs >>= ys* ⟷ *rev ys* ≤ *rev xs*
proof
  assume *xs >>= ys*
  then obtain *zs* **where** *xs = zs @ ys* **..**
  then have *rev xs = rev ys @ rev zs* **by** *simp*
  then show *rev ys <= rev xs* **..**
next
  assume *rev ys <= rev xs*
  then obtain *zs* **where** *rev xs = rev ys @ zs* **..**
  then have *rev* (*rev xs*) = *rev zs @ rev* (*rev ys*) **by** *simp*
  then have *xs = rev zs @ ys* **by** *simp*

then show *xs >>= ys* **..**
qed

lemma *distinct-postfix*: *distinct xs ⟹ xs >>= ys ⟹ distinct ys*
by (*clarsimp elim*!: *postfixE*)

lemma *postfix-map*: *xs >>= ys ⟹ map f xs >>= map f ys*
by (*auto elim*!: *postfixE intro*: *postfixI*)

lemma *postfix-drop*: *as >>= drop n as*
unfolding *postfix-def*
apply (*rule exI* [**where** *x* = *take n as*])
apply *simp*
done

lemma *postfix-take*: *xs >>= ys ⟹ xs = take* (*length xs − length ys*) *xs* @ *ys*
by (*clarsimp elim*!: *postfixE*)

lemma *parallelD1*: *x ∥ y ⟹ ¬ x ≤ y*
by *blast*

lemma *parallelD2*: *x ∥ y ⟹ ¬ y ≤ x*
by *blast*

lemma *parallel-Nil1* [*simp*]: *¬ x ∥ []*
unfolding *parallel-def* **by** *simp*

lemma *parallel-Nil2* [*simp*]: *¬ [] ∥ x*
unfolding *parallel-def* **by** *simp*

lemma *Cons-parallelI1*: *a ≠ b ⟹ a # as ∥ b # bs*
by *auto*

lemma *Cons-parallelI2*: *⟦ a = b; as ∥ bs ⟧ ⟹ a # as ∥ b # bs*
by (*metis Cons-prefix-Cons parallelE parallelI*)

lemma *not-equal-is-parallel*:
assumes *neq*: *xs ≠ ys*
and *len*: *length xs = length ys*
shows *xs ∥ ys*
using *len neq*
proof (*induct rule*: *list-induct2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a as b bs*)
have *ih*: *as ≠ bs ⟹ as ∥ bs* **by** *fact*
show *?case*
proof (*cases a = b*)

case *True*
then have *as* ≠ *bs* **using** *Cons* **by** *simp*
then show *?thesis* **by** (*rule Cons-parallelI2* [*OF True ih*])
next
case *False*
then show *?thesis* **by** (*rule Cons-parallelI1*)
qed
qed

56.5 Executable code

lemma *less-eq-code* [*code func*]:
([]::'*a*::{*eq*, *ord*} *list*) ≤ *xs* ⟷ *True*
(*x*::'*a*::{*eq*, *ord*}) # *xs* ≤ [] ⟷ *False*
(*x*::'*a*::{*eq*, *ord*}) # *xs* ≤ *y* # *ys* ⟷ *x* = *y* ∧ *xs* ≤ *ys*
by *simp-all*

lemma *less-code* [*code func*]:
xs < ([]::'*a*::{*eq*, *ord*} *list*) ⟷ *False*
[] < (*x*::'*a*::{*eq*, *ord*})# *xs* ⟷ *True*
(*x*::'*a*::{*eq*, *ord*}) # *xs* < *y* # *ys* ⟷ *x* = *y* ∧ *xs* < *ys*
unfolding *strict-prefix-def* **by** *auto*

lemmas [*code func*] = *postfix-to-prefix*

end

57 List-lexord: Lexicographic order on lists

theory *List-lexord*
imports *List*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def [*code func del*]: (*xs*::('*a*::*ord*) *list*) < *ys* ⟷ (*xs*, *ys*) ∈ *lexord* {(*u*,*v*). *u* < *v*}

definition
list-le-def [*code func del*]: (*xs*::('*a*::*ord*) *list*) ≤ *ys* ⟷ (*xs* < *ys* ∨ *xs* = *ys*)

instance ..

end

instance *list* :: (*order*) *order*

apply (*intro-classes*, *unfold list-less-def list-le-def*)
apply *safe*
apply (*rule-tac r1* = {(*a*::'*a*,*b*). *a* < *b*} **in** *lexord-irreflexive* [*THEN notE*])
apply *simp*
apply *assumption*
apply (*blast intro*: *lexord-trans transI order-less-trans*)
apply (*rule-tac r1* = {(*a*::'*a*,*b*). *a* < *b*} **in** *lexord-irreflexive* [*THEN notE*])
apply *simp*
apply (*blast intro*: *lexord-trans transI order-less-trans*)
done

instance *list* :: (*linorder*) *linorder*
apply (*intro-classes*, *unfold list-le-def list-less-def*, *safe*)
apply (*cut-tac x* = *x* **and** *y* = *y* **and** *r* = {(*a*,*b*). *a* < *b*} **in** *lexord-linear*)
apply *force*
apply *simp*
done

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition
[*code func del*]: (*inf* :: '*a list* ⇒ -) = *min*

definition
[*code func del*]: (*sup* :: '*a list* ⇒ -) = *max*

instance
by *intro-classes*
(*auto simp add*: *inf-list-def sup-list-def min-max.sup-inf-distrib1*)

end

lemma *not-less-Nil* [*simp*]: ¬ (*x* < [])
by (*unfold list-less-def*) *simp*

lemma *Nil-less-Cons* [*simp*]: [] < *a* # *x*
by (*unfold list-less-def*) *simp*

lemma *Cons-less-Cons* [*simp*]: *a* # *x* < *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* < *y*
by (*unfold list-less-def*) *simp*

lemma *le-Nil* [*simp*]: *x* ≤ [] ⟷ *x* = []
by (*unfold list-le-def*, *cases x*) *auto*

lemma *Nil-le-Cons* [*simp*]: [] ≤ *x*
by (*unfold list-le-def*, *cases x*) *auto*

lemma *Cons-le-Cons* [*simp*]: *a* # *x* ≤ *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* ≤ *y*

by (*unfold list-le-def*) *auto*

lemma *less-code* [*code func*]:
xs < ([]::'*a*::{*eq*, *order*} *list*) ⟷ *False*
[] < (*x*::'*a*::{*eq*, *order*}) # *xs* ⟷ *True*
(*x*::'*a*::{*eq*, *order*}) # *xs* < *y* # *ys* ⟷ *x* < *y* ∨ *x* = *y* ∧ *xs* < *ys*
by *simp-all*

lemma *less-eq-code* [*code func*]:
x # *xs* ≤ ([]::'*a*::{*eq*, *order*} *list*) ⟷ *False*
[] ≤ (*xs*::'*a*::{*eq*, *order*} *list*) ⟷ *True*
(*x*::'*a*::{*eq*, *order*}) # *xs* ≤ *y* # *ys* ⟷ *x* < *y* ∨ *x* = *y* ∧ *xs* ≤ *ys*
by *simp-all*

end

58 Sublist-Order: Sublist Ordering

theory *Sublist-Order*
imports *Main*
begin

This theory defines sublist ordering on lists. A list *ys* is a sublist of a list *xs*, iff one obtains *ys* by erasing some elements from *xs*.

58.1 Definitions and basic lemmas

instantiation *list* :: (*type*) *order*
begin

inductive *less-eq-list* **where**
empty [*simp*, *intro*!]: [] ≤ *xs*
| *drop*: *ys* ≤ *xs* ⟹ *ys* ≤ *x* # *xs*
| *take*: *ys* ≤ *xs* ⟹ *x* # *ys* ≤ *x* # *xs*

lemmas *ileq-empty* = *empty*
lemmas *ileq-drop* = *drop*
lemmas *ileq-take* = *take*

lemma *ileq-cases* [*cases set*, *case-names empty drop take*]:
assumes *xs* ≤ *ys*
and *xs* = [] ⟹ *P*
and ⋀*z zs*. *ys* = *z* # *zs* ⟹ *xs* ≤ *zs* ⟹ *P*
and ⋀*x zs ws*. *xs* = *x* # *zs* ⟹ *ys* = *x* # *ws* ⟹ *zs* ≤ *ws* ⟹ *P*
shows *P*
using *assms* **by** (*blast elim*: *less-eq-list.cases*)

lemma *ileq-induct* [*induct set*, *case-names empty drop take*]:

```
  assumes xs ≤ ys
    and ⋀zs. P [] zs
    and ⋀z zs ws. ws ≤ zs ⟹  P ws zs ⟹ P ws (z # zs)
    and ⋀z zs ws. ws ≤ zs ⟹ P ws zs ⟹ P (z # ws) (z # zs)
  shows P xs ys
  using assms by (induct rule: less-eq-list.induct) blast+

definition
  [code func del]: (xs :: 'a list) < ys ⟷ xs ≤ ys ∧ xs ≠ ys

lemma ileq-length: xs ≤ ys ⟹ length xs ≤ length ys
  by (induct rule: ileq-induct) auto
lemma ileq-below-empty [simp]: xs ≤ [] ⟷ xs = []
  by (auto dest: ileq-length)

instance proof
  fix xs ys :: 'a list
  show xs < ys ⟷ xs ≤ ys ∧ xs ≠ ys unfolding less-list-def ..
next
  fix xs :: 'a list
  show xs ≤ xs by (induct xs) (auto intro!: ileq-empty ileq-drop ileq-take)
next
  fix xs ys :: 'a list

  { fix n
    have !!l l'. ⟦l≤l'; l'≤l; n=length l + length l'⟧ ⟹ l=l'
    proof (induct n rule: nat-less-induct)
      case (1 n l l') from 1.prems(1) show ?case proof (cases rule: ileq-cases)
        case empty with 1.prems(2) show ?thesis by auto
      next
        case (drop a l2') with 1.prems(2) have length l'≤length l length l ≤ length
l2' 1+length l2' = length l' by (auto dest: ileq-length)
        hence False by simp thus ?thesis ..
      next
        case (take a l1' l2') hence LEN': length l1' + length l2' < length l + length
l' by simp
        from 1.prems have LEN: length l' = length l by (auto dest!: ileq-length)
          from 1.prems(2) show ?thesis proof (cases rule: ileq-cases[case-names
empty' drop' take'])
          case empty' with take LEN show ?thesis by simp
        next
          case (drop' ah l2h) with take LEN have length l1' ≤ length l2h 1+length
l2h = length l2' length l2' = length l1' by (auto dest: ileq-length)
          hence False by simp thus ?thesis ..
        next
          case (take' ah l1h l2h)
          with take have 2: ah=a l1h=l2' l2h=l1' l1' ≤ l2' l2' ≤ l1' by auto
          with LEN' 1.hyps 1.prems(3) have l1'=l2' by blast
          with take 2 show ?thesis by simp
```

```
      qed
     qed
   qed
  }
  moreover assume xs ≤ ys ys ≤ xs
  ultimately show xs = ys by blast
next
  fix xs ys zs :: 'a list
  {
   fix n
   have !!x y z. ⟦x ≤ y; y ≤ z; n=length x + length y + length z⟧ ⟹ x ≤ z
   proof (induct rule: nat-less-induct[case-names I])
     case (I n x y z)
     from I.prems(2) show ?case proof (cases rule: ileq-cases)
       case empty with I.prems(1) show ?thesis by auto
     next
      case (drop a z') hence length x + length y + length z' < length x + length
y + length z by simp
       with I.hyps I.prems(3,1) drop(2) have x≤z' by blast
       with drop(1) show ?thesis by (auto intro: ileq-drop)
     next
        case (take a y' z') from I.prems(1) show ?thesis proof (cases rule:
ileq-cases[case-names empty' drop' take'])
        case empty' thus ?thesis by auto
       next
        case (drop' ah y'h) with take have x≤y' y'≤z' length x + length y' +
length z' < length x + length y + length z by auto
        with I.hyps I.prems(3) have x≤z' by (blast)
        with take(2) show ?thesis  by (auto intro: ileq-drop)
       next
        case (take' ah x' y'h) with take have 2: x=a#x' x'≤y' y'≤z' length x'
+ length y' + length z' < length x + length y + length z by auto
        with I.hyps I.prems(3) have x'≤z' by blast
        with 2 take(2) show ?thesis by (auto intro: ileq-take)
       qed
     qed
   qed
  }
  moreover assume xs ≤ ys ys ≤ zs
  ultimately show xs ≤ zs by blast
qed

end

lemmas ileq-intros = ileq-empty ileq-drop ileq-take

lemma ileq-drop-many: xs ≤ ys ⟹ xs ≤ zs @ ys
  by (induct zs) (auto intro: ileq-drop)
lemma ileq-take-many: xs ≤ ys ⟹ zs @ xs ≤ zs @ ys
```

by (*induct zs*) (*auto intro*: *ileq-take*)

lemma *ileq-same-length*: $xs \leq ys \Longrightarrow length\ xs = length\ ys \Longrightarrow xs = ys$
by (*induct rule*: *ileq-induct*) (*auto dest*: *ileq-length*)
lemma *ileq-same-append* [*simp*]: $x \# xs \leq xs \longleftrightarrow False$
by (*auto dest*: *ileq-length*)

lemma *ilt-length* [*intro*]:
assumes $xs < ys$
shows $length\ xs < length\ ys$
proof −
from *assms* **have** $xs \leq ys$ **and** $xs \neq ys$ **by** (*simp-all add*: *less-list-def*)
moreover with *ileq-length* **have** $length\ xs \leq length\ ys$ **by** *auto*
ultimately show *?thesis* **by** (*auto intro*: *ileq-same-length*)
qed

lemma *ilt-empty* [*simp*]: $[] < xs \longleftrightarrow xs \neq []$
by (*unfold less-list-def*, *auto*)
lemma *ilt-emptyI*: $xs \neq [] \Longrightarrow [] < xs$
by (*unfold less-list-def*, *auto*)
lemma *ilt-emptyD*: $[] < xs \Longrightarrow xs \neq []$
by (*unfold less-list-def*, *auto*)
lemma *ilt-below-empty*[*simp*]: $xs < [] \Longrightarrow False$
by (*auto dest*: *ilt-length*)
lemma *ilt-drop*: $xs < ys \Longrightarrow xs < x \# ys$
by (*unfold less-list-def*) (*auto intro*: *ileq-intros*)
lemma *ilt-take*: $xs < ys \Longrightarrow x \# xs < x \# ys$
by (*unfold less-list-def*) (*auto intro*: *ileq-intros*)
lemma *ilt-drop-many*: $xs < ys \Longrightarrow xs < zs\ @\ ys$
by (*induct zs*) (*auto intro*: *ilt-drop*)
lemma *ilt-take-many*: $xs < ys \Longrightarrow zs\ @\ xs < zs\ @\ ys$
by (*induct zs*) (*auto intro*: *ilt-take*)

58.2 Appending elements

lemma *ileq-rev-take*: $xs \leq ys \Longrightarrow xs\ @\ [x] \leq ys\ @\ [x]$
by (*induct rule*: *ileq-induct*) (*auto intro*: *ileq-intros ileq-drop-many*)
lemma *ilt-rev-take*: $xs < ys \Longrightarrow xs\ @\ [x] < ys\ @\ [x]$
by (*unfold less-list-def*) (*auto dest*: *ileq-rev-take*)
lemma *ileq-rev-drop*: $xs \leq ys \Longrightarrow xs \leq ys\ @\ [x]$
by (*induct rule*: *ileq-induct*) (*auto intro*: *ileq-intros*)
lemma *ileq-rev-drop-many*: $xs \leq ys \Longrightarrow xs \leq ys\ @\ zs$
by (*induct zs rule*: *rev-induct*) (*auto dest*: *ileq-rev-drop*)

58.3 Relation to standard list operations

lemma *ileq-map*: $xs \leq ys \Longrightarrow map\ f\ xs \leq map\ f\ ys$
by (*induct rule*: *ileq-induct*) (*auto intro*: *ileq-intros*)
lemma *ileq-filter-left*[*simp*]: $filter\ f\ xs \leq xs$
by (*induct xs*) (*auto intro*: *ileq-intros*)

lemma *ileq-filter*: $xs \leq ys \Longrightarrow$ *filter f xs* $\leq$ *filter f ys*
by (*induct rule*: *ileq-induct*) (*auto intro*: *ileq-intros*)

end

References

[1] Abrial and Laffitte. Towards the mechanization of the proofs of some classical theorems of set theory. Unpublished.

[2] J. Avigad and K. Donnelly. Formalizing O notation in Isabelle/HOL. In D. Basin and M. Rusiowitch, editors, *Automated Reasoning: second international conference, IJCAR 2004*, pages 357–371. Springer, 2004.

[3] H. Davenport. *The Higher Arithmetic.* Cambridge University Press, 1992.

[4] A. Oberschelp. *Rekursionstheorie.* BI-Wissenschafts-Verlag, 1993.

[5] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.