Miscellaneous HOL Examples

December 12, 2016

Contents

1 Some Isar command definitions

theory *Commands*
imports *Main*
keywords
print-test :: *diag* **and**

global-test :: *thy-decl* **and**
local-test :: *thy-decl*
begin

1.1 Diagnostic command: no state change

⟨*ML*⟩

print-test *x*
print-test $\lambda x.\ x = a$

1.2 Old-style global theory declaration

⟨*ML*⟩

global-test *a*
global-test *b*
print-test *a*

1.3 Local theory specification

⟨*ML*⟩

local-test *true* = *True*
print-test *true*
thm *true-def*

local-test *identity* = $\lambda x.\ x$
print-test *identity x*
thm *identity-def*

context fixes *x y* :: *nat*
begin

local-test *test* = $x + y$
print-test *test*
thm *test-def*

end

print-test *test 0 1*
thm *test-def*

end

2 Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*

begin

The set of natural numbers is infinite.

lemma *infinite-nat-iff-unbounded-le*: *infinite* (*S*::*nat set*) $\longleftrightarrow$ ($\forall\, m.\ \exists\, n{\geq}m.\ n \in S$)
⟨*proof*⟩

lemma *infinite-nat-iff-unbounded*: *infinite* (*S*::*nat set*) $\longleftrightarrow$ ($\forall\, m.\ \exists\, n{>}m.\ n \in S$)
⟨*proof*⟩

lemma *finite-nat-iff-bounded*: *finite* (*S*::*nat set*) $\longleftrightarrow$ ($\exists\, k.\ S \subseteq \{..{<}k\}$)
⟨*proof*⟩

lemma *finite-nat-iff-bounded-le*: *finite* (*S*::*nat set*) $\longleftrightarrow$ ($\exists\, k.\ S \subseteq \{..\ k\}$)
⟨*proof*⟩

lemma *finite-nat-bounded*: *finite* (*S*::*nat set*) $\Longrightarrow$ $\exists\, k.\ S \subseteq \{..{<}k\}$
⟨*proof*⟩

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k, there is some larger number that is an element of the set.

lemma *unbounded-k-infinite*: $\forall\, m{>}k.\ \exists\, n{>}m.\ n \in S \Longrightarrow$ *infinite* (*S*::*nat set*)
⟨*proof*⟩

lemma *nat-not-finite*: *finite* (*UNIV*::*nat set*) $\Longrightarrow R$
⟨*proof*⟩

lemma *range-inj-infinite*:
inj (*f*::*nat* $\Rightarrow$ *'a*) $\Longrightarrow$ *infinite* (*range f*)
⟨*proof*⟩

The set of integers is also infinite.

lemma *infinite-int-iff-infinite-nat-abs*: *infinite* (*S*::*int set*) $\longleftrightarrow$ *infinite* ((*nat o abs*) ‘ *S*)
⟨*proof*⟩

proposition *infinite-int-iff-unbounded-le*: *infinite* (*S*::*int set*) $\longleftrightarrow$ ($\forall\, m.\ \exists\, n.\ |n| \geq m \wedge n \in S$)
⟨*proof*⟩

proposition *infinite-int-iff-unbounded*: *infinite* (*S*::*int set*) $\longleftrightarrow$ ($\forall\, m.\ \exists\, n.\ |n| > m \wedge n \in S$)
⟨*proof*⟩

proposition *finite-int-iff-bounded*: *finite* (*S*::*int set*) $\longleftrightarrow$ ($\exists\, k.$ *abs* ‘ *S* $\subseteq \{..{<}k\}$)
⟨*proof*⟩

proposition *finite-int-iff-bounded-le*: *finite* (S::*int set*) $\longleftrightarrow$ ($\exists\, k.$ *abs* ' $S \subseteq \{..\, k\}$)
⟨*proof*⟩

2.1 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

lemma *not-INFM* [*simp*]: $\neg$ (*INFM* $x.$ $P\ x$) $\longleftrightarrow$ (*MOST* $x.$ $\neg$ $P\ x$) ⟨*proof*⟩
lemma *not-MOST* [*simp*]: $\neg$ (*MOST* $x.$ $P\ x$) $\longleftrightarrow$ (*INFM* $x.$ $\neg$ $P\ x$) ⟨*proof*⟩

lemma *INFM-const* [*simp*]: (*INFM* x::$'a.$ P) $\longleftrightarrow$ $P \wedge$ *infinite* (*UNIV*::$'a$ *set*)
⟨*proof*⟩

lemma *MOST-const* [*simp*]: (*MOST* x::$'a.$ P) $\longleftrightarrow$ $P \vee$ *finite* (*UNIV*::$'a$ *set*)
⟨*proof*⟩

lemma *INFM-imp-distrib*: (*INFM* $x.$ $P\ x \longrightarrow Q\ x$) $\longleftrightarrow$ ((*MOST* $x.$ $P\ x$) $\longrightarrow$ (*INFM* $x.$ $Q\ x$))
⟨*proof*⟩

lemma *MOST-imp-iff*: *MOST* $x.$ $P\ x \Longrightarrow$ (*MOST* $x.$ $P\ x \longrightarrow Q\ x$) $\longleftrightarrow$ (*MOST* $x.$ $Q\ x$)
⟨*proof*⟩

lemma *INFM-conjI*: *INFM* $x.$ $P\ x \Longrightarrow$ *MOST* $x.$ $Q\ x \Longrightarrow$ *INFM* $x.$ $P\ x \wedge Q\ x$
⟨*proof*⟩

Properties of quantifiers with injective functions.

lemma *INFM-inj*: *INFM* $x.$ $P\ (f\ x) \Longrightarrow$ *inj* $f \Longrightarrow$ *INFM* $x.$ $P\ x$
⟨*proof*⟩

lemma *MOST-inj*: *MOST* $x.$ $P\ x \Longrightarrow$ *inj* $f \Longrightarrow$ *MOST* $x.$ $P\ (f\ x)$
⟨*proof*⟩

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
$\neg$ (*INFM* $x.$ $x = a$)
$\neg$ (*INFM* $x.$ $a = x$)
⟨*proof*⟩

lemma *MOST-neq* [*simp*]:
MOST $x.$ $x \neq a$
MOST $x.$ $a \neq x$
⟨*proof*⟩

lemma *INFM-neq* [*simp*]:
(*INFM* x::$'a.$ $x \neq a$) $\longleftrightarrow$ *infinite* (*UNIV*::$'a$ *set*)

$(INFM\ x{::}'a.\ a \neq x) \longleftrightarrow infinite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *MOST-eq* [*simp*]:
$(MOST\ x{::}'a.\ x = a) \longleftrightarrow finite\ (UNIV{::}'a\ set)$
$(MOST\ x{::}'a.\ a = x) \longleftrightarrow finite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *MOST-eq-imp*:
$MOST\ x.\ x = a \longrightarrow P\ x$
$MOST\ x.\ a = x \longrightarrow P\ x$
⟨*proof*⟩

Properties of quantifiers over the naturals.

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n{::}nat)) \longleftrightarrow (\exists\, m.\ \forall\, n{>}m.\ P\ n)$
⟨*proof*⟩

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n{::}nat)) \longleftrightarrow (\exists\, m.\ \forall\, n{\geq}m.\ P\ n)$
⟨*proof*⟩

lemma *INFM-nat*: $(\exists_\infty n.\ P\ (n{::}nat)) \longleftrightarrow (\forall\, m.\ \exists\, n{>}m.\ P\ n)$
⟨*proof*⟩

lemma *INFM-nat-le*: $(\exists_\infty n.\ P\ (n{::}nat)) \longleftrightarrow (\forall\, m.\ \exists\, n{\geq}m.\ P\ n)$
⟨*proof*⟩

lemma *MOST-INFM*: $infinite\ (UNIV{::}'a\ set) \Longrightarrow MOST\ x{::}'a.\ P\ x \Longrightarrow INFM\ x{::}'a.\ P\ x$
⟨*proof*⟩

lemma *MOST-Suc-iff*: $(MOST\ n.\ P\ (Suc\ n)) \longleftrightarrow (MOST\ n.\ P\ n)$
⟨*proof*⟩

lemma
shows *MOST-SucI*: $MOST\ n.\ P\ n \Longrightarrow MOST\ n.\ P\ (Suc\ n)$
and *MOST-SucD*: $MOST\ n.\ P\ (Suc\ n) \Longrightarrow MOST\ n.\ P\ n$
⟨*proof*⟩

lemma *MOST-ge-nat*: $MOST\ n{::}nat.\ m \leq n$
⟨*proof*⟩

lemma *Inf-many-def*: $Inf\text{-}many\ P \longleftrightarrow infinite\ \{x.\ P\ x\}$ ⟨*proof*⟩
lemma *Alm-all-def*: $Alm\text{-}all\ P \longleftrightarrow \neg\ (INFM\ x.\ \neg\ P\ x)$ ⟨*proof*⟩
lemma *INFM-iff-infinite*: $(INFM\ x.\ P\ x) \longleftrightarrow infinite\ \{x.\ P\ x\}$ ⟨*proof*⟩
lemma *MOST-iff-cofinite*: $(MOST\ x.\ P\ x) \longleftrightarrow finite\ \{x.\ \neg\ P\ x\}$ ⟨*proof*⟩
lemma *INFM-EX*: $(\exists_\infty x.\ P\ x) \Longrightarrow (\exists\, x.\ P\ x)$ ⟨*proof*⟩
lemma *ALL-MOST*: $\forall\, x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$ ⟨*proof*⟩
lemma *INFM-mono*: $\exists_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \exists_\infty x.\ Q\ x$ ⟨*proof*⟩

lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *INFM-disj-distrib*: $(\exists_\infty x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_\infty x.\ P\ x) \vee (\exists_\infty x.\ Q\ x)$ ⟨*proof*⟩
lemma *MOST-rev-mp*: $\forall_\infty x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x \longrightarrow Q\ x \Longrightarrow \forall_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *MOST-conj-distrib*: $(\forall_\infty x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall_\infty x.\ P\ x) \wedge (\forall_\infty x.\ Q\ x)$ ⟨*proof*⟩
lemma *MOST-conjI*: *MOST* $x.\ P\ x \Longrightarrow$ *MOST* $x.\ Q\ x \Longrightarrow$ *MOST* $x.\ P\ x \wedge Q\ x$ ⟨*proof*⟩
lemma *INFM-finite-Bex-distrib*: *finite* $A \Longrightarrow (INFM\ y.\ \exists x{\in}A.\ P\ x\ y) \longleftrightarrow (\exists x{\in}A.\ INFM\ y.\ P\ x\ y)$ ⟨*proof*⟩
lemma *MOST-finite-Ball-distrib*: *finite* $A \Longrightarrow (MOST\ y.\ \forall x{\in}A.\ P\ x\ y) \longleftrightarrow (\forall x{\in}A.\ MOST\ y.\ P\ x\ y)$ ⟨*proof*⟩
lemma *INFM-E*: *INFM* $x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow thesis) \Longrightarrow thesis$ ⟨*proof*⟩
lemma *MOST-I*: $(\bigwedge x.\ P\ x) \Longrightarrow$ *MOST* $x.\ P\ x$ ⟨*proof*⟩
lemmas *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

2.2 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

Could be generalized to $enumerate'\ S\ n = (SOME\ t.\ t \in s \wedge finite\ \{s \in S.\ s < t\} \wedge card\ \{s \in S.\ s < t\} = n)$.

primrec (**in** *wellorder*) *enumerate* :: *'a set* ⇒ *nat* ⇒ *'a*
where
  enumerate-0: *enumerate S 0* = (*LEAST n. n* ∈ *S*)
| *enumerate-Suc*: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*LEAST n. n* ∈ *S*}) *n*

lemma *enumerate-Suc'*: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*enumerate S 0*}) *n*
  ⟨*proof*⟩

lemma *enumerate-in-set*: *infinite S* ⟹ *enumerate S n* ∈ *S*
  ⟨*proof*⟩

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

lemma *enumerate-step*: *infinite S* ⟹ *enumerate S n* < *enumerate S* (*Suc n*)
  ⟨*proof*⟩

lemma *enumerate-mono*: *m* < *n* ⟹ *infinite S* ⟹ *enumerate S m* < *enumerate S n*
  ⟨*proof*⟩

lemma *le-enumerate*:
  assumes *S*: *infinite S*
  shows *n* ≤ *enumerate S n*
  ⟨*proof*⟩

lemma *enumerate-Suc''*:
fixes $S :: 'a::wellorder\ set$
assumes *infinite* S
shows $enumerate\ S\ (Suc\ n) = (LEAST\ s.\ s \in S \wedge enumerate\ S\ n < s)$
⟨*proof*⟩

lemma *enumerate-Ex*:
assumes S: *infinite* $(S::nat\ set)$
shows $s \in S \Longrightarrow \exists\, n.\ enumerate\ S\ n = s$
⟨*proof*⟩

lemma *bij-enumerate*:
fixes $S :: nat\ set$
assumes S: *infinite* S
shows *bij-betw* $(enumerate\ S)$ $UNIV\ S$
⟨*proof*⟩

A pair of weird and wonderful lemmas from HOL Light

lemma *finite-transitivity-chain*:
assumes *finite* A
and R: $\bigwedge x.\ \sim R\ x\ x \bigwedge x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z$
and A: $\bigwedge x.\ x \in A \Longrightarrow \exists\, y.\ y \in A \wedge R\ x\ y$
shows $A = \{\}$
⟨*proof*⟩

corollary *Union-maximal-sets*:
assumes *finite* $\mathcal{F}$
shows $\bigcup \{T \in \mathcal{F}.\ \forall\, U{\in}\mathcal{F}.\ \neg\ T \subset U\} = \bigcup \mathcal{F}$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

3 Ad Hoc Overloading

theory *Adhoc-Overloading-Examples*
imports
Main
~~/*src*/*HOL*/*Library*/*Infinite-Set*
~~/*src*/*Tools*/*Adhoc-Overloading*
begin

Adhoc overloading allows to overload a constant depending on its type. Typically this involves to introduce an uninterpreted constant (used for input and output) and then add some variants (used internally).

3.1 Plain Ad Hoc Overloading

Consider the type of first-order terms.

```
datatype ('a, 'b) term =
  Var 'b |
  Fun 'a ('a, 'b) term list
```

The set of variables of a term might be computed as follows.

```
fun term-vars :: ('a, 'b) term ⇒ 'b set where
  term-vars (Var x) = {x} |
  term-vars (Fun f ts) = ⋃ set (map term-vars ts)
```

However, also for *rules* (i.e., pairs of terms) and term rewrite systems (i.e., sets of rules), the set of variables makes sense. Thus we introduce an unspecified constant *vars*.

```
consts vars :: 'a ⇒ 'b set
```

Which is then overloaded with variants for terms, rules, and TRSs.

```
adhoc-overloading
  vars term-vars

value vars (Fun ''f'' [Var 0, Var 1])

fun rule-vars :: ('a, 'b) term × ('a, 'b) term ⇒ 'b set where
  rule-vars (l, r) = vars l ∪ vars r

adhoc-overloading
  vars rule-vars

value vars (Var 1, Var 0)

definition trs-vars :: (('a, 'b) term × ('a, 'b) term) set ⇒ 'b set where
  trs-vars R = ⋃ (rule-vars ‘ R)

adhoc-overloading
  vars trs-vars

value vars {(Var 1, Var 0)}
```

Sometimes it is necessary to add explicit type constraints before a variant can be determined.

```
value vars (R :: (('a, 'b) term × ('a, 'b) term) set)
```

It is also possible to remove variants.

```
no-adhoc-overloading
  vars term-vars rule-vars
```

As stated earlier, the overloaded constant is only used for input and output. Internally, always a variant is used, as can be observed by the configuration option *show-variants*.

adhoc-overloading
  vars term-vars

declare [[*show-variants*]]

term *vars* (*Var 1*)

3.2 Adhoc Overloading inside Locales

As example we use permutations that are parametrized over an atom type $'a$.

definition *perms* :: $('a \Rightarrow 'a)$ *set* **where**
  $perms = \{f.\ bij\ f \wedge finite\ \{x.\ f\ x \neq x\}\}$

typedef $'a$ *perm* = *perms* :: $('a \Rightarrow 'a)$ *set*
  ⟨*proof*⟩

First we need some auxiliary lemmas.

lemma *permsI* [*Pure.intro*]:
  assumes *bij f* **and** *MOST x. f x = x*
  shows $f \in perms$
  ⟨*proof*⟩

lemma *perms-imp-bij*:
  $f \in perms \Longrightarrow bij\ f$
  ⟨*proof*⟩

lemma *perms-imp-MOST-eq*:
  $f \in perms \Longrightarrow MOST\ x.\ f\ x = x$
  ⟨*proof*⟩

lemma *id-perms* [*simp*]:
  $id \in perms$
  $(\lambda x.\ x) \in perms$
  ⟨*proof*⟩

lemma *perms-comp* [*simp*]:
  assumes *f*: $f \in perms$ **and** *g*: $g \in perms$
  shows $(f \circ g) \in perms$
  ⟨*proof*⟩

lemma *perms-inv*:
  assumes *f*: $f \in perms$
  shows $inv\ f \in perms$
  ⟨*proof*⟩

lemma *bij-Rep-perm*: *bij* (*Rep-perm p*)
⟨*proof*⟩

instantiation *perm* :: (*type*) *group-add*
begin

definition *0* = *Abs-perm id*
definition − *p* = *Abs-perm* (*inv* (*Rep-perm p*))
definition *p* + *q* = *Abs-perm* (*Rep-perm p* ∘ *Rep-perm q*)
definition (*p1*::'*a perm*) − *p2* = *p1* + − *p2*

lemma *Rep-perm-0*: *Rep-perm 0* = *id*
⟨*proof*⟩

lemma *Rep-perm-add*:
Rep-perm (*p1* + *p2*) = *Rep-perm p1* ∘ *Rep-perm p2*
⟨*proof*⟩

lemma *Rep-perm-uminus*:
Rep-perm (− *p*) = *inv* (*Rep-perm p*)
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemmas *Rep-perm-simps* =
Rep-perm-0
Rep-perm-add
Rep-perm-uminus

4 Permutation Types

We want to be able to apply permutations to arbitrary types. To this end we introduce a constant *PERMUTE* together with convenient infix syntax.

consts *PERMUTE* :: '*a perm* ⇒ '*b* ⇒ '*b* (**infixr** · *75*)

Then we add a locale for types '*b* that support appliciation of permutations.

locale *permute* =
fixes *permute* :: '*a perm* ⇒ '*b* ⇒ '*b*
assumes *permute-zero* [*simp*]: *permute 0 x* = *x*
and *permute-plus* [*simp*]: *permute* (*p* + *q*) *x* = *permute p* (*permute q x*)
begin

adhoc-overloading
PERMUTE permute

end

Permuting atoms.

definition *permute-atom* :: *'a perm* ⇒ *'a* ⇒ *'a* **where**
permute-atom p a = (*Rep-perm p*) *a*

adhoc-overloading
PERMUTE permute-atom

interpretation *atom-permute*: *permute permute-atom*
⟨*proof*⟩

Permuting permutations.

definition *permute-perm* :: *'a perm* ⇒ *'a perm* ⇒ *'a perm* **where**
permute-perm p q = *p* + *q* − *p*

adhoc-overloading
PERMUTE permute-perm

interpretation *perm-permute*: *permute permute-perm*
⟨*proof*⟩

Permuting functions.

locale *fun-permute* =
dom: *permute perm1* + *ran*: *permute perm2*
for *perm1* :: *'a perm* ⇒ *'b* ⇒ *'b*
and *perm2* :: *'a perm* ⇒ *'c* ⇒ *'c*
begin

adhoc-overloading
PERMUTE perm1 perm2

definition *permute-fun* :: *'a perm* ⇒ (*'b* ⇒ *'c*) ⇒ (*'b* ⇒ *'c*) **where**
permute-fun p f = (λ*x*. *p* · (*f* (−*p* · *x*)))

adhoc-overloading
PERMUTE permute-fun

end

sublocale *fun-permute* ⊆ *permute permute-fun*
⟨*proof*⟩

lemma (*Abs-perm id* :: *nat perm*) · *Suc 0* = *Suc 0*
⟨*proof*⟩

interpretation *atom-fun-permute*: *fun-permute permute-atom permute-atom*
⟨*proof*⟩

adhoc-overloading
PERMUTE atom-fun-permute.permute-fun

lemma (*Abs-perm id* :: *'a perm*) ⋅ *id* = *id*
⟨*proof*⟩

end

5 Example of Declaring an Oracle

theory *Iff-Oracle*
imports *Main*
begin

5.1 Oracle declaration

This oracle makes tautologies of the form $P = (P = (P = P))$. The length is specified by an integer, which is checked to be even and positive.

⟨*ML*⟩

5.2 Oracle as low-level rule

⟨*ML*⟩
These oracle calls had better fail.

⟨*ML*⟩

5.3 Oracle as proof method

⟨*ML*⟩

lemma $A \longleftrightarrow A$
⟨*proof*⟩

lemma $A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A$
⟨*proof*⟩

lemma $A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A \longleftrightarrow A$
⟨*proof*⟩

lemma *A*
⟨*proof*⟩

end

theory *Coercion-Examples*
imports *Main*
begin

declare[[*coercion-enabled*]]

consts *func* :: (*nat* ⇒ *int*) ⇒ *nat*
consts *arg* :: *int* ⇒ *nat*

consts *func′* :: *int* ⇒ *int*
consts *arg′* :: *nat*

abbreviation *nat-of-bool* :: *bool* ⇒ *nat*
where
nat-of-bool ≡ *of-bool*

declare [[*coercion nat-of-bool*]]

declare [[*coercion int*]]

declare [[*coercion-map map*]]

definition *map-fun* :: (*'a* ⇒ *'b*) ⇒ (*'c* ⇒ *'d*) ⇒ (*'b* ⇒ *'c*) ⇒ (*'a* ⇒ *'d*) **where**
map-fun f g h = *g o h o f*

declare [[*coercion-map* λ *f g h* . *g o h o f*]]

primrec *map-prod* :: (*'a* ⇒ *'c*) ⇒ (*'b* ⇒ *'d*) ⇒ (*'a* ∗ *'b*) ⇒ (*'c* ∗ *'d*) **where**
map-prod f g (*x*,*y*) = (*f x*, *g y*)

declare [[*coercion-map map-prod*]]

term (*1*::*nat*) = *True*

term *True* = (*1*::*nat*)

term (*1*::*nat*) = (*True* = (*1*::*nat*))

term $op = (True = (1::nat))$

term $[1::nat, True]$

term $[True, 1::nat]$

term $[1::nat] = [True]$

term $[True] = [1::nat]$

term $[[True]] = [[1::nat]]$

term $[[[[[[[[[True]]]]]]]]] = [[[[[[[[[1::nat]]]]]]]]]$

term $[[True],[42::nat]] = rev\ [[True]]$

term $rev\ [10000::nat] = [False,\ 42000::nat,\ True]$

term $\lambda\ x\ .\ x = (3::nat)$

term $(\lambda\ x\ .\ x = (3::nat))\ True$

term $map\ (\lambda\ x\ .\ x = (3::nat))$

term $map\ (\lambda\ x\ .\ x = (3::nat))\ [True, 1::nat]$

consts $bnn :: (bool \Rightarrow nat) \Rightarrow nat$
consts $nb :: nat \Rightarrow bool$
consts $ab :: 'a \Rightarrow bool$

term $bnn\ nb$

term $bnn\ ab$

term $\lambda\ x\ .\ x = (3::int)$

term $map\ (\lambda\ x\ .\ x = (3::int))\ [True]$

term $map\ (\lambda\ x\ .\ x = (3::int))\ [True, 1::nat]$

term $map\ (\lambda\ x\ .\ x = (3::int))\ [True, 1::nat, 1::int]$

term $[1::nat, True, 1::int, False]$

term $map\ (map\ (\lambda\ x\ .\ x = (3::int)))\ [[True],[1::nat],[True, 1::int]]$

consts $cbool :: 'a \Rightarrow bool$
consts $cnat :: 'a \Rightarrow nat$
consts $cint :: 'a \Rightarrow int$

term [*id*, *cbool*, *cnat*, *cint*]

consts *funfun* :: (*'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b*
consts *flip* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'b* ⇒ *'a* ⇒ *'c*

term *flip funfun*

term *map funfun* [*id*,*cnat*,*cint*,*cbool*]

term *map* (*flip funfun True*)

term *map* (*flip funfun True*) [*id*,*cnat*,*cint*,*cbool*]

consts *ii* :: *int* ⇒ *int*
consts *aaa* :: *'a* ⇒ *'a* ⇒ *'a*
consts *nlist* :: *nat list*
consts *ilil* :: *int list* ⇒ *int list*

term *ii* (*aaa* (*1*::*nat*) *True*)

term *map ii nlist*

term *ilil nlist*

definition *xs* :: *bool list* **where** *xs* = [*True*]

term (*xs*::*nat list*)

term (*1*::*nat*) = *True*

term *True* = (*1*::*nat*)

term *int* (*1*::*nat*)

term ((*True*::*nat*)::*int*)

term *1*::*nat*

term *nat 1*

definition *C* :: *nat*
where *C* = *123*

consts *g* :: *int* ⇒ *int*

consts $h :: nat \Rightarrow nat$

term $(g\ (1{::}nat)) + (h\ 2)$

term $g\ 1$

term $1{+}(1{::}nat)$

term $((1{::}int) + (1{::}nat),(1{::}int))$

definition $ys :: bool\ list\ list\ list\ list\ list$ **where** $ys{=}[[[[[True]]]]]$

term $ys{=}[[[[[1{::}nat]]]]]$

typedecl $('a,\ 'b,\ 'c)\ F$
consts $Fmap :: ('a \Rightarrow 'd) \Rightarrow ('a,\ 'b,\ 'c)\ F \Rightarrow ('d,\ 'b,\ 'c)\ F$
consts $z :: (bool,\ nat,\ bool)\ F$
declare $[[coercion\text{-}map\ Fmap :: ('a \Rightarrow 'd) \Rightarrow ('a,\ 'b,\ 'c)\ F \Rightarrow ('d,\ 'b,\ 'c)\ F]]$
term $z :: (nat,\ nat,\ bool)\ F$

consts
  $case\text{-}nil :: 'a \Rightarrow 'b$
  $case\text{-}cons :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
  $case\text{-}abs :: ('c \Rightarrow 'b) \Rightarrow 'b$
  $case\text{-}elem :: 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b$

declare $[[coercion\text{-}args\ case\text{-}cons\ -\ -]]$
declare $[[coercion\text{-}args\ case\text{-}abs\ -]]$
declare $[[coercion\text{-}args\ case\text{-}elem\ -\ +]]$

term $case\text{-}cons\ (case\text{-}abs\ (\lambda n.\ case\text{-}abs\ (\lambda is.\ case\text{-}elem\ (((n{::}nat),(is{::}int\ list)))$
$(n\#is))))\ case\text{-}nil$

consts $n :: nat\ m :: nat$
term $-\ (n + m)$
declare $[[coercion\text{-}args\ uminus\ -]]$
declare $[[coercion\text{-}args\ plus\ +\ +]]$
term $-\ (n + m)$

end

6 Abstract Natural Numbers primitive recursion

theory *Abstract-NAT*
imports *Main*
begin

Axiomatic Natural Numbers (Peano) – a monomorphic theory.

locale $NAT =$

fixes *zero* :: $'n$
and *succ* :: $'n \Rightarrow 'n$
assumes *succ-inject* [*simp*]: $succ\ m = succ\ n \longleftrightarrow m = n$
and *succ-neq-zero* [*simp*]: $succ\ m \neq zero$
and *induct* [*case-names zero succ*, *induct type*: $'n$]:
$P\ zero \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (succ\ n)) \Longrightarrow P\ n$
begin

lemma *zero-neq-succ* [*simp*]: $zero \neq succ\ m$
⟨*proof*⟩

Primitive recursion as a (functional) relation – polymorphic!

inductive *Rec* :: $'a \Rightarrow ('n \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'n \Rightarrow 'a \Rightarrow bool$
for $e :: 'a$ **and** $r :: 'n \Rightarrow 'a \Rightarrow 'a$
where
Rec-zero: $Rec\ e\ r\ zero\ e$
| *Rec-succ*: $Rec\ e\ r\ m\ n \Longrightarrow Rec\ e\ r\ (succ\ m)\ (r\ m\ n)$

lemma *Rec-functional*:
fixes $x :: 'n$
shows $\exists!y{::}'a.\ Rec\ e\ r\ x\ y$
⟨*proof*⟩

The recursion operator – polymorphic!

definition *rec* :: $'a \Rightarrow ('n \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'n \Rightarrow 'a$
where $rec\ e\ r\ x = (THE\ y.\ Rec\ e\ r\ x\ y)$

lemma *rec-eval*:
assumes *Rec*: $Rec\ e\ r\ x\ y$
shows $rec\ e\ r\ x = y$
⟨*proof*⟩

lemma *rec-zero* [*simp*]: $rec\ e\ r\ zero = e$
⟨*proof*⟩

lemma *rec-succ* [*simp*]: $rec\ e\ r\ (succ\ m) = r\ m\ (rec\ e\ r\ m)$
⟨*proof*⟩

Example: addition (monomorphic)

definition *add* :: $'n \Rightarrow 'n \Rightarrow 'n$
where $add\ m\ n = rec\ n\ (\lambda\text{-}\ k.\ succ\ k)\ m$

lemma *add-zero* [*simp*]: $add\ zero\ n = n$
and *add-succ* [*simp*]: $add\ (succ\ m)\ n = succ\ (add\ m\ n)$
⟨*proof*⟩

lemma *add-assoc*: $add\ (add\ k\ m)\ n = add\ k\ (add\ m\ n)$

⟨*proof*⟩

lemma *add-zero-right*: *add m zero* = *m*
⟨*proof*⟩

lemma *add-succ-right*: *add m* (*succ n*) = *succ* (*add m n*)
⟨*proof*⟩

lemma *add* (*succ* (*succ* (*succ zero*))) (*succ* (*succ zero*)) =
succ (*succ* (*succ* (*succ* (*succ zero*))))
⟨*proof*⟩

Example: replication (polymorphic)

definition *repl* :: *'n* ⇒ *'a* ⇒ *'a list*
where *repl n x* = *rec* [] (λ- *xs*. *x* # *xs*) *n*

lemma *repl-zero* [*simp*]: *repl zero x* = []
and *repl-succ* [*simp*]: *repl* (*succ n*) *x* = *x* # *repl n x*
⟨*proof*⟩

lemma *repl* (*succ* (*succ* (*succ zero*))) *True* = [*True*, *True*, *True*]
⟨*proof*⟩

end

Just see that our abstract specification makes sense ...

interpretation *NAT 0 Suc*
⟨*proof*⟩

end

7 Proof by guessing

theory *Guess*
imports *Main*
begin

notepad
begin
⟨*proof*⟩
end

end

8 Examples of function definitions

theory *Functions*

imports *Main* ~~/*src*/*HOL*/*Library*/*Monad-Syntax*
begin

8.1 Very basic

fun *fib* :: *nat* ⇒ *nat*
where
  fib 0 = *1*
| *fib* (*Suc 0*) = *1*
| *fib* (*Suc* (*Suc n*)) = *fib n* + *fib* (*Suc n*)

Partial simp and induction rules:

thm *fib.psimps*
thm *fib.pinduct*

There is also a cases rule to distinguish cases along the definition:

thm *fib.cases*

Total simp and induction rules:

thm *fib.simps*
thm *fib.induct*

Elimination rules:

thm *fib.elims*

8.2 Currying

fun *add*
where
  add 0 y = *y*
| *add* (*Suc x*) *y* = *Suc* (*add x y*)

thm *add.simps*
thm *add.induct* — Note the curried induction predicate

8.3 Nested recursion

function *nz*
where
  nz 0 = *0*
| *nz* (*Suc x*) = *nz* (*nz x*)
⟨*proof*⟩

lemma *nz-is-zero*: — A lemma we need to prove termination
  assumes *trm*: *nz-dom x*
  shows *nz x* = *0*
⟨*proof*⟩

termination *nz*

⟨*proof*⟩

thm *nz.simps*
thm *nz.induct*

8.3.1 Here comes McCarthy's 91-function

function *f91* :: *nat* ⇒ *nat*
where
  f91 n = (*if 100* < *n then n* − *10 else f91* (*f91* (*n* + *11*)))
⟨*proof*⟩

Prove a lemma before attempting a termination proof:

lemma *f91-estimate*:
  assumes *trm*: *f91-dom n*
  shows *n* < *f91 n* + *11*
⟨*proof*⟩

termination
⟨*proof*⟩

Now trivial (even though it does not belong here):

lemma *f91 n* = (*if 100* < *n then n* − *10 else 91*)
  ⟨*proof*⟩

8.4 More general patterns

8.4.1 Overlapping patterns

Currently, patterns must always be compatible with each other, since no automatic splitting takes place. But the following definition of GCD is OK, although patterns overlap:

fun *gcd2* :: *nat* ⇒ *nat* ⇒ *nat*
where
  gcd2 x 0 = *x*
| *gcd2 0 y* = *y*
| *gcd2* (*Suc x*) (*Suc y*) = (*if x* < *y then gcd2* (*Suc x*) (*y* − *x*)
                                else gcd2 (*x* − *y*) (*Suc y*))

thm *gcd2.simps*
thm *gcd2.induct*

8.4.2 Guards

We can reformulate the above example using guarded patterns:

function *gcd3* :: *nat* ⇒ *nat* ⇒ *nat*
where
  gcd3 x 0 = *x*

| *gcd3 0 y = y*
| *gcd3 (Suc x) (Suc y) = gcd3 (Suc x) (y − x)* **if** *x < y*
| *gcd3 (Suc x) (Suc y) = gcd3 (x − y) (Suc y)* **if** *¬ x < y*
⟨*proof*⟩
termination ⟨*proof*⟩

thm *gcd3.simps*
thm *gcd3.induct*

General patterns allow even strange definitions:

function *ev :: nat ⇒ bool*
where
*ev (2 * n) = True*
| *ev (2 * n + 1) = False*
⟨*proof*⟩
termination ⟨*proof*⟩

thm *ev.simps*
thm *ev.induct*
thm *ev.cases*

8.5 Mutual Recursion

fun *evn od :: nat ⇒ bool*
where
evn 0 = True
| *od 0 = False*
| *evn (Suc n) = od n*
| *od (Suc n) = evn n*

thm *evn.simps*
thm *od.simps*

thm *evn-od.induct*
thm *evn-od.termination*

thm *evn.elims*
thm *od.elims*

8.6 Definitions in local contexts

locale *my-monoid =*
fixes *opr :: 'a ⇒ 'a ⇒ 'a*
and *un :: 'a*
assumes *assoc*: *opr (opr x y) z = opr x (opr y z)*
and *lunit*: *opr un x = x*
and *runit*: *opr x un = x*
begin

fun *foldR* :: *'a list* ⇒ *'a*
where
  foldR [] = *un*
| *foldR* (*x* # *xs*) = *opr x* (*foldR xs*)

fun *foldL* :: *'a list* ⇒ *'a*
where
  foldL [] = *un*
| *foldL* [*x*] = *x*
| *foldL* (*x* # *y* # *ys*) = *foldL* (*opr x y* # *ys*)

thm *foldL.simps*

lemma *foldR-foldL*: *foldR xs* = *foldL xs*
  ⟨*proof*⟩

thm *foldR-foldL*

end

thm *my-monoid.foldL.simps*
thm *my-monoid.foldR-foldL*

8.7 *fun-cases*

8.7.1 Predecessor

fun *pred* :: *nat* ⇒ *nat*
where
  pred 0 = *0*
| *pred* (*Suc n*) = *n*

thm *pred.elims*

lemma
  assumes *pred x* = *y*
  obtains *x* = *0 y* = *0* | *n* **where** *x* = *Suc n y* = *n*
  ⟨*proof*⟩

If the predecessor of a number is 0, that number must be 0 or 1.

fun-cases *pred0E*[*elim*]: *pred n* = *0*

lemma *pred n* = *0* ⟹ *n* = *0* ∨ *n* = *Suc 0*
  ⟨*proof*⟩

Other expressions on the right-hand side also work, but whether the generated rule is useful depends on how well the simplifier can simplify it. This example works well:

fun-cases *pred42E*[*elim*]: *pred n* = *42*

lemma *pred n = 42 ⟹ n = 43*
⟨*proof*⟩

8.7.2 List to option

fun *list-to-option* :: *'a list ⇒ 'a option*
where
list-to-option [x] = Some x
| *list-to-option - = None*

fun-cases *list-to-option-NoneE*: *list-to-option xs = None*
and *list-to-option-SomeE*: *list-to-option xs = Some x*

lemma *list-to-option xs = Some y ⟹ xs = [y]*
⟨*proof*⟩

8.7.3 Boolean Functions

fun *xor* :: *bool ⇒ bool ⇒ bool*
where
xor False False = False
| *xor True True = False*
| *xor - - = True*

thm *xor.elims*

fun-cases does not only recognise function equations, but also works with functions that return a boolean, e.g.:

fun-cases *xor-TrueE*: *xor a b* **and** *xor-FalseE*: *¬xor a b*
print-theorems

8.7.4 Many parameters

fun *sum4* :: *nat ⇒ nat ⇒ nat ⇒ nat ⇒ nat*
where *sum4 a b c d = a + b + c + d*

fun-cases *sum40E*: *sum4 a b c d = 0*

lemma *sum4 a b c d = 0 ⟹ a = 0*
⟨*proof*⟩

8.8 Partial Function Definitions

Partial functions in the option monad:

partial-function (*option*)
collatz :: *nat ⇒ nat list option*
where
collatz n =

```
  (if n ≤ 1 then Some [n]
   else if even n
     then do { ns ← collatz (n div 2); Some (n # ns) }
     else do { ns ← collatz (3 * n + 1);  Some (n # ns)})

declare collatz.simps[code]
value collatz 23
```

Tail-recursive functions:

```
partial-function (tailrec) fixpoint :: ('a ⇒ 'a) ⇒ 'a ⇒ 'a
where
  fixpoint f x = (if f x = x then x else fixpoint f (f x))
```

8.9 Regression tests

The following examples mainly serve as tests for the function package.

```
fun listlen :: 'a list ⇒ nat
where
  listlen [] = 0
| listlen (x#xs) = Suc (listlen xs)
```

8.9.1 Context recursion

```
fun f :: nat ⇒ nat
where
  zero: f 0 = 0
| succ: f (Suc n) = (if f n = 0 then 0 else f n)
```

8.9.2 A combination of context and nested recursion

```
function h :: nat ⇒ nat
where
  h 0 = 0
| h (Suc n) = (if h n = 0 then h (h n) else h n)
⟨proof⟩
```

8.9.3 Context, but no recursive call

```
fun i :: nat ⇒ nat
where
  i 0 = 0
| i (Suc n) = (if n = 0 then 0 else i n)
```

8.9.4 Tupled nested recursion

```
fun fa :: nat ⇒ nat ⇒ nat
where
  fa 0 y = 0
| fa (Suc n) y = (if fa n y = 0 then 0 else fa n y)
```

8.9.5 Let

fun *j* :: *nat* ⇒ *nat*
where
  j 0 = *0*
| *j* (*Suc n*) = (*let u* = *n in Suc* (*j u*))

There were some problems with fresh names ...

function *k* :: *nat* ⇒ *nat*
where
  k x = (*let a* = *x*; *b* = *x in k x*)
  ⟨*proof*⟩

function *f2* :: (*nat* × *nat*) ⇒ (*nat* × *nat*)
where
  f2 p = (*let* (*x*,*y*) = *p in f2* (*y*,*x*))
  ⟨*proof*⟩

8.9.6 Abbreviations

fun *f3* :: *'a set* ⇒ *bool*
where
  f3 x = *finite x*

8.9.7 Simple Higher-Order Recursion

datatype *'a tree* = *Leaf 'a* | *Branch 'a tree list*

fun *treemap* :: (*'a* ⇒ *'a*) ⇒ *'a tree* ⇒ *'a tree*
where
  treemap fn (*Leaf n*) = (*Leaf* (*fn n*))
| *treemap fn* (*Branch l*) = (*Branch* (*map* (*treemap fn*) *l*))

fun *tinc* :: *nat tree* ⇒ *nat tree*
where
  tinc (*Leaf n*) = *Leaf* (*Suc n*)
| *tinc* (*Branch l*) = *Branch* (*map tinc l*)

fun *testcase* :: *'a tree* ⇒ *'a list*
where
  testcase (*Leaf a*) = [*a*]
| *testcase* (*Branch x*) =
    (*let xs* = *concat* (*map testcase x*);
        ys = *concat* (*map testcase x*) *in*
     xs @ *ys*)

8.9.8 Pattern matching on records

record *point* =

Xcoord :: *int*
Ycoord :: *int*

function *swp* :: *point* ⇒ *point*
where
swp (| *Xcoord* = *x*, *Ycoord* = *y* |) = (| *Xcoord* = *y*, *Ycoord* = *x* |)
⟨*proof*⟩
termination ⟨*proof*⟩

8.9.9 The diagonal function

fun *diag* :: *bool* ⇒ *bool* ⇒ *bool* ⇒ *nat*
where
diag x True False = *1*
| *diag False y True* = *2*
| *diag True False z* = *3*
| *diag True True True* = *4*
| *diag False False False* = *5*

8.9.10 Many equations (quadratic blowup)

datatype *DT* =
A | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *P*
| *Q* | *R* | *S* | *T* | *U* | *V*

fun *big* :: *DT* ⇒ *nat*
where
big A = *0*
| *big B* = *0*
| *big C* = *0*
| *big D* = *0*
| *big E* = *0*
| *big F* = *0*
| *big G* = *0*
| *big H* = *0*
| *big I* = *0*
| *big J* = *0*
| *big K* = *0*
| *big L* = *0*
| *big M* = *0*
| *big N* = *0*
| *big P* = *0*
| *big Q* = *0*
| *big R* = *0*
| *big S* = *0*
| *big T* = *0*
| *big U* = *0*
| *big V* = *0*

8.9.11 Automatic pattern splitting

fun *f4* :: *nat* ⇒ *nat* ⇒ *bool*
where
  f4 0 0 = *True*
| *f4* - - = *False*

8.9.12 Polymorphic partial-function

partial-function (*option*) *f5* :: *'a list* ⇒ *'a option*
where
  f5 x = *f5 x*

end

9 Examples of automatically derived induction rules

theory *Induction-Schema*
imports *Main*
begin

9.1 Some simple induction principles on nat

lemma *nat-standard-induct*:
  ⟦*P 0*; ⋀*n*. *P n* ⟹ *P* (*Suc n*)⟧ ⟹ *P x*
⟨*proof*⟩

lemma *nat-induct2*:
  ⟦ *P 0*; *P* (*Suc 0*); ⋀*k*. *P k* ==> *P* (*Suc k*) ==> *P* (*Suc* (*Suc k*)) ⟧
  ⟹ *P n*
⟨*proof*⟩

lemma *minus-one-induct*:
  ⟦⋀*n*::*nat*. (*n* ≠ *0* ⟹ *P* (*n* − *1*)) ⟹ *P n*⟧ ⟹ *P x*
⟨*proof*⟩

theorem *diff-induct*:
  (!!*x*. *P x 0*) ==> (!!*y*. *P 0* (*Suc y*)) ==>
    (!!*x y*. *P x y* ==> *P* (*Suc x*) (*Suc y*)) ==> *P m n*
⟨*proof*⟩

lemma *list-induct2'*:
  ⟦ *P* [] []⟧;
  ⋀*x xs*. *P* (*x*#*xs*) []⟧;
  ⋀*y ys*. *P* [] (*y*#*ys*)⟧;
   ⋀*x xs y ys*. *P xs ys* ⟹ *P* (*x*#*xs*) (*y*#*ys*) ⟧
 ⟹ *P xs ys*
⟨*proof*⟩

theorem *even-odd-induct*:

assumes $R\ 0$
assumes $Q\ 0$
assumes $\bigwedge n.\ Q\ n \Longrightarrow R\ (Suc\ n)$
assumes $\bigwedge n.\ R\ n \Longrightarrow Q\ (Suc\ n)$
shows $R\ n\ Q\ n$
⟨*proof*⟩

end

10 Test of Locale Interpretation

theory *LocaleTest2*
imports *Main GCD*
begin

11 Interpretation of Defined Concepts

Naming convention for global objects: prefixes D and d

11.1 Lattices

Much of the lattice proofs are from HOL/Lattice.

11.1.1 Definitions

locale *dpo* =
fixes *le* :: ['*a*, '*a*] => *bool* (**infixl** $\sqsubseteq$ *50*)
assumes *refl* [*intro*, *simp*]: $x \sqsubseteq x$
and *antisym* [*intro*]: [| $x \sqsubseteq y$; $y \sqsubseteq x$ |] ==> $x = y$
and *trans* [*trans*]: [| $x \sqsubseteq y$; $y \sqsubseteq z$ |] ==> $x \sqsubseteq z$

begin

theorem *circular*:
[| $x \sqsubseteq y$; $y \sqsubseteq z$; $z \sqsubseteq x$ |] ==> $x = y$ & $y = z$
⟨*proof*⟩

definition
less :: ['*a*, '*a*] => *bool* (**infixl** $\sqsubset$ *50*)
where $(x \sqsubset y) = (x \sqsubseteq y\ \&\ x \sim= y)$

theorem *abs-test*:
$op \sqsubset = (\%x\ y.\ x \sqsubset y)$
⟨*proof*⟩

definition
is-inf :: ['*a*, '*a*, '*a*] => *bool*

where *is-inf x y i* = $(i \sqsubseteq x \wedge i \sqsubseteq y \wedge (\forall z.\ z \sqsubseteq x \wedge z \sqsubseteq y \longrightarrow z \sqsubseteq i))$

definition
is-sup :: [*'a*, *'a*, *'a*] => *bool*
where *is-sup x y s* = $(x \sqsubseteq s \wedge y \sqsubseteq s \wedge (\forall z.\ x \sqsubseteq z \wedge y \sqsubseteq z \longrightarrow s \sqsubseteq z))$

end

locale *dlat* = *dpo* +
assumes *ex-inf*: *EX inf*. *dpo.is-inf le x y inf*
and *ex-sup*: *EX sup*. *dpo.is-sup le x y sup*

begin

definition
meet :: [*'a*, *'a*] => *'a* (**infixl** $\sqcap$ *70*)
where $x \sqcap y$ = (*THE inf*. *is-inf x y inf*)

definition
join :: [*'a*, *'a*] => *'a* (**infixl** $\sqcup$ *65*)
where $x \sqcup y$ = (*THE sup*. *is-sup x y sup*)

lemma *is-infI* [*intro?*]: $i \sqsubseteq x \Longrightarrow i \sqsubseteq y \Longrightarrow$
$(\bigwedge z.\ z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq i) \Longrightarrow$ *is-inf x y i*
⟨*proof*⟩

lemma *is-inf-lower* [*elim?*]:
is-inf x y i $\Longrightarrow (i \sqsubseteq x \Longrightarrow i \sqsubseteq y \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

lemma *is-inf-greatest* [*elim?*]:
is-inf x y i $\Longrightarrow z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq i$
⟨*proof*⟩

theorem *is-inf-uniq*: *is-inf x y i* $\Longrightarrow$ *is-inf x y i'* $\Longrightarrow i = i'$
⟨*proof*⟩

theorem *is-inf-related* [*elim?*]: $x \sqsubseteq y \Longrightarrow$ *is-inf x y x*
⟨*proof*⟩

lemma *meet-equality* [*elim?*]: *is-inf x y i* $\Longrightarrow x \sqcap y = i$
⟨*proof*⟩

lemma *meetI* [*intro?*]:
$i \sqsubseteq x \Longrightarrow i \sqsubseteq y \Longrightarrow (\bigwedge z.\ z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq i) \Longrightarrow x \sqcap y = i$
⟨*proof*⟩

lemma *is-inf-meet* [*intro?*]: *is-inf x y* $(x \sqcap y)$
⟨*proof*⟩

lemma *meet-left* [*intro?*]:
$x \sqcap y \sqsubseteq x$
⟨*proof*⟩

lemma *meet-right* [*intro?*]:
$x \sqcap y \sqsubseteq y$
⟨*proof*⟩

lemma *meet-le* [*intro?*]:
$[\!| z \sqsubseteq x; z \sqsubseteq y |\!] ==> z \sqsubseteq x \sqcap y$
⟨*proof*⟩

lemma *is-supI* [*intro?*]: $x \sqsubseteq s \Longrightarrow y \sqsubseteq s \Longrightarrow$
$(\bigwedge z.\ x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow s \sqsubseteq z) \Longrightarrow$ *is-sup* $x\ y\ s$
⟨*proof*⟩

lemma *is-sup-least* [*elim?*]:
is-sup $x\ y\ s \Longrightarrow x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow s \sqsubseteq z$
⟨*proof*⟩

lemma *is-sup-upper* [*elim?*]:
is-sup $x\ y\ s \Longrightarrow (x \sqsubseteq s \Longrightarrow y \sqsubseteq s \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

theorem *is-sup-uniq*: *is-sup* $x\ y\ s \Longrightarrow$ *is-sup* $x\ y\ s' \Longrightarrow s = s'$
⟨*proof*⟩

theorem *is-sup-related* [*elim?*]: $x \sqsubseteq y \Longrightarrow$ *is-sup* $x\ y\ y$
⟨*proof*⟩

lemma *join-equality* [*elim?*]: *is-sup* $x\ y\ s \Longrightarrow x \sqcup y = s$
⟨*proof*⟩

lemma *joinI* [*intro?*]: $x \sqsubseteq s \Longrightarrow y \sqsubseteq s \Longrightarrow$
$(\bigwedge z.\ x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow s \sqsubseteq z) \Longrightarrow x \sqcup y = s$
⟨*proof*⟩

lemma *is-sup-join* [*intro?*]: *is-sup* $x\ y\ (x \sqcup y)$
⟨*proof*⟩

lemma *join-left* [*intro?*]:
$x \sqsubseteq x \sqcup y$
⟨*proof*⟩

lemma *join-right* [*intro?*]:
$y \sqsubseteq x \sqcup y$
⟨*proof*⟩

lemma *join-le* [*intro?*]:
[| $x \sqsubseteq z$; $y \sqsubseteq z$ |] ==> $x \sqcup y \sqsubseteq z$
⟨*proof*⟩

theorem *meet-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
⟨*proof*⟩

theorem *meet-commute*: $x \sqcap y = y \sqcap x$
⟨*proof*⟩

theorem *meet-join-absorb*: $x \sqcap (x \sqcup y) = x$
⟨*proof*⟩

theorem *join-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
⟨*proof*⟩

theorem *join-commute*: $x \sqcup y = y \sqcup x$
⟨*proof*⟩

theorem *join-meet-absorb*: $x \sqcup (x \sqcap y) = x$
⟨*proof*⟩

theorem *meet-idem*: $x \sqcap x = x$
⟨*proof*⟩

theorem *meet-related* [*elim?*]: $x \sqsubseteq y \Longrightarrow x \sqcap y = x$
⟨*proof*⟩

theorem *meet-related2* [*elim?*]: $y \sqsubseteq x \Longrightarrow x \sqcap y = y$
⟨*proof*⟩

theorem *join-related* [*elim?*]: $x \sqsubseteq y \Longrightarrow x \sqcup y = y$
⟨*proof*⟩

theorem *join-related2* [*elim?*]: $y \sqsubseteq x \Longrightarrow x \sqcup y = x$
⟨*proof*⟩

Additional theorems

theorem *meet-connection*: $(x \sqsubseteq y) = (x \sqcap y = x)$
⟨*proof*⟩

theorem *meet-connection2*: $(x \sqsubseteq y) = (y \sqcap x = x)$
⟨*proof*⟩

theorem *join-connection*: $(x \sqsubseteq y) = (x \sqcup y = y)$
⟨*proof*⟩

theorem *join-connection2*: $(x \sqsubseteq y) = (x \sqcup y = y)$
⟨*proof*⟩

Naming according to Jacobson I, p. 459.

lemmas *L1* = *join-commute meet-commute*
lemmas *L2* = *join-assoc meet-assoc*

lemmas *L4* = *join-meet-absorb meet-join-absorb*

end

locale *ddlat* = *dlat* +
  assumes *meet-distr*:
    dlat.meet le x (*dlat.join le y z*) =
    dlat.join le (*dlat.meet le x y*) (*dlat.meet le x z*)

begin

lemma *join-distr*:
  $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

Jacobson I, p. 462

⟨*proof*⟩

end

locale *dlo* = *dpo* +
  assumes *total*: $x \sqsubseteq y \mid y \sqsubseteq x$

begin

lemma *less-total*: $x \sqsubset y \mid x = y \mid y \sqsubset x$
  ⟨*proof*⟩

end

sublocale *dlo* < *dlat*
⟨*proof*⟩

sublocale *dlo* < *ddlat*
⟨*proof*⟩

11.1.2 Total order <= on *int*

interpretation *int*: *dpo op* <= :: [*int*, *int*] => *bool*
  rewrites (*dpo.less* (*op* <=) (*x*::*int*) *y*) = ($x < y$)

We give interpretation for less, but not *is-inf* and *is-sub*.

⟨*proof*⟩

thm *int.circular*

lemma $[\![(x::int) \leq y; y \leq z; z \leq x]\!] \Longrightarrow x = y \wedge y = z$
⟨*proof*⟩
thm *int.abs-test*
lemma $(op < :: [int, int] => bool) = op <$
⟨*proof*⟩

interpretation *int*: *dlat op <=* :: [*int*, *int*] => *bool*
rewrites *meet-eq*: *dlat.meet* $(op <=)$ $(x::int)$ $y = min\ x\ y$
and *join-eq*: *dlat.join* $(op <=)$ $(x::int)$ $y = max\ x\ y$
⟨*proof*⟩

interpretation *int*: *dlo op <=* :: [*int*, *int*] => *bool*
⟨*proof*⟩

Interpreted theorems from the locales, involving defined terms.

thm *int.less-def*

from dpo

thm *int.meet-left*

from dlat

thm *int.meet-distr*

from ddlat

thm *int.less-total*

from dlo

11.1.3 Total order <= on *nat*

interpretation *nat*: *dpo op <=* :: [*nat*, *nat*] => *bool*
rewrites *dpo.less* $(op <=)$ $(x::nat)$ $y = (x < y)$

We give interpretation for less, but not *is-inf* and *is-sub*.

⟨*proof*⟩

interpretation *nat*: *dlat op <=* :: [*nat*, *nat*] => *bool*
rewrites *dlat.meet* $(op <=)$ $(x::nat)$ $y = min\ x\ y$
and *dlat.join* $(op <=)$ $(x::nat)$ $y = max\ x\ y$
⟨*proof*⟩

interpretation *nat*: *dlo op <=* :: [*nat*, *nat*] => *bool*
⟨*proof*⟩

Interpreted theorems from the locales, involving defined terms.

thm *nat.less-def*

from dpo

thm *nat.meet-left*

from dlat

thm *nat.meet-distr*

from ddlat

thm *nat.less-total*

from ldo

11.1.4 Lattice *dvd* on *nat*

interpretation *nat-dvd*: *dpo op dvd* :: [*nat*, *nat*] => *bool*
  rewrites *dpo.less* (*op dvd*) (*x*::*nat*) *y* = (*x dvd y* & *x* ~= *y*)

We give interpretation for less, but not *is-inf* and *is-sub*.

⟨*proof*⟩

interpretation *nat-dvd*: *dlat op dvd* :: [*nat*, *nat*] => *bool*
  rewrites *dlat.meet* (*op dvd*) (*x*::*nat*) *y* = *gcd x y*
    and *dlat.join* (*op dvd*) (*x*::*nat*) *y* = *lcm x y*
⟨*proof*⟩

Interpreted theorems from the locales, involving defined terms.

thm *nat-dvd.less-def*

from dpo

lemma ((*x*::*nat*) *dvd y* & *x* ~= *y*) = (*x dvd y* & *x* ~= *y*)
  ⟨*proof*⟩
thm *nat-dvd.meet-left*

from dlat

lemma *gcd x y dvd* (*x*::*nat*)
  ⟨*proof*⟩

11.2 Group example with defined operations *inv* and *unit*

11.2.1 Locale declarations and lemmas

locale *Dsemi* =
  fixes *prod* (**infixl** ** 65*)
  assumes *assoc*: (*x* ** *y*) ** *z* = *x* ** (*y* ** *z*)

locale *Dmonoid* = *Dsemi* +
  fixes *one*
  assumes *l-one* [*simp*]: *one* ** *x* = *x*
    and *r-one* [*simp*]: *x* ** *one* = *x*

begin

definition
inv **where** $inv\ x = (THE\ y.\ x\ **\ y = one\ \&\ y\ **\ x = one)$
definition
unit **where** $unit\ x = (EX\ y.\ x\ **\ y = one\ \ \&\ y\ **\ x = one)$

lemma *inv-unique*:
assumes *eq*: $y\ **\ x = one\ x\ **\ y' = one$
shows $y = y'$
⟨*proof*⟩

lemma *unit-one* [*intro*, *simp*]:
$unit\ one$
⟨*proof*⟩

lemma *unit-l-inv-ex*:
$unit\ x\ ==>\ \exists\, y.\ y\ **\ x = one$
⟨*proof*⟩

lemma *unit-r-inv-ex*:
$unit\ x\ ==>\ \exists\, y.\ x\ **\ y = one$
⟨*proof*⟩

lemma *unit-l-inv*:
$unit\ x\ ==>\ inv\ x\ **\ x = one$
⟨*proof*⟩

lemma *unit-r-inv*:
$unit\ x\ ==>\ x\ **\ inv\ x = one$
⟨*proof*⟩

lemma *unit-inv-unit* [*intro*, *simp*]:
$unit\ x\ ==>\ unit\ (inv\ x)$
⟨*proof*⟩

lemma *unit-l-cancel* [*simp*]:
$unit\ x\ ==>\ (x\ **\ y = x\ **\ z) = (y = z)$
⟨*proof*⟩

lemma *unit-inv-inv* [*simp*]:
$unit\ x\ ==>\ inv\ (inv\ x) = x$
⟨*proof*⟩

lemma *inv-inj-on-unit*:
$inj\text{-}on\ inv\ \{x.\ unit\ x\}$
⟨*proof*⟩

lemma *unit-inv-comm*:

assumes *inv*: $x ** y = one$
and *G*: *unit x* *unit y*
shows $y ** x = one$
⟨*proof*⟩

end

locale *Dgrp* = *Dmonoid* +
assumes *unit* [*intro*, *simp*]: *Dmonoid.unit* (*op* **) *one x*

begin

lemma *l-inv-ex* [*simp*]:
$\exists y.\ y ** x = one$
⟨*proof*⟩

lemma *r-inv-ex* [*simp*]:
$\exists y.\ x ** y = one$
⟨*proof*⟩

lemma *l-inv* [*simp*]:
inv x ** *x* = *one*
⟨*proof*⟩

lemma *l-cancel* [*simp*]:
$(x ** y = x ** z) = (y = z)$
⟨*proof*⟩

lemma *r-inv* [*simp*]:
$x ** inv\ x = one$
⟨*proof*⟩

lemma *r-cancel* [*simp*]:
$(y ** x = z ** x) = (y = z)$
⟨*proof*⟩

lemma *inv-one* [*simp*]:
inv one = *one*
⟨*proof*⟩

lemma *inv-inv* [*simp*]:
inv (*inv x*) = *x*
⟨*proof*⟩

lemma *inv-inj*:
inj-on inv UNIV
⟨*proof*⟩

lemma *inv-mult-group*:
inv (*x* ** *y*) = *inv y* ** *inv x*
⟨*proof*⟩

lemma *inv-comm*:
x ** *y* = *one* ==> *y* ** *x* = *one*
⟨*proof*⟩

lemma *inv-equality*:
y ** *x* = *one* ==> *inv x* = *y*
⟨*proof*⟩

end

locale *Dhom* = *prod*: *Dgrp prod one* + *sum*: *Dgrp sum zero*
for *prod* (**infixl** ** 65) **and** *one* **and** *sum* (**infixl** +++ 60) **and** *zero* +
fixes *hom*
assumes *hom-mult* [*simp*]: *hom* (*x* ** *y*) = *hom x* +++ *hom y*

begin

lemma *hom-one* [*simp*]:
hom one = *zero*
⟨*proof*⟩

end

11.2.2 Interpretation of Functions

interpretation *Dfun*: *Dmonoid op o id* :: 'a => 'a
rewrites *Dmonoid.unit* (*op o*) *id f* = *bij* (*f*::'a => 'a)

⟨*proof*⟩

thm *Dmonoid.unit-def Dfun.unit-def*

thm *Dmonoid.inv-inj-on-unit Dfun.inv-inj-on-unit*

lemma *unit-id*:
(*f* :: *unit* => *unit*) = *id*
⟨*proof*⟩

interpretation *Dfun*: *Dgrp op o id* :: *unit* => *unit*
rewrites *Dmonoid.inv* (*op o*) *id f* = *inv* (*f* :: *unit* => *unit*)
⟨*proof*⟩

thm *Dfun.unit-l-inv Dfun.l-inv*

thm *Dfun.inv-equality*
thm *Dfun.inv-equality*

end

12 Using extensible records in HOL – points and coloured points

theory *Records*
imports *Main*
begin

12.1 Points

record *point* =
  xpos :: *nat*
  ypos :: *nat*

Apart many other things, above record declaration produces the following theorems:

thm *point.simps*
thm *point.iffs*
thm *point.defs*

The set of theorems *point.simps* is added automatically to the standard simpset, *point.iffs* is added to the Classical Reasoner and Simplifier context.

Record declarations define new types and type abbreviations:

point = (|*xpos* :: *nat*, *ypos* :: *nat*|) = () *point-ext-type*
'*a point-scheme* = (|*xpos* :: *nat*, *ypos* :: *nat*, ... :: '*a*|) = '*a point-ext-type*

consts *foo2* :: (| *xpos* :: *nat*, *ypos* :: *nat* |)
consts *foo4* :: '*a* => (| *xpos* :: *nat*, *ypos* :: *nat*, ... :: '*a* |)

12.1.1 Introducing concrete records and record schemes

definition *foo1* :: *point*
  where *foo1* = (| *xpos* = *1*, *ypos* = *0* |)

definition *foo3* :: '*a* => '*a point-scheme*
  where *foo3 ext* = (| *xpos* = *1*, *ypos* = *0*, ... = *ext* |)

12.1.2 Record selection and record update

definition *getX* :: '*a point-scheme* => *nat*
  where *getX r* = *xpos r*

definition *setX* :: '*a point-scheme* => *nat* => '*a point-scheme*
  where *setX r n* = *r* (| *xpos* := *n* |)

12.1.3 Some lemmas about records

Basic simplifications.

lemma *point.make n p* = (| *xpos* = *n*, *ypos* = *p* |)
⟨*proof*⟩

lemma *xpos* (| *xpos* = *m*, *ypos* = *n*, ... = *p* |) = *m*
⟨*proof*⟩

lemma (| *xpos* = *m*, *ypos* = *n*, ... = *p* |) (| *xpos*:= *0* |) = (| *xpos* = *0*, *ypos* = *n*, ... = *p* |)
⟨*proof*⟩

Equality of records.

lemma *n* = *n'* ==> *p* = *p'* ==> (| *xpos* = *n*, *ypos* = *p* |) = (| *xpos* = *n'*, *ypos* = *p'* |)
— introduction of concrete record equality
⟨*proof*⟩

lemma (| *xpos* = *n*, *ypos* = *p* |) = (| *xpos* = *n'*, *ypos* = *p'* |) ==> *n* = *n'*
— elimination of concrete record equality
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) (| *ypos* := *m* |) = *r* (| *ypos* := *m* |) (| *xpos* := *n* |)
— introduction of abstract record equality
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) = *r* (| *xpos* := *n'* |) ==> *n* = *n'*
— elimination of abstract record equality (manual proof)
⟨*proof*⟩

Surjective pairing

lemma *r* = (| *xpos* = *xpos r*, *ypos* = *ypos r* |)
⟨*proof*⟩

lemma *r* = (| *xpos* = *xpos r*, *ypos* = *ypos r*, ... = *point.more r* |)
⟨*proof*⟩

Representation of records by cases or (degenerate) induction.

lemma *r*(| *xpos* := *n* |) (| *ypos* := *m* |) = *r* (| *ypos* := *m* |) (| *xpos* := *n* |)
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) (| *ypos* := *m* |) = *r* (| *ypos* := *m* |) (| *xpos* := *n* |)
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) (| *xpos* := *m* |) = *r* (| *xpos* := *m* |)
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) (| *xpos* := *m* |) = *r* (| *xpos* := *m* |)
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) (| *xpos* := *m* |) = *r* (| *xpos* := *m* |)
⟨*proof*⟩

Concrete records are type instances of record schemes.

definition *foo5* :: *nat*
where *foo5* = *getX* (| *xpos* = *1*, *ypos* = *0* |)

Manipulating the "..." (more) part.

definition *incX* :: *'a point-scheme* => *'a point-scheme*
where *incX r* = (| *xpos* = *xpos r* + *1*, *ypos* = *ypos r*, ... = *point.more r* |)

lemma *incX r* = *setX r* (*Suc* (*getX r*))
⟨*proof*⟩

An alternative definition.

definition *incX'* :: *'a point-scheme* => *'a point-scheme*
where *incX' r* = *r* (| *xpos* := *xpos r* + *1* |)

12.2 Coloured points: record extension

datatype *colour* = *Red* | *Green* | *Blue*

record *cpoint* = *point* +
colour :: *colour*

The record declaration defines a new type constructor and abbreviations:

cpoint = (| *xpos* :: *nat*, *ypos* :: *nat*, *colour* :: *colour* |) =
() *cpoint-ext-type point-ext-type*
'a cpoint-scheme = (| *xpos* :: *nat*, *ypos* :: *nat*, *colour* :: *colour*, ... :: *'a* |) =
'a cpoint-ext-type point-ext-type

consts *foo6* :: *cpoint*
consts *foo7* :: (| *xpos* :: *nat*, *ypos* :: *nat*, *colour* :: *colour* |)
consts *foo8* :: *'a cpoint-scheme*
consts *foo9* :: (| *xpos* :: *nat*, *ypos* :: *nat*, *colour* :: *colour*, ... :: *'a* |)

Functions on *point* schemes work for *cpoints* as well.

definition *foo10* :: *nat*
where *foo10* = *getX* (| *xpos* = *2*, *ypos* = *0*, *colour* = *Blue* |)

12.2.1 Non-coercive structural subtyping

Term *foo11* has type *cpoint*, not type *point* — Great!

definition *foo11* :: *cpoint*
  where *foo11* = *setX* (| *xpos* = *2*, *ypos* = *0*, *colour* = *Blue* |) *0*

12.3 Other features

Field names contribute to record identity.

record *point′* =
  xpos′ :: *nat*
  ypos′ :: *nat*

May not apply *getX* to (| *xpos′* = *2*, *ypos′* = *0* |) – type error.

Polymorphic records.

record *'a point″* = *point* +
  content :: *'a*

type-synonym *cpoint″* = *colour point″*

Updating a record field with an identical value is simplified.

lemma *r* (| *xpos* := *xpos r* |) = *r*
  ⟨*proof*⟩

Only the most recent update to a component survives simplification.

lemma *r* (| *xpos* := *x*, *ypos* := *y*, *xpos* := *x′* |) = *r* (| *ypos* := *y*, *xpos* := *x′* |)
  ⟨*proof*⟩

In some cases its convenient to automatically split (quantified) records. For this purpose there is the simproc `Record.split_simproc` and the tactic `Record.split_simp_tac`. The simplification procedure only splits the records, whereas the tactic also simplifies the resulting goal with the standard record simplification rules. A (generalized) predicate on the record is passed as parameter that decides whether or how 'deep' to split the record. It can peek on the subterm starting at the quantified occurrence of the record (including the quantifier). The value `0` indicates no split, a value greater `0` splits up to the given bound of record extension and finally the value `~1` completely splits the record. `Record.split_simp_tac` additionally takes a list of equations for simplification and can also split fixed record variables.

lemma (∀ *r*. *P* (*xpos r*)) ⟶ (∀ *x*. *P x*)
  ⟨*proof*⟩

lemma (∀ *r*. *P* (*xpos r*)) ⟶ (∀ *x*. *P x*)
  ⟨*proof*⟩

lemma $(\exists\, r.\ P\ (xpos\ r)) \longrightarrow (\exists\, x.\ P\ x)$
⟨*proof*⟩

lemma $(\exists\, r.\ P\ (xpos\ r)) \longrightarrow (\exists\, x.\ P\ x)$
⟨*proof*⟩

lemma $\bigwedge r.\ P\ (xpos\ r) \Longrightarrow (\exists\, x.\ P\ x)$
⟨*proof*⟩

lemma $\bigwedge r.\ P\ (xpos\ r) \Longrightarrow (\exists\, x.\ P\ x)$
⟨*proof*⟩

lemma $P\ (xpos\ r) \Longrightarrow (\exists\, x.\ P\ x)$
⟨*proof*⟩

lemma *True*
⟨*proof*⟩

The effect of simproc `Record.ex_sel_eq_simproc` is illustrated by the following lemma.

lemma $\exists\, r.\ xpos\ r = x$
⟨*proof*⟩

12.4 A more complex record expression

record $('a,\ 'b,\ 'c)$ *bar* = *bar1* :: $'a$
bar2 :: $'b$
bar3 :: $'c$
bar21 :: $'b \times 'a$
bar32 :: $'c \times 'b$
bar31 :: $'c \times 'a$

print-record $('a,'b,'c)$ *bar*

12.5 Some code generation

export-code *foo1 foo3 foo5 foo10* **checking** *SML*

Code generation can also be switched off, for instance for very large records

declare [[*record-codegen* = *false*]]

record *not-so-large-record* =
bar520 :: *nat*
bar521 :: *nat* ∗ *nat*

declare [[*record-codegen* = *true*]]

end

13 A general "while" combinator

theory *While-Combinator*
imports *Main*
begin

13.1 Partial version

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

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

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

lemma *while-option-stop*: *while-option b c s* = *Some t* $\Longrightarrow$ $\sim\ b\ t$
⟨*proof*⟩

theorem *while-option-rule*:
assumes *step*: !!*s*. *P s* ==> *b s* ==> *P* (*c s*)
and *result*: *while-option b c s* = *Some t*
and *init*: *P s*
shows *P t*
⟨*proof*⟩

lemma *funpow-commute*:
$[\![\forall k' < k.\ f\ (c\ ((c\hat{}\hat{}k')\ s)) = c'\ (f\ ((c\hat{}\hat{}k')\ s))]\!] \Longrightarrow f\ ((c\hat{}\hat{}k)\ s) = (c'\hat{}\hat{}k)\ (f\ s)$
⟨*proof*⟩

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

lemma *while-option-commute*:
assumes $\bigwedge s.\ b\ s = b'\ (f\ s)$ $\bigwedge s.\ [\![b\ s]\!] \Longrightarrow f\ (c\ s) = c'\ (f\ s)$
shows *map-option f* (*while-option b c s*) = *while-option b' c'* (*f s*)
⟨*proof*⟩

13.2 Total version

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

lemma *while-unfold* [*code*]:
while b c s = (*if b s then while b c* (*c s*) *else s*)
⟨*proof*⟩

lemma *def-while-unfold*:
assumes *fdef*: *f* == *while test do*
shows *f x* = (*if test x then f*(*do x*) *else x*)
⟨*proof*⟩

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

theorem *while-rule-lemma*:
assumes *invariant*: !!*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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

Proving termination:

theorem *wf-while-option-Some*:
assumes *wf* {(*t*, *s*). (*P s* ∧ *b s*) ∧ *t* = *c s*}
and !!*s*. *P s* ⟹ *b s* ⟹ *P*(*c s*) **and** *P s*
shows *EX t*. *while-option b c s* = *Some t*
⟨*proof*⟩

lemma *wf-rel-while-option-Some*:
assumes *wf*: *wf R*
assumes *smaller*: ⋀*s*. *P s* ∧ *b s* ⟹ (*c s*, *s*) ∈ *R*
assumes *inv*: ⋀*s*. *P s* ∧ *b s* ⟹ *P*(*c s*)
assumes *init*: *P s*
shows ∃ *t*. *while-option b c s* = *Some t*
⟨*proof*⟩

theorem *measure-while-option-Some*: **fixes** *f* :: *'s* ⇒ *nat*
shows (!!*s*. *P s* ⟹ *b s* ⟹ *P*(*c s*) ∧ *f*(*c s*) < *f s*)
⟹ *P s* ⟹ *EX t*. *while-option b c s* = *Some t*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

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

lemma *gfp-while-lattice*:
fixes $f :: 'a$::*complete-lattice* $\Rightarrow 'a$
assumes *mono f* **and** *finite* (*UNIV* :: $'a$ *set*)
shows *gfp f* = *while* ($\lambda A.$ *f A* $\neq$ *A*) *f top*
⟨*proof*⟩

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

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

context
fixes $p :: 'a \Rightarrow$ *bool*
and $f :: 'a \Rightarrow 'a$ *list*
and $x :: 'a$
begin

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

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

definition *rtrancl-while* :: ($'a$ *list* $*$ $'a$ *set*) *option*
where *rtrancl-while* = *while-option rtrancl-while-test rtrancl-while-step* ([*x*],{*x*})

qualified fun *rtrancl-while-invariant* :: $'a$ *list* $\times$ $'a$ *set* $\Rightarrow$ *bool*
where *rtrancl-while-invariant* (*ws*, *Z*) =
(*x* $\in$ *Z* $\wedge$ *set ws* $\subseteq$ *Z* $\wedge$ *distinct ws* $\wedge$ {(*x*,*y*). *y* $\in$ *set*(*f x*)} `` (*Z* $-$ *set ws*) $\subseteq$ *Z* $\wedge$
Z $\subseteq$ {(*x*,*y*). *y* $\in$ *set*(*f x*)}$\hat{}*$ `` {*x*} $\wedge$ ($\forall$ *z*$\in$*Z* $-$ *set ws*. *p z*))

qualified lemma *rtrancl-while-invariant*:
assumes *inv*: *rtrancl-while-invariant st* **and** *test*: *rtrancl-while-test st*
shows *rtrancl-while-invariant* (*rtrancl-while-step st*)
⟨*proof*⟩

lemma *rtrancl-while-Some*: **assumes** *rtrancl-while* = *Some*(*ws*,*Z*)
shows *if ws* = []

then $Z = \{(x,y).\ y \in set(f\ x)\}\hat{}*\ ``\ \{x\} \wedge (\forall z{\in}Z.\ p\ z)$
else $\neg p(hd\ ws) \wedge hd\ ws \in \{(x,y).\ y \in set(f\ x)\}\hat{}*\ ``\ \{x\}$
⟨*proof*⟩

lemma *rtrancl-while-finite-Some*:
assumes *finite* $(\{(x,\ y).\ y \in set\ (f\ x)\}^*\ ``\ \{x\})$ (**is** *finite ?Cl*)
shows $\exists y.$ *rtrancl-while* = *Some y*
⟨*proof*⟩

end

end

14 An application of the While combinator

theory *While-Combinator-Example*
imports ~~/*src*/*HOL*/*Library*/*While-Combinator*
begin

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* {}))
⟨*proof*⟩

14.1 Example

Cannot use *set-eq-subset* because it leads to looping because the antisymmetry 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*⟩

end

15 Monoids and Groups as predicates over record schemes

theory *MonoidGroup* **imports** *Main* **begin**

record *'a monoid-sig* =
times :: *'a* => *'a* => *'a*
one :: *'a*

record *'a group-sig* = *'a monoid-sig* +
inv :: *'a* => *'a*

definition
monoid :: (| *times* :: *'a* => *'a* => *'a*, *one* :: *'a*, ... :: *'b* |) => *bool* **where**
monoid M = (∀ *x y z*.
times M (*times M x y*) *z* = *times M x* (*times M y z*) ∧
times M (*one M*) *x* = *x* ∧ *times M x* (*one M*) = *x*)

definition
group :: (| *times* :: *'a* => *'a* => *'a*, *one* :: *'a*, *inv* :: *'a* => *'a*, ... :: *'b* |) => *bool*
where
group G = (*monoid G* ∧ (∀ *x*. *times G* (*inv G x*) *x* = *one G*))

definition
reverse :: (| *times* :: *'a* => *'a* => *'a*, *one* :: *'a*, ... :: *'b* |) =>
(| *times* :: *'a* => *'a* => *'a*, *one* :: *'a*, ... :: *'b* |) **where**
reverse M = *M* (| *times* := λ*x y*. *times M y x* |)

end

16 Binary arithmetic examples

theory *BinEx*
imports *Complex-Main*
begin

16.1 Regression Testing for Cancellation Simprocs

lemma *l* + *2* + *2* + *2* + (*l* + *2*) + (*oo* + *2*) = (*uu*::*int*)
⟨*proof*⟩

lemma *2***u* = (*u*::*int*)
⟨*proof*⟩

lemma (*i* + *j* + *12* + (*k*::*int*)) − *15* = *y*
⟨*proof*⟩

lemma (*i* + *j* + *12* + (*k*::*int*)) − *5* = *y*
⟨*proof*⟩

lemma *y* − *b* < (*b*::*int*)
⟨*proof*⟩

lemma *y* − (*3***b* + *c*) < (*b*::*int*) − *2***c*
⟨*proof*⟩

lemma (*2***x* − (*u***v*) + *y*) − *v***3***u* = (*w*::*int*)
⟨*proof*⟩

lemma (*2***x***u***v* + (*u***v*)**4* + *y*) − *v***u***4* = (*w*::*int*)
⟨*proof*⟩

lemma $(2*x*u*v + (u*v)*4 + y) - v*u = (w::int)$
⟨*proof*⟩

lemma $u*v - (x*u*v + (u*v)*4 + y) = (w::int)$
⟨*proof*⟩

lemma $(i + j + 12 + (k::int)) = u + 15 + y$
⟨*proof*⟩

lemma $(i + j*2 + 12 + (k::int)) = j + 5 + y$
⟨*proof*⟩

lemma $2*y + 3*z + 6*w + 2*y + 3*z + 2*u = 2*y' + 3*z' + 6*w' + 2*y' + 3*z' + u + (vv::int)$
⟨*proof*⟩

lemma $a + -(b+c) + b = (d::int)$
⟨*proof*⟩

lemma $a + -(b+c) - b = (d::int)$
⟨*proof*⟩

lemma $(i + j + -2 + (k::int)) - (u + 5 + y) = zz$
⟨*proof*⟩

lemma $(i + j + -3 + (k::int)) < u + 5 + y$
⟨*proof*⟩

lemma $(i + j + 3 + (k::int)) < u + -6 + y$
⟨*proof*⟩

lemma $(i + j + -12 + (k::int)) - 15 = y$
⟨*proof*⟩

lemma $(i + j + 12 + (k::int)) - -15 = y$
⟨*proof*⟩

lemma $(i + j + -12 + (k::int)) - -15 = y$
⟨*proof*⟩

lemma $- (2*i) + 3 + (2*i + 4) = (0::int)$
⟨*proof*⟩

lemma $(pi * (real\ u * 2) = pi * (real\ (xa\ v) * - 2))$
⟨*proof*⟩

16.2 Arithmetic Method Tests

lemma !!a::int. [| $a <= b$; $c <= d$; $x+y<z$ |] ==> $a+c <= b+d$
⟨*proof*⟩

lemma !!a::int. [| $a < b$; $c < d$ |] ==> $a-d+ 2 <= b+(-c)$
⟨*proof*⟩

lemma !!a::int. [| $a < b$; $c < d$ |] ==> $a+c+ 1 < b+d$
⟨*proof*⟩

lemma !!a::int. [| $a <= b$; $b+b <= c$ |] ==> $a+a <= c$
⟨*proof*⟩

lemma !!a::int. [| $a+b <= i+j$; $a<=b$; $i<=j$ |] ==> $a+a <= j+j$
⟨*proof*⟩

lemma !!a::int. [| $a+b < i+j$; $a<b$; $i<j$ |] ==> $a+a - - (- 1) < j+j - 3$
⟨*proof*⟩

lemma !!a::int. $a+b+c <= i+j+k$ & $a<=b$ & $b<=c$ & $i<=j$ & $j<=k$ --> $a+a+a <= k+k+k$
⟨*proof*⟩

lemma !!a::int. [| $a+b+c+d <= i+j+k+l$; $a<=b$; $b<=c$; $c<=d$; $i<=j$; $j<=k$; $k<=l$ |]
    ==> $a <= l$
⟨*proof*⟩

lemma !!a::int. [| $a+b+c+d <= i+j+k+l$; $a<=b$; $b<=c$; $c<=d$; $i<=j$; $j<=k$; $k<=l$ |]
    ==> $a+a+a+a <= l+l+l+l$
⟨*proof*⟩

lemma !!a::int. [| $a+b+c+d <= i+j+k+l$; $a<=b$; $b<=c$; $c<=d$; $i<=j$; $j<=k$; $k<=l$ |]
    ==> $a+a+a+a+a <= l+l+l+l+i$
⟨*proof*⟩

lemma !!a::int. [| $a+b+c+d <= i+j+k+l$; $a<=b$; $b<=c$; $c<=d$; $i<=j$; $j<=k$; $k<=l$ |]
    ==> $a+a+a+a+a+a <= l+l+l+l+i+l$
⟨*proof*⟩

lemma !!a::int. [| $a+b+c+d <= i+j+k+l$; $a<=b$; $b<=c$; $c<=d$; $i<=j$; $j<=k$; $k<=l$ |]
    ==> $6*a <= 5*l+i$
⟨*proof*⟩

16.3 The Integers

Addition

lemma $(13::int) + 19 = 32$
⟨*proof*⟩

lemma $(1234::int) + 5678 = 6912$
⟨*proof*⟩

lemma $(1359::int) + -2468 = -1109$
⟨*proof*⟩

lemma $(93746::int) + -46375 = 47371$
⟨*proof*⟩

Negation

lemma $- (65745::int) = -65745$
⟨*proof*⟩

lemma $- (-54321::int) = 54321$
⟨*proof*⟩

Multiplication

lemma $(13::int) * 19 = 247$
⟨*proof*⟩

lemma $(-84::int) * 51 = -4284$
⟨*proof*⟩

lemma $(255::int) * 255 = 65025$
⟨*proof*⟩

lemma $(1359::int) * -2468 = -3354012$
⟨*proof*⟩

lemma $(89::int) * 10 \neq 889$
⟨*proof*⟩

lemma $(13::int) < 18 - 4$
⟨*proof*⟩

lemma $(-345::int) < -242 + -100$
⟨*proof*⟩

lemma $(13557456::int) < 18678654$
⟨*proof*⟩

lemma $(999999::int) \leq (1000001 + 1) - 2$

⟨*proof*⟩

lemma (*1234567*::*int*) ≤ *1234567*
⟨*proof*⟩

No integer overflow!

lemma *1234567* ∗ (*1234567*::*int*) < *1234567*∗*1234567*∗*1234567*
⟨*proof*⟩

Quotient and Remainder

lemma (*10*::*int*) *div 3* = *3*
⟨*proof*⟩

lemma (*10*::*int*) *mod 3* = *1*
⟨*proof*⟩

A negative divisor

lemma (*10*::*int*) *div* −*3* = −*4*
⟨*proof*⟩

lemma (*10*::*int*) *mod* −*3* = −*2*
⟨*proof*⟩

A negative dividend[1]

lemma (−*10*::*int*) *div 3* = −*4*
⟨*proof*⟩

lemma (−*10*::*int*) *mod 3* = *2*
⟨*proof*⟩

A negative dividend *and* divisor

lemma (−*10*::*int*) *div* −*3* = *3*
⟨*proof*⟩

lemma (−*10*::*int*) *mod* −*3* = −*1*
⟨*proof*⟩

A few bigger examples

lemma (*8452*::*int*) *mod 3* = *1*
⟨*proof*⟩

lemma (*59485*::*int*) *div 434* = *137*
⟨*proof*⟩

lemma (*1000006*::*int*) *mod 10* = *6*

[1]The definition agrees with mathematical convention and with ML, but not with the hardware of most computers

⟨*proof*⟩

Division by shifting

lemma *10000000 div 2 = (5000000::int)*
⟨*proof*⟩

lemma *10000001 mod 2 = (1::int)*
⟨*proof*⟩

lemma *10000055 div 32 = (312501::int)*
⟨*proof*⟩

lemma *10000055 mod 32 = (23::int)*
⟨*proof*⟩

lemma *100094 div 144 = (695::int)*
⟨*proof*⟩

lemma *100094 mod 144 = (14::int)*
⟨*proof*⟩

Powers

lemma *2 ^ 10 = (1024::int)*
⟨*proof*⟩

lemma *(- 3) ^ 7 = (-2187::int)*
⟨*proof*⟩

lemma *13 ^ 7 = (62748517::int)*
⟨*proof*⟩

lemma *3 ^ 15 = (14348907::int)*
⟨*proof*⟩

lemma *(- 5) ^ 11 = (-48828125::int)*
⟨*proof*⟩

16.4 The Natural Numbers

Successor

lemma *Suc 99999 = 100000*
⟨*proof*⟩

Addition

lemma *(13::nat) + 19 = 32*
⟨*proof*⟩

lemma $(1234::nat) + 5678 = 6912$
⟨*proof*⟩

lemma $(973646::nat) + 6475 = 980121$
⟨*proof*⟩

Subtraction

lemma $(32::nat) - 14 = 18$
⟨*proof*⟩

lemma $(14::nat) - 15 = 0$
⟨*proof*⟩

lemma $(14::nat) - 1576644 = 0$
⟨*proof*⟩

lemma $(48273776::nat) - 3873737 = 44400039$
⟨*proof*⟩

Multiplication

lemma $(12::nat) * 11 = 132$
⟨*proof*⟩

lemma $(647::nat) * 3643 = 2357021$
⟨*proof*⟩

Quotient and Remainder

lemma $(10::nat)\ div\ 3 = 3$
⟨*proof*⟩

lemma $(10::nat)\ mod\ 3 = 1$
⟨*proof*⟩

lemma $(10000::nat)\ div\ 9 = 1111$
⟨*proof*⟩

lemma $(10000::nat)\ mod\ 9 = 1$
⟨*proof*⟩

lemma $(10000::nat)\ div\ 16 = 625$
⟨*proof*⟩

lemma $(10000::nat)\ mod\ 16 = 0$
⟨*proof*⟩

Powers

lemma $2\ \hat{}\ 12 = (4096::nat)$

⟨*proof*⟩

lemma *3 ^ 10 = (59049::nat)*
⟨*proof*⟩

lemma *12 ^ 7 = (35831808::nat)*
⟨*proof*⟩

lemma *3 ^ 14 = (4782969::nat)*
⟨*proof*⟩

lemma *5 ^ 11 = (48828125::nat)*
⟨*proof*⟩

Testing the cancellation of complementary terms

lemma $y + (x + -x) = (0::int) + y$
⟨*proof*⟩

lemma $y + (-x + (- y + x)) = (0::int)$
⟨*proof*⟩

lemma $-x + (y + (- y + x)) = (0::int)$
⟨*proof*⟩

lemma $x + (x + (- x + (- x + (- y + - z)))) = (0::int) - y - z$
⟨*proof*⟩

lemma $x + x - x - x - y - z = (0::int) - y - z$
⟨*proof*⟩

lemma $x + y + z - (x + z) = y - (0::int)$
⟨*proof*⟩

lemma $x + (y + (y + (y + (-x + -x)))) = (0::int) + y - x + y + y$
⟨*proof*⟩

lemma $x + (y + (y + (y + (-y + -x)))) = y + (0::int) + y$
⟨*proof*⟩

lemma $x + y - x + z - x - y - z + x < (1::int)$
⟨*proof*⟩

16.5 Real Arithmetic

16.5.1 Addition

lemma $(1359::real) + -2468 = -1109$
⟨*proof*⟩

lemma $(93746::real) + -46375 = 47371$
⟨*proof*⟩

16.5.2 Negation

lemma $- (65745::real) = -65745$
⟨*proof*⟩

lemma $- (-54321::real) = 54321$
⟨*proof*⟩

16.5.3 Multiplication

lemma $(-84::real) * 51 = -4284$
⟨*proof*⟩

lemma $(255::real) * 255 = 65025$
⟨*proof*⟩

lemma $(1359::real) * -2468 = -3354012$
⟨*proof*⟩

16.5.4 Inequalities

lemma $(89::real) * 10 \neq 889$
⟨*proof*⟩

lemma $(13::real) < 18 - 4$
⟨*proof*⟩

lemma $(-345::real) < -242 + -100$
⟨*proof*⟩

lemma $(13557456::real) < 18678654$
⟨*proof*⟩

lemma $(999999::real) \leq (1000001 + 1) - 2$
⟨*proof*⟩

lemma $(1234567::real) \leq 1234567$
⟨*proof*⟩

16.5.5 Powers

lemma $2 \hat{} 15 = (32768::real)$
⟨*proof*⟩

lemma $(- 3) \hat{} 7 = (-2187::real)$
⟨*proof*⟩

lemma $13 \hat{} 7 = (62748517::real)$
⟨*proof*⟩

lemma $3 \hat{} 15 = (14348907::real)$
⟨*proof*⟩

lemma $(-5) \hat{} 11 = (-48828125::real)$
⟨*proof*⟩

16.5.6 Tests

lemma $(x + y = x) = (y = (0::real))$
⟨*proof*⟩

lemma $(x + y = y) = (x = (0::real))$
⟨*proof*⟩

lemma $(x + y = (0::real)) = (x = -y)$
⟨*proof*⟩

lemma $(x + y = (0::real)) = (y = -x)$
⟨*proof*⟩

lemma $((x + y) < (x + z)) = (y < (z::real))$
⟨*proof*⟩

lemma $((x + z) < (y + z)) = (x < (y::real))$
⟨*proof*⟩

lemma $(\neg\ x < y) = (y \leq (x::real))$
⟨*proof*⟩

lemma $\neg\ (x < y \wedge y < (x::real))$
⟨*proof*⟩

lemma $(x::real) < y ==> \neg\ y < x$
⟨*proof*⟩

lemma $((x::real) \neq y) = (x < y \vee y < x)$
⟨*proof*⟩

lemma $(\neg\ x \leq y) = (y < (x::real))$
⟨*proof*⟩

lemma $x \leq y \vee y \leq (x::real)$
⟨*proof*⟩

lemma $x \leq y \vee y < (x::real)$
⟨*proof*⟩

lemma $x < y \vee y \leq (x::real)$
⟨*proof*⟩

lemma $x \leq (x::real)$
⟨*proof*⟩

lemma $((x::real) \leq y) = (x < y \vee x = y)$
⟨*proof*⟩

lemma $((x::real) \leq y \wedge y \leq x) = (x = y)$
⟨*proof*⟩

lemma $\neg(x < y \wedge y \leq (x::real))$
⟨*proof*⟩

lemma $\neg(x \leq y \wedge y < (x::real))$
⟨*proof*⟩

lemma $(-x < (0::real)) = (0 < x)$
⟨*proof*⟩

lemma $((0::real) < -x) = (x < 0)$
⟨*proof*⟩

lemma $(-x \leq (0::real)) = (0 \leq x)$
⟨*proof*⟩

lemma $((0::real) \leq -x) = (x \leq 0)$
⟨*proof*⟩

lemma $(x::real) = y \vee x < y \vee y < x$
⟨*proof*⟩

lemma $(x::real) = 0 \vee 0 < x \vee 0 < -x$
⟨*proof*⟩

lemma $(0::real) \leq x \vee 0 \leq -x$
⟨*proof*⟩

lemma $((x::real) + y \leq x + z) = (y \leq z)$
⟨*proof*⟩

lemma $((x::real) + z \leq y + z) = (x \leq y)$
⟨*proof*⟩

lemma $(w::real) < x \wedge y < z \Longrightarrow w + y < x + z$
⟨*proof*⟩

lemma $(w::real) \leq x \wedge y \leq z \Longrightarrow w + y \leq x + z$
⟨*proof*⟩

lemma $(0::real) \leq x \wedge 0 \leq y \Longrightarrow 0 \leq x + y$
⟨*proof*⟩

lemma $(0::real) < x \wedge 0 < y \Longrightarrow 0 < x + y$
⟨*proof*⟩

lemma $(-x < y) = (0 < x + (y::real))$
⟨*proof*⟩

lemma $(x < -y) = (x + y < (0::real))$
⟨*proof*⟩

lemma $(y < x + -z) = (y + z < (x::real))$
⟨*proof*⟩

lemma $(x + -y < z) = (x < z + (y::real))$
⟨*proof*⟩

lemma $x \leq y \Longrightarrow x < y + (1::real)$
⟨*proof*⟩

lemma $(x - y) + y = (x::real)$
⟨*proof*⟩

lemma $y + (x - y) = (x::real)$
⟨*proof*⟩

lemma $x - x = (0::real)$
⟨*proof*⟩

lemma $(x - y = 0) = (x = (y::real))$
⟨*proof*⟩

lemma $((0::real) \leq x + x) = (0 \leq x)$
⟨*proof*⟩

lemma $(-x \leq x) = ((0::real) \leq x)$
⟨*proof*⟩

lemma $(x \leq -x) = (x \leq (0::real))$
⟨*proof*⟩

lemma $(-x = (0::real)) = (x = 0)$
⟨*proof*⟩

lemma $-(x - y) = y - (x::real)$

⟨*proof*⟩

lemma $((0::real) < x - y) = (y < x)$
⟨*proof*⟩

lemma $((0::real) \leq x - y) = (y \leq x)$
⟨*proof*⟩

lemma $(x + y) - x = (y::real)$
⟨*proof*⟩

lemma $(-x = y) = (x = (-y::real))$
⟨*proof*⟩

lemma $x < (y::real) ==> \neg(x = y)$
⟨*proof*⟩

lemma $(x \leq x + y) = ((0::real) \leq y)$
⟨*proof*⟩

lemma $(y \leq x + y) = ((0::real) \leq x)$
⟨*proof*⟩

lemma $(x < x + y) = ((0::real) < y)$
⟨*proof*⟩

lemma $(y < x + y) = ((0::real) < x)$
⟨*proof*⟩

lemma $(x - y) - x = (-y::real)$
⟨*proof*⟩

lemma $(x + y < z) = (x < z - (y::real))$
⟨*proof*⟩

lemma $(x - y < z) = (x < z + (y::real))$
⟨*proof*⟩

lemma $(x < y - z) = (x + z < (y::real))$
⟨*proof*⟩

lemma $(x \leq y - z) = (x + z \leq (y::real))$
⟨*proof*⟩

lemma $(x - y \leq z) = (x \leq z + (y::real))$
⟨*proof*⟩

lemma $(-x < -y) = (y < (x::real))$
⟨*proof*⟩

lemma $(-x \leq -y) = (y \leq (x::real))$
⟨*proof*⟩

lemma $(a + b) - (c + d) = (a - c) + (b - (d::real))$
⟨*proof*⟩

lemma $(0::real) - x = -x$
⟨*proof*⟩

lemma $x - (0::real) = x$
⟨*proof*⟩

lemma $w \leq x \wedge y < z \Longrightarrow w + y < x + (z::real)$
⟨*proof*⟩

lemma $w < x \wedge y \leq z \Longrightarrow w + y < x + (z::real)$
⟨*proof*⟩

lemma $(0::real) \leq x \wedge 0 < y \Longrightarrow 0 < x + (y::real)$
⟨*proof*⟩

lemma $(0::real) < x \wedge 0 \leq y \Longrightarrow 0 < x + y$
⟨*proof*⟩

lemma $-x - y = -(x + (y::real))$
⟨*proof*⟩

lemma $x - (-y) = x + (y::real)$
⟨*proof*⟩

lemma $-x - -y = y - (x::real)$
⟨*proof*⟩

lemma $(a - b) + (b - c) = a - (c::real)$
⟨*proof*⟩

lemma $(x = y - z) = (x + z = (y::real))$
⟨*proof*⟩

lemma $(x - y = z) = (x = z + (y::real))$
⟨*proof*⟩

lemma $x - (x - y) = (y::real)$
⟨*proof*⟩

lemma $x - (x + y) = -(y::real)$
⟨*proof*⟩

lemma $x = y \Longrightarrow x \leq (y::real)$
⟨*proof*⟩

lemma $(0::real) < x \Longrightarrow \neg(x = 0)$
⟨*proof*⟩

lemma $(x + y) * (x - y) = (x * x) - (y * y)$
⟨*proof*⟩

lemma $(-x = -y) = (x = (y::real))$
⟨*proof*⟩

lemma $(-x < -y) = (y < (x::real))$
⟨*proof*⟩

lemma $!!a::real.\ a \leq b \Longrightarrow c \leq d \Longrightarrow x + y < z \Longrightarrow a + c \leq b + d$
⟨*proof*⟩

lemma $!!a::real.\ a < b \Longrightarrow c < d \Longrightarrow a - d \leq b + (-c)$
⟨*proof*⟩

lemma $!!a::real.\ a \leq b \Longrightarrow b + b \leq c \Longrightarrow a + a \leq c$
⟨*proof*⟩

lemma $!!a::real.\ a + b \leq i + j \Longrightarrow a \leq b \Longrightarrow i \leq j \Longrightarrow a + a \leq j + j$
⟨*proof*⟩

lemma $!!a::real.\ a + b < i + j \Longrightarrow a < b \Longrightarrow i < j \Longrightarrow a + a < j + j$
⟨*proof*⟩

lemma $!!a::real.\ a + b + c \leq i + j + k \wedge a \leq b \wedge b \leq c \wedge i \leq j \wedge j \leq k \longrightarrow a + a + a \leq k + k + k$
⟨*proof*⟩

lemma $!!a::real.\ a + b + c + d \leq i + j + k + l \Longrightarrow a \leq b \Longrightarrow b \leq c \Longrightarrow c \leq d \Longrightarrow i \leq j \Longrightarrow j \leq k \Longrightarrow k \leq l \Longrightarrow a \leq l$
⟨*proof*⟩

lemma $!!a::real.\ a + b + c + d \leq i + j + k + l \Longrightarrow a \leq b \Longrightarrow b \leq c \Longrightarrow c \leq d \Longrightarrow i \leq j \Longrightarrow j \leq k \Longrightarrow k \leq l \Longrightarrow a + a + a + a \leq l + l + l + l$
⟨*proof*⟩

lemma $!!a::real.\ a + b + c + d \leq i + j + k + l \Longrightarrow a \leq b \Longrightarrow b \leq c \Longrightarrow c \leq d \Longrightarrow i \leq j \Longrightarrow j \leq k \Longrightarrow k \leq l \Longrightarrow a + a + a + a + a \leq l + l + l + l + i$
⟨*proof*⟩

lemma $!!a::real.\ a + b + c + d \leq i + j + k + l \Longrightarrow a \leq b \Longrightarrow b \leq c$

$==> c \leq d ==> i \leq j ==> j \leq k ==> k \leq l ==> a + a + a + a + a + a \leq l + l + l + l + i + l$
⟨*proof*⟩

16.6 Complex Arithmetic

lemma $(1359 + 93746*\mathrm{i}) - (2468 + 46375*\mathrm{i}) = -1109 + 47371*\mathrm{i}$
⟨*proof*⟩

lemma $- (65745 + -47371*\mathrm{i}) = -65745 + 47371*\mathrm{i}$
⟨*proof*⟩

Multiplication requires distributive laws. Perhaps versions instantiated to literal constants should be added to the simpset.

lemma $(1 + \mathrm{i}) * (1 - \mathrm{i}) = 2$
⟨*proof*⟩

lemma $(1 + 2*\mathrm{i}) * (1 + 3*\mathrm{i}) = -5 + 5*\mathrm{i}$
⟨*proof*⟩

lemma $(-84 + 255*\mathrm{i}) + (51 * 255*\mathrm{i}) = -84 + 13260 * \mathrm{i}$
⟨*proof*⟩

No inequalities or linear arithmetic: the complex numbers are unordered!

No powers (not supported yet)

end

17 Examples for hexadecimal and binary numerals

theory *Hex-Bin-Examples* **imports** *Main*
begin

Hex and bin numerals can be used like normal decimal numerals in input

lemma $0xFF = 255$ ⟨*proof*⟩
lemma $0xF = 0b1111$ ⟨*proof*⟩

Just like decimal numeral they are polymorphic, for arithmetic they need to be constrained

lemma $0x0A + 0x10 = (0x1A :: nat)$ ⟨*proof*⟩

The number of leading zeros is irrelevant

lemma $0b00010000 = 0x10$ ⟨*proof*⟩

Unary minus works as for decimal numerals

lemma $- 0x0A = - 10$ ⟨*proof*⟩

Hex and bin numerals are printed as decimal: $2::'a$

term *0b10*
term *0x0A*

The numerals 0 and 1 are syntactically different from the constants 0 and 1. For the usual numeric types, their values are the same, though.

lemma *0x01 = 1* ⟨*proof*⟩
lemma *0x00 = 0* ⟨*proof*⟩

lemma *0x01 = (1::nat)* ⟨*proof*⟩
lemma *0b0000 = (0::int)* ⟨*proof*⟩

end

18 Antiquotations

theory *Antiquote*
imports *Main*
begin

A simple example on quote / antiquote in higher-order abstract syntax.

definition *var* :: *'a* ⇒ (*'a* ⇒ *nat*) ⇒ *nat* (*VAR - [1000] 999*)
  where *var x env = env x*

definition *Expr* :: ((*'a* ⇒ *nat*) ⇒ *nat*) ⇒ (*'a* ⇒ *nat*) ⇒ *nat*
  where *Expr exp env = exp env*

syntax
  -Expr :: *'a* ⇒ *'a* (*EXPR - [1000] 999*)

⟨*ML*⟩

term *EXPR (a + b + c)*
term *EXPR (a + b + c + VAR x + VAR y + 1)*
term *EXPR (VAR (f w) + VAR x)*

term *Expr (λenv. env x)* — improper
term *Expr (λenv. f env)*
term *Expr (λenv. f env + env x)* — improper
term *Expr (λenv. f env y z)*
term *Expr (λenv. f env + g y env)*
term *Expr (λenv. f env + g env y + h a env z)*

end

19 Multiple nested quotations and anti-quotations

theory *Multiquote*

imports *Main*
begin

Multiple nested quotations and anti-quotations – basically a generalized version of de-Bruijn representation.

syntax
-quote :: $'b \Rightarrow ('a \Rightarrow 'b)$ $(\ll\text{-}\gg [0]\ 1000)$
-antiquote :: $('a \Rightarrow 'b) \Rightarrow 'b$ $(\acute{}\text{-}\ [1000]\ 1000)$

⟨*ML*⟩

basic examples

term $\ll a + b + c\gg$
term $\ll a + b + c + \acute{}x + \acute{}y + 1\gg$
term $\ll\acute{}(f\ w) + \acute{}x\gg$
term $\ll f\ \acute{}x\ \acute{}y\ z\gg$

advanced examples

term $\ll\ll\acute{}\acute{}x + \acute{}y\gg\gg$
term $\ll\ll\acute{}\acute{}x + \acute{}y\gg \circ \acute{}f\gg$
term $\ll\acute{}(f \circ \acute{}g)\gg$
term $\ll\ll\acute{}\acute{}(f \circ \acute{}g)\gg\gg$

end

20 Partial equivalence relations

theory *PER*
imports *Main*
begin

Higher-order quotients are defined over partial equivalence relations (PERs) instead of total ones. We provide axiomatic type classes $equiv < partial\text{-}equiv$ and a type constructor $'a\ quot$ with basic operations. This development is based on:

Oscar Slotosch: *Higher Order Quotients and their Implementation in Isabelle HOL.* Elsa L. Gunter and Amy Felty, editors, Theorem Proving in Higher Order Logics: TPHOLs '97, Springer LNCS 1275, 1997.

20.1 Partial equivalence

Type class *partial-equiv* models partial equivalence relations (PERs) using the polymorphic $\sim :: 'a \Rightarrow 'a \Rightarrow bool$ relation, which is required to be symmetric and transitive, but not necessarily reflexive.

class *partial-equiv* =
fixes *eqv* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\sim$ *50*)

assumes *partial-equiv-sym* [*elim?*]: $x \sim y \Longrightarrow y \sim x$
assumes *partial-equiv-trans* [*trans*]: $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$

The domain of a partial equivalence relation is the set of reflexive elements. Due to symmetry and transitivity this characterizes exactly those elements that are connected with *any* other one.

definition
$domain :: 'a::partial\text{-}equiv\ set$ **where**
$domain = \{x.\ x \sim x\}$

lemma *domainI* [*intro*]: $x \sim x \Longrightarrow x \in domain$
⟨*proof*⟩

lemma *domainD* [*dest*]: $x \in domain \Longrightarrow x \sim x$
⟨*proof*⟩

theorem *domainI'* [*elim?*]: $x \sim y \Longrightarrow x \in domain$
⟨*proof*⟩

20.2 Equivalence on function spaces

The $\sim$ relation is lifted to function spaces. It is important to note that this is *not* the direct product, but a structural one corresponding to the congruence property.

instantiation *fun* :: (*partial-equiv*, *partial-equiv*) *partial-equiv*
begin

definition $f \sim g \longleftrightarrow (\forall x \in domain.\ \forall y \in domain.\ x \sim y \longrightarrow f\ x \sim g\ y)$

lemma *partial-equiv-funI* [*intro?*]:
$(\bigwedge x\ y.\ x \in domain \Longrightarrow y \in domain \Longrightarrow x \sim y \Longrightarrow f\ x \sim g\ y) \Longrightarrow f \sim g$
⟨*proof*⟩

lemma *partial-equiv-funD* [*dest?*]:
$f \sim g \Longrightarrow x \in domain \Longrightarrow y \in domain \Longrightarrow x \sim y \Longrightarrow f\ x \sim g\ y$
⟨*proof*⟩

The class of partial equivalence relations is closed under function spaces (in *both* argument positions).

instance ⟨*proof*⟩

end

20.3 Total equivalence

The class of total equivalence relations on top of PERs. It coincides with the standard notion of equivalence, i.e. $\sim :: 'a \Rightarrow 'a \Rightarrow bool$ is required to

be reflexive, transitive and symmetric.

class *equiv* =
  assumes *eqv-refl* [*intro*]: $x \sim x$

On total equivalences all elements are reflexive, and congruence holds unconditionally.

theorem *equiv-domain* [*intro*]: $(x::'a::equiv) \in domain$
⟨*proof*⟩

theorem *equiv-cong* [*dest?*]: $f \sim g \Longrightarrow x \sim y \Longrightarrow f\ x \sim g\ (y::'a::equiv)$
⟨*proof*⟩

20.4 Quotient types

The quotient type $'a\ quot$ consists of all *equivalence classes* over elements of the base type $'a$.

definition $quot = \{\{x.\ a \sim x\} |\ a::'a::partial\text{-}equiv.\ True\}$

typedef (**overloaded**) $'a\ quot = quot :: 'a::partial\text{-}equiv\ set\ set$
  ⟨*proof*⟩

lemma *quotI* [*intro*]: $\{x.\ a \sim x\} \in quot$
  ⟨*proof*⟩

lemma *quotE* [*elim*]: $R \in quot \Longrightarrow (\bigwedge a.\ R = \{x.\ a \sim x\} \Longrightarrow C) \Longrightarrow C$
  ⟨*proof*⟩

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

definition *eqv-class* :: $('a::partial\text{-}equiv) \Rightarrow 'a\ quot$ $(\lfloor\text{-}\rfloor)$
  where $\lfloor a \rfloor = Abs\text{-}quot\ \{x.\ a \sim x\}$

theorem *quot-rep*: $\exists\, a.\ A = \lfloor a \rfloor$
⟨*proof*⟩

lemma *quot-cases* [*cases type*: *quot*]:
  obtains (*rep*) a **where** $A = \lfloor a \rfloor$
  ⟨*proof*⟩

20.5 Equality on quotients

Equality of canonical quotient elements corresponds to the original relation as follows.

theorem *eqv-class-eqI* [*intro*]: $a \sim b \Longrightarrow \lfloor a \rfloor = \lfloor b \rfloor$
⟨*proof*⟩

theorem *eqv-class-eqD'* [*dest?*]: $\lfloor a \rfloor = \lfloor b \rfloor \Longrightarrow a \in domain \Longrightarrow a \sim b$
⟨*proof*⟩

theorem *eqv-class-eqD* [*dest?*]: $\lfloor a \rfloor = \lfloor b \rfloor \Longrightarrow a \sim (b::'a::equiv)$
⟨*proof*⟩

lemma *eqv-class-eq'* [*simp*]: $a \in domain \Longrightarrow \lfloor a \rfloor = \lfloor b \rfloor \longleftrightarrow a \sim b$
⟨*proof*⟩

lemma *eqv-class-eq* [*simp*]: $\lfloor a \rfloor = \lfloor b \rfloor \longleftrightarrow a \sim (b::'a::equiv)$
⟨*proof*⟩

20.6 Picking representing elements

definition *pick* :: $'a::partial\text{-}equiv\ quot \Rightarrow 'a$
where $pick\ A = (SOME\ a.\ A = \lfloor a \rfloor)$

theorem *pick-eqv'* [*intro?*, *simp*]: $a \in domain \Longrightarrow pick\ \lfloor a \rfloor \sim a$
⟨*proof*⟩

theorem *pick-eqv* [*intro*, *simp*]: $pick\ \lfloor a \rfloor \sim (a::'a::equiv)$
⟨*proof*⟩

theorem *pick-inverse*: $\lfloor pick\ A \rfloor = (A::'a::equiv\ quot)$
⟨*proof*⟩

end

21 Summing natural numbers

theory *NatSum* **imports** *Main* **begin**

Summing natural numbers, squares, cubes, etc.

Thanks to Sloane's On-Line Encyclopedia of Integer Sequences, http://www.research.att.com/~njas/sequences.

lemmas [*simp*] =
ring-distribs
diff-mult-distrib *diff-mult-distrib2* — for type nat

The sum of the first n odd numbers equals n squared.

lemma *sum-of-odds*: $(\sum i{=}0..{<}n.\ Suc\ (i + i)) = n * n$
⟨*proof*⟩

The sum of the first n odd squares.

lemma *sum-of-odd-squares*:
$3 * (\sum i{=}0..{<}n.\ Suc(2{*}i) * Suc(2{*}i)) = n * (4 * n * n - 1)$
⟨*proof*⟩

The sum of the first n odd cubes

lemma *sum-of-odd-cubes*:
$(\sum i{=}0..{<}n.\ Suc\ (2{*}i) * Suc\ (2{*}i) * Suc\ (2{*}i)) =$
$n * n * (2 * n * n - 1)$
⟨*proof*⟩

The sum of the first n positive integers equals $n\ (n + 1)\ /\ 2$.

lemma *sum-of-naturals*:
$2 * (\sum i{=}0..n.\ i) = n * Suc\ n$
⟨*proof*⟩

lemma *sum-of-squares*:
$6 * (\sum i{=}0..n.\ i * i) = n * Suc\ n * Suc\ (2 * n)$
⟨*proof*⟩

lemma *sum-of-cubes*:
$4 * (\sum i{=}0..n.\ i * i * i) = n * n * Suc\ n * Suc\ n$
⟨*proof*⟩

A cute identity:

lemma *sum-squared*: $(\sum i{=}0..n.\ i)\hat{}2 = (\sum i{=}0..n{::}nat.\ i\hat{}3)$
⟨*proof*⟩

Sum of fourth powers: three versions.

lemma *sum-of-fourth-powers*:
$30 * (\sum i{=}0..n.\ i * i * i * i) =$
$n * Suc\ n * Suc\ (2 * n) * (3 * n * n + 3 * n - 1)$
⟨*proof*⟩

Two alternative proofs, with a change of variables and much more subtraction, performed using the integers.

lemma *int-sum-of-fourth-powers*:
$30 * int\ (\sum i{=}0..{<}m.\ i * i * i * i) =$
$int\ m * (int\ m - 1) * (int(2 * m) - 1) *$
$(int(3 * m * m) - int(3 * m) - 1)$
⟨*proof*⟩

lemma *of-nat-sum-of-fourth-powers*:
$30 * of\text{-}nat\ (\sum i{=}0..{<}m.\ i * i * i * i) =$
$of\text{-}nat\ m * (of\text{-}nat\ m - 1) * (of\text{-}nat\ (2 * m) - 1) *$
$(of\text{-}nat\ (3 * m * m) - of\text{-}nat\ (3 * m) - (1{::}int))$
⟨*proof*⟩

Sums of geometric series: *2*, *3* and the general case.

lemma *sum-of-2-powers*: $(\sum i{=}0..{<}n.\ 2\hat{}i) = 2\hat{}n - (1{::}nat)$
⟨*proof*⟩

lemma *sum-of-3-powers*: $2 * (\sum i{=}0..{<}n.\ 3\hat{}i) = 3\hat{}n - (1{::}nat)$
⟨*proof*⟩

lemma *sum-of-powers*: $0 < k ==> (k - 1) * (\sum i{=}0..{<}n.\ k\hat{}i) = k\hat{}n - (1{::}nat)$
⟨*proof*⟩

end

⟨*ML*⟩

22 Three Divides Theorem

theory *ThreeDivides*
imports *Main* ~~/*src*/*HOL*/*Library*/*LaTeXsugar*
begin

22.1 Abstract

The following document presents a proof of the Three Divides N theorem formalised in the Isabelle/Isar theorem proving system.

Theorem: 3 divides n if and only if 3 divides the sum of all digits in n.

Informal Proof: Take $n = \sum n_j * 10^j$ where n_j is the j'th least significant digit of the decimal denotation of the number n and the sum ranges over all digits. Then

$$(n - \sum n_j) = \sum n_j * (10^j - 1)$$

We know $\forall j\ 3|(10^j - 1)$ and hence $3|LHS$, therefore

$$\forall n\ 3|n \Longleftrightarrow 3|\sum n_j$$

□

22.2 Formal proof

22.2.1 Miscellaneous summation lemmas

If a divides $A\ x$ for all x then a divides any sum over terms of the form $(A\ x)*(P\ x)$ for arbitrary P.

lemma *div-sum*:
fixes *a*::*nat* **and** *n*::*nat*
shows $\forall x.\ a\ dvd\ A\ x \Longrightarrow a\ dvd\ (\sum x{<}n.\ A\ x * D\ x)$
⟨*proof*⟩

22.2.2 Generalised Three Divides

This section solves a generalised form of the three divides problem. Here we show that for any sequence of numbers the theorem holds. In the next section we specialise this theorem to apply directly to the decimal expansion of the natural numbers.

Here we show that the first statement in the informal proof is true for all natural numbers. Note we are using $D\ i$ to denote the i'th element in a sequence of numbers.

lemma *digit-diff-split*:
fixes *n*::*nat* **and** *nd*::*nat* **and** *x*::*nat*
shows $n = (\sum x{\in}\{..{<}nd\}.\ (D\ x){*}((10{::}nat)\hat{}x)) \Longrightarrow$
$(n - (\sum x{<}nd.\ (D\ x))) = (\sum x{<}nd.\ (D\ x){*}(10\hat{}x - 1))$
⟨*proof*⟩

Now we prove that 3 always divides numbers of the form $10^x - 1$.

lemma *three-divs-0*:
shows (*3*::*nat*) *dvd* (*10*^*x* − *1*)
⟨*proof*⟩

Expanding on the previous lemma and lemma *div-sum*.

lemma *three-divs-1*:
fixes *D* :: *nat* ⇒ *nat*
shows *3 dvd* $(\sum x{<}nd.\ D\ x * (10\hat{}x - 1))$
⟨*proof*⟩

Using lemmas *digit-diff-split* and *three-divs-1* we now prove the following lemma.

lemma *three-divs-2*:
fixes *nd*::*nat* **and** *D*::*nat*⇒*nat*
shows *3 dvd* $((\sum x{<}nd.\ (D\ x){*}(10\hat{}x)) - (\sum x{<}nd.\ (D\ x)))$
⟨*proof*⟩

We now present the final theorem of this section. For any sequence of numbers (defined by a function D), we show that 3 divides the expansive sum $\sum (D\ x) * 10^x$ over x if and only if 3 divides the sum of the individual numbers $\sum D\ x$.

lemma *three-div-general*:
fixes *D* :: *nat* ⇒ *nat*
shows (*3 dvd* $(\sum x{<}nd.\ D\ x * 10\hat{}x)) = (3\ dvd\ (\sum x{<}nd.\ D\ x))$
⟨*proof*⟩

22.2.3 Three Divides Natural

This section shows that for all natural numbers we can generate a sequence of digits less than ten that represent the decimal expansion of the number. We then use the lemma *three-div-general* to prove our final theorem.

Definitions of length and digit sum.

This section introduces some functions to calculate the required properties of natural numbers. We then proceed to prove some properties of these functions.

The function *nlen* returns the number of digits in a natural number n.

fun *nlen* :: *nat* ⇒ *nat*
where
  nlen 0 = 0
| *nlen x = 1 + nlen (x div 10)*

The function *sumdig* returns the sum of all digits in some number n.

definition
  sumdig :: *nat* ⇒ *nat* **where**
  sumdig n = (∑ *x* < *nlen n*. *n div 10^x mod 10*)

Some properties of these functions follow.

lemma *nlen-zero*:
  0 = nlen x ⟹ *x = 0*
  ⟨*proof*⟩

lemma *nlen-suc*:
  Suc m = nlen n ⟹ *m = nlen* (*n div 10*)
  ⟨*proof*⟩

The following lemma is the principle lemma required to prove our theorem. It states that an expansion of some natural number n into a sequence of its individual digits is always possible.

lemma *exp-exists*:
  m = (∑ *x*<*nlen m*. (*m div* (*10*::*nat*) ^*x mod 10*) ∗ *10^x*)
⟨*proof*⟩

Final theorem.

We now combine the general theorem *three-div-general* and existence result of *exp-exists* to prove our final theorem.

theorem *three-divides-nat*:
  shows (*3 dvd n*) = (*3 dvd sumdig n*)
⟨*proof*⟩

end

23 The Cubic and Quartic Root Formulas

theory *Cubic-Quartic*
imports *Complex-Main*
begin

24 The Cubic Formula

definition *ccbrt z* = (*SOME* (*w*::*complex*). *w* ^ *3* = *z*)

lemma *ccbrt*: (*ccbrt z*) ^ *3* = *z*
⟨*proof*⟩

The reduction to a simpler form:

lemma *cubic-reduction*:
fixes *a* :: *complex*
assumes
a ≠ *0* ∧ *x* = *y* − *b* / (*3* ∗ *a*) ∧ *p* = (*3*∗ *a* ∗ *c* − *b* ^ *2*) / (*9* ∗ *a* ^ *2*) ∧
q = (*9* ∗ *a* ∗ *b* ∗ *c* − *2* ∗ *b* ^ *3* − *27* ∗ *a* ^ *2* ∗ *d*) / (*54* ∗ *a* ^ *3*)
shows *a* ∗ *x* ^ *3* + *b* ∗ *x* ^ *2* + *c* ∗ *x* + *d* = *0* ⟷ *y* ^ *3* + *3* ∗ *p* ∗ *y* − *2* ∗ *q* = *0*
⟨*proof*⟩

The solutions of the special form:

lemma *cubic-basic*:
fixes *s* :: *complex*
assumes
s ^ *2* = *q* ^ *2* + *p* ^ *3* ∧
s1 ^ *3* = (*if p* = *0 then 2* ∗ *q else q* + *s*) ∧
s2 = −*s1* ∗ (*1* + *i* ∗ *t*) / *2* ∧
s3 = −*s1* ∗ (*1* − *i* ∗ *t*) / *2* ∧
i ^ *2* + *1* = *0* ∧
t ^ *2* = *3*
shows
if p = *0*
then y ^ *3* + *3* ∗ *p* ∗ *y* − *2* ∗ *q* = *0* ⟷ *y* = *s1* ∨ *y* = *s2* ∨ *y* = *s3*
else s1 ≠ *0* ∧
(*y* ^ *3* + *3* ∗ *p* ∗ *y* − *2* ∗ *q* = *0* ⟷ (*y* = *s1* − *p* / *s1* ∨ *y* = *s2* − *p* / *s2* ∨
y = *s3* − *p* / *s3*))
⟨*proof*⟩

Explicit formula for the roots:

lemma *cubic*:
assumes *a0*: *a* ≠ *0*
shows
let
p = (*3* ∗ *a* ∗ *c* − *b* ^ *2*) / (*9* ∗ *a* ^ *2*);
q = (*9* ∗ *a* ∗ *b* ∗ *c* − *2* ∗ *b* ^ *3* − *27* ∗ *a* ^ *2* ∗ *d*) / (*54* ∗ *a* ^ *3*);
s = *csqrt*(*q* ^ *2* + *p* ^ *3*);
s1 = (*if p* = *0 then ccbrt*(*2* ∗ *q*) *else ccbrt*(*q* + *s*));
s2 = −*s1* ∗ (*1* + i ∗ *csqrt 3*) / *2*;
s3 = −*s1* ∗ (*1* − i ∗ *csqrt 3*) / *2*
in
if p = *0 then*
a ∗ *x* ^ *3* + *b* ∗ *x* ^ *2* + *c* ∗ *x* + *d* = *0* ⟷
x = *s1* − *b* / (*3* ∗ *a*) ∨

$x = s2 - b / (3 * a) \vee$
$x = s3 - b / (3 * a)$
else
$s1 \neq 0 \wedge$
$(a * x\hat{}3 + b * x\hat{}2 + c * x + d = 0 \longleftrightarrow$
$x = s1 - p / s1 - b / (3 * a) \vee$
$x = s2 - p / s2 - b / (3 * a) \vee$
$x = s3 - p / s3 - b / (3 * a))$
⟨*proof*⟩

25 The Quartic Formula

lemma *quartic*:
$(y::real)\hat{}3 - b * y\hat{}2 + (a * c - 4 * d) * y - a\hat{}2 * d + 4 * b * d - c\hat{}2 = 0$
$\wedge$
$R\hat{}2 = a\hat{}2 / 4 - b + y \wedge$
$s\hat{}2 = y\hat{}2 - 4 * d \wedge$
$(D\hat{}2 = (\textit{if } R = 0 \textit{ then } 3 * a\hat{}2 / 4 - 2 * b + 2 * s$
$\textit{else } 3 * a\hat{}2 / 4 - R\hat{}2 - 2 * b + (4 * a * b - 8 * c - a\hat{}3) /$
$(4 * R))) \wedge$
$(E\hat{}2 = (\textit{if } R = 0 \textit{ then } 3 * a\hat{}2 / 4 - 2 * b - 2 * s$
$\textit{else } 3 * a\hat{}2 / 4 - R\hat{}2 - 2 * b - (4 * a * b - 8 * c - a\hat{}3) /$
$(4 * R)))$
$\Longrightarrow x\hat{}4 + a * x\hat{}3 + b * x\hat{}2 + c * x + d = 0 \longleftrightarrow$
$x = -a / 4 + R / 2 + D / 2 \vee$
$x = -a / 4 + R / 2 - D / 2 \vee$
$x = -a / 4 - R / 2 + E / 2 \vee$
$x = -a / 4 - R / 2 - E / 2$
⟨*proof*⟩

end

26 The Pythagorean Theorem

theory *Pythagoras*
imports *Complex-Main*
begin

Expressed in real numbers:

lemma *pythagoras-verbose*:
$((A1::real) - B1) * (C1 - B1) + (A2 - B2) * (C2 - B2) = 0 \Longrightarrow$
$(C1 - A1) * (C1 - A1) + (C2 - A2) * (C2 - A2) =$
$((B1 - A1) * (B1 - A1) + (B2 - A2) * (B2 - A2)) + (C1 - B1) * (C1 -$
$B1) + (C2 - B2) * (C2 - B2)$
⟨*proof*⟩

Expressed in vectors:

type-synonym *point* = *real* × *real*

lemma *pythagoras*:
defines *ort*:*orthogonal* ≡ (λ(*A*::*point*) *B*. *fst A* ∗ *fst B* + *snd A* ∗ *snd B* = *0*)
and *vc*:*vector* ≡ (λ(*A*::*point*) *B*. (*fst A* − *fst B*, *snd A* − *snd B*))
and *vcn*:*vecsqnorm* ≡ (λ*A*::*point*. *fst A* ^ *2* + *snd A* ^*2*)
assumes *o*: *orthogonal* (*vector A B*) (*vector C B*)
shows *vecsqnorm*(*vector C A*) = *vecsqnorm*(*vector B A*) + *vecsqnorm*(*vector C B*)
⟨*proof*⟩

end

27 Higher-Order Logic: Intuitionistic predicate calculus problems

theory *Intuitionistic* **imports** *Main* **begin**

lemma (~~(*P*&*Q*)) = ((~~*P*) & (~~*Q*))
⟨*proof*⟩

lemma ~~ ((~*P* −−> *Q*) −−> (~*P* −−> ~*Q*) −−> *P*)
⟨*proof*⟩

lemma (~~(*P*−−>*Q*)) = (~~*P* −−> ~~*Q*)
⟨*proof*⟩

lemma (~~~*P*) = (~*P*)
⟨*proof*⟩

lemma ~~((*P* −−> *Q* | *R*) −−> (*P*−−>*Q*) | (*P*−−>*R*))
⟨*proof*⟩

lemma (*P*=*Q*) = (*Q*=*P*)
⟨*proof*⟩

lemma ((*P* −−> (*Q* | (*Q*−−>*R*))) −−> *R*) −−> *R*
⟨*proof*⟩

lemma (((*G*−−>*A*) −−> *J*) −−> *D* −−> *E*) −−> (((*H*−−>*B*)−−>*I*)−−>*C*−−>*J*)
−−> (*A*−−>*H*) −−> *F* −−> *G* −−> (((*C*−−>*B*)−−>*I*)−−>*D*)−−>(*A*−−>*C*)
−−> (((*F*−−>*A*)−−>*B*) −−> *I*) −−> *E*
⟨*proof*⟩

lemma $P \dashrightarrow \sim\sim P$
⟨*proof*⟩

lemma $\sim\sim(\sim\sim P \dashrightarrow P)$
⟨*proof*⟩

lemma $\sim\sim P \;\&\; \sim\sim(P \dashrightarrow Q) \dashrightarrow \sim\sim Q$
⟨*proof*⟩

lemma $((P{=}Q) \dashrightarrow P\&Q\&R) \;\&$
$((Q{=}R) \dashrightarrow P\&Q\&R) \;\&$
$((R{=}P) \dashrightarrow P\&Q\&R) \dashrightarrow P\&Q\&R$
⟨*proof*⟩

lemma $((P{=}Q) \dashrightarrow P\&Q\&R\&S\&T) \;\&$
$((Q{=}R) \dashrightarrow P\&Q\&R\&S\&T) \;\&$
$((R{=}S) \dashrightarrow P\&Q\&R\&S\&T) \;\&$
$((S{=}T) \dashrightarrow P\&Q\&R\&S\&T) \;\&$
$((T{=}P) \dashrightarrow P\&Q\&R\&S\&T) \dashrightarrow P\&Q\&R\&S\&T$
⟨*proof*⟩

lemma $(ALL\ x.\ EX\ y.\ ALL\ z.\ p(x) \;\&\; q(y) \;\&\; r(z)) =$
$(ALL\ z.\ EX\ y.\ ALL\ x.\ p(x) \;\&\; q(y) \;\&\; r(z))$
⟨*proof*⟩

lemma $\sim\ (EX\ x.\ ALL\ y.\ p\ y\ x = (\sim\ p\ x\ x))$
⟨*proof*⟩

lemma $\sim\sim((P\dashrightarrow Q) \;=\; (\sim Q \dashrightarrow \sim P))$
⟨*proof*⟩

lemma $\sim\sim(\sim\sim P \;=\; P)$
⟨*proof*⟩

lemma $\sim(P{-}{-}{>}Q) \;{-}{-}{>}\; (Q{-}{-}{>}P)$
⟨*proof*⟩

lemma $\sim\sim((\sim P{-}{-}{>}Q) \;=\; (\sim Q \;{-}{-}{>}\; P))$
⟨*proof*⟩

lemma $\sim\sim((P|Q{-}{-}{>}P|R) \;{-}{-}{>}\; P|(Q{-}{-}{>}R))$
⟨*proof*⟩

lemma $\sim\sim(P \mid \sim P)$
⟨*proof*⟩

lemma $\sim\sim(P \mid \sim\sim\sim P)$
⟨*proof*⟩

lemma $\sim\sim(((P{-}{-}{>}Q) \;{-}{-}{>}\; P) \;\; {-}{-}{>}\;\; P)$
⟨*proof*⟩

lemma $((P|Q) \;\&\; (\sim P|Q) \;\&\; (P|\sim Q)) \;{-}{-}{>}\; \sim\; (\sim P \mid \sim Q)$
⟨*proof*⟩

lemma $(Q{-}{-}{>}R) \;{-}{-}{>}\; (R{-}{-}{>}P\&Q) \;{-}{-}{>}\; (P{-}{-}{>}(Q|R)) \;{-}{-}{>}\; (P{=}Q)$
⟨*proof*⟩

lemma $P{=}P$
⟨*proof*⟩

lemma $\sim\sim(((P \;=\; Q) = R) \;=\; (P = (Q = R)))$
⟨*proof*⟩

lemma $((P \;=\; Q) = R) \;\; {-}{-}{>}\;\; \sim\sim(P = (Q = R))$
⟨*proof*⟩

lemma $(P \mid (Q \;\&\; R)) \;= ((P \mid Q) \;\&\; (P \mid R))$
⟨*proof*⟩

lemma $\sim\sim((P = Q) = ((Q \mid \sim P) \;\&\; (\sim Q|P)))$
⟨*proof*⟩

lemma $\sim\sim((P \dashrightarrow Q) = (\sim P \mid Q))$
⟨*proof*⟩

lemma $\sim\sim((P \dashrightarrow Q) \mid (Q \dashrightarrow P))$
⟨*proof*⟩

lemma $\sim\sim(((P \;\&\; (Q \dashrightarrow R)) \dashrightarrow S) = ((\sim P \mid Q \mid S) \;\&\; (\sim P \mid \sim R \mid S)))$
⟨*proof*⟩

lemma $(P\&Q) = (P = (Q = (P|Q)))$
⟨*proof*⟩

lemma $(EX\ x.\ P(x) \dashrightarrow Q) \dashrightarrow (ALL\ x.\ P(x)) \dashrightarrow Q$
⟨*proof*⟩

lemma $((ALL\ x.\ P(x)) \dashrightarrow Q) \dashrightarrow \sim (ALL\ x.\ P(x) \;\&\; \sim Q)$
⟨*proof*⟩

lemma $((ALL\ x.\ \sim P(x)) \dashrightarrow Q) \dashrightarrow \sim (ALL\ x.\ \sim (P(x)|Q))$
⟨*proof*⟩

lemma $(ALL\ x.\ P(x)) \mid Q \dashrightarrow (ALL\ x.\ P(x) \mid Q)$
⟨*proof*⟩

lemma $(EX\ x.\ P \dashrightarrow Q(x)) \dashrightarrow (P \dashrightarrow (EX\ x.\ Q(x)))$
⟨*proof*⟩

lemma $\sim\sim(EX\ x.\ ALL\ y\ z.\ (P(y) \dashrightarrow Q(z)) \dashrightarrow (P(x) \dashrightarrow Q(x)))$
⟨*proof*⟩

lemma $(ALL\ x\ y.\ EX\ z.\ ALL\ w.\ (P(x)\&Q(y)\text{--}{>}R(z)\&S(w)))$
$\text{--}{>}\ (EX\ x\ y.\ P(x)\ \&\ Q(y))\ \text{--}{>}\ (EX\ z.\ R(z))$
⟨*proof*⟩

lemma $(EX\ x.\ P\text{--}{>}Q(x))\ \&\ (EX\ x.\ Q(x)\text{--}{>}P)\ \text{--}{>}\ {\sim}{\sim}(EX\ x.\ P{=}Q(x))$
⟨*proof*⟩

lemma $(ALL\ x.\ P = Q(x))\ \text{--}{>}\ (P = (ALL\ x.\ Q(x)))$
⟨*proof*⟩

lemma ${\sim}{\sim}\ ((ALL\ x.\ P\ |\ Q(x))\ =\ (P\ |\ (ALL\ x.\ Q(x))))$
⟨*proof*⟩

lemma $(EX\ x.\ P(x))\ \&$
$(ALL\ x.\ L(x)\ \text{--}{>}\ {\sim}\ (M(x)\ \&\ R(x)))\ \&$
$(ALL\ x.\ P(x)\ \text{--}{>}\ (M(x)\ \&\ L(x)))\ \&$
$((ALL\ x.\ P(x)\text{--}{>}Q(x))\ |\ (EX\ x.\ P(x)\&R(x)))$
$\text{--}{>}\ (EX\ x.\ Q(x)\&P(x))$
⟨*proof*⟩

lemma $(EX\ x.\ P(x)\ \&\ {\sim}Q(x))\ \&$
$(ALL\ x.\ P(x)\ \text{--}{>}\ R(x))\ \&$
$(ALL\ x.\ M(x)\ \&\ L(x)\ \text{--}{>}\ P(x))\ \&$
$((EX\ x.\ R(x)\ \&\ {\sim}\ Q(x))\ \text{--}{>}\ (ALL\ x.\ L(x)\ \text{--}{>}\ {\sim}\ R(x)))$
$\text{--}{>}\ (ALL\ x.\ M(x)\ \text{--}{>}\ {\sim}L(x))$
⟨*proof*⟩

lemma $(ALL\ x.\ P(x)\ \text{--}{>}\ (ALL\ x.\ Q(x)))\ \&$
$({\sim}{\sim}(ALL\ x.\ Q(x)|R(x))\ \text{--}{>}\ (EX\ x.\ Q(x)\&S(x)))\ \&$
$({\sim}{\sim}(EX\ x.\ S(x))\ \text{--}{>}\ (ALL\ x.\ L(x)\ \text{--}{>}\ M(x)))$
$\text{--}{>}\ (ALL\ x.\ P(x)\ \&\ L(x)\ \text{--}{>}\ M(x))$
⟨*proof*⟩

lemma $(((EX\ x.\ P(x))\ \&\ (EX\ y.\ Q(y)))\ \text{--}{>}$
$(((ALL\ x.\ (P(x)\ \text{--}{>}\ R(x)))\ \&\ (ALL\ y.\ (Q(y)\ \text{--}{>}\ S(y))))\ =$
$(ALL\ x\ y.\ ((P(x)\ \&\ Q(y))\ \text{--}{>}\ (R(x)\ \&\ S(y))))))$
⟨*proof*⟩

lemma $(ALL\ x.\ (P(x)\ |\ Q(x))\ \text{--}{>}\ {\sim}\ R(x))\ \&$

$(ALL\ x.\ (Q(x)\ \text{--}\!\!>\ \sim\ S(x))\ \text{--}\!\!>\ P(x)\ \&\ R(x))$
$\text{--}\!\!>\ (ALL\ x.\ \sim\sim S(x))$
⟨*proof*⟩

lemma $\sim(EX\ x.\ P(x)\ \&\ (Q(x)\ |\ R(x)))\ \&$
$(EX\ x.\ L(x)\ \&\ P(x))\ \&$
$(ALL\ x.\ \sim\ R(x)\ \text{--}\!\!>\ M(x))$
$\text{--}\!\!>\ (EX\ x.\ L(x)\ \&\ M(x))$
⟨*proof*⟩

lemma $(ALL\ x.\ P(x)\ \&\ (Q(x)|R(x))\text{--}\!\!>S(x))\ \&$
$(ALL\ x.\ S(x)\ \&\ R(x)\ \text{--}\!\!>\ L(x))\ \&$
$(ALL\ x.\ M(x)\ \text{--}\!\!>\ R(x))$
$\text{--}\!\!>\ (ALL\ x.\ P(x)\ \&\ M(x)\ \text{--}\!\!>\ L(x))$
⟨*proof*⟩

lemma $(ALL\ x.\ \sim\sim(P(a)\ \&\ (P(x)\text{--}\!\!>P(b))\text{--}\!\!>P(c)))\ =$
$(ALL\ x.\ \sim\sim((\sim P(a)\ |\ P(x)\ |\ P(c))\ \&\ (\sim P(a)\ |\ \sim P(b)\ |\ P(c))))$
⟨*proof*⟩

lemma
$(ALL\ x.\ EX\ y.\ J\ x\ y)\ \&$
$(ALL\ x.\ EX\ y.\ G\ x\ y)\ \&$
$(ALL\ x\ y.\ J\ x\ y\ |\ G\ x\ y\ \text{--}\!\!>\ (ALL\ z.\ J\ y\ z\ |\ G\ y\ z\ \text{--}\!\!>\ H\ x\ z))$
$\text{--}\!\!>\ (ALL\ x.\ EX\ y.\ H\ x\ y)$
⟨*proof*⟩

lemma $\sim\ (EX\ x.\ ALL\ y.\ F\ y\ x = (\sim F\ y\ y))$
⟨*proof*⟩

lemma $(EX\ y.\ ALL\ x.\ F\ x\ y = F\ x\ x)\ \text{--}\!\!>$
$\sim(ALL\ x.\ EX\ y.\ ALL\ z.\ F\ z\ y = (\sim\ F\ z\ x))$
⟨*proof*⟩

lemma $(ALL\ x.\ f(x)\ \text{--}\!\!>$
$(EX\ y.\ g(y)\ \&\ h\ x\ y\ \&\ (EX\ y.\ g(y)\ \&\ \sim\ h\ x\ y)))\ \&$
$(EX\ x.\ j(x)\ \&\ (ALL\ y.\ g(y)\ \text{--}\!\!>\ h\ x\ y))$
$\text{--}\!\!>\ (EX\ x.\ j(x)\ \&\ \sim f(x))$
⟨*proof*⟩

lemma $(a{=}b\ |\ c{=}d)\ \&\ (a{=}c\ |\ b{=}d)\ \text{--}\!\!>\ a{=}d\ |\ b{=}c$

⟨*proof*⟩

lemma ((*EX z w*. (*ALL x y*. (*P x y* = ((*x* = *z*) & (*y* = *w*))))) −−>
(*EX z*. (*ALL x*. (*EX w*. ((*ALL y*. (*P x y* = (*y* = *w*))) = (*x* = *z*))))))
⟨*proof*⟩

lemma ((*EX z w*. (*ALL x y*. (*P x y* = ((*x* = *z*) & (*y* = *w*))))) −−>
(*EX w*. (*ALL y*. (*EX z*. ((*ALL x*. (*P x y* = (*x* = *z*))) = (*y* = *w*))))))
⟨*proof*⟩

lemma (*ALL x*. (*EX y*. *P*(*y*) & *x*=*f*(*y*)) −−> *P*(*x*)) = (*ALL x*. *P*(*x*) −−>
P(*f*(*x*)))
⟨*proof*⟩

lemma *P* (*f a b*) (*f b c*) & *P* (*f b c*) (*f a c*) &
(*ALL x y z*. *P x y* & *P y z* −−> *P x z*) −−> *P* (*f a b*) (*f a c*)
⟨*proof*⟩

lemma *ALL x*. *P x* (*f x*) = (*EX y*. (*ALL z*. *P z y* −−> *P z* (*f x*)) & *P x y*)
⟨*proof*⟩

end

28 CTL formulae

theory *CTL*
imports *Main*
begin

We formalize basic concepts of Computational Tree Logic (CTL) [2, 1] within the simply-typed set theory of HOL.

By using the common technique of "shallow embedding", a CTL formula is identified with the corresponding set of states where it holds. Consequently, CTL operations such as negation, conjunction, disjunction simply become complement, intersection, union of sets. We only require a separate operation for implication, as point-wise inclusion is usually not encountered in plain set-theory.

lemmas [*intro*!] = *Int-greatest Un-upper2 Un-upper1 Int-lower1 Int-lower2*

type-synonym *'a ctl* = *'a set*

definition *imp* :: *'a ctl* ⇒ *'a ctl* ⇒ *'a ctl* (**infixr** → *75*)

where $p \to q = - p \cup q$

lemma [*intro*!]: $p \cap p \to q \subseteq q$ ⟨*proof*⟩
lemma [*intro*!]: $p \subseteq (q \to p)$ ⟨*proof*⟩

The CTL path operators are more interesting; they are based on an arbitrary, but fixed model $\mathcal{M}$, which is simply a transition relation over states $'a$.

axiomatization $\mathcal{M} :: ('a \times 'a)\ set$

The operators **EX**, **EF**, **EG** are taken as primitives, while **AX**, **AF**, **AG** are defined as derived ones. The formula **EX** p holds in a state s, iff there is a successor state s' (with respect to the model $\mathcal{M}$), such that p holds in s'. The formula **EF** p holds in a state s, iff there is a path in $\mathcal{M}$, starting from s, such that there exists a state s' on the path, such that p holds in s'. The formula **EG** p holds in a state s, iff there is a path, starting from s, such that for all states s' on the path, p holds in s'. It is easy to see that **EF** p and **EG** p may be expressed using least and greatest fixed points [2].

definition *EX* (**EX** - [*80*] *90*)
where [*simp*]: **EX** $p = \{s.\ \exists s'.\ (s,\ s') \in \mathcal{M} \wedge s' \in p\}$

definition *EF* (**EF** - [*80*] *90*)
where [*simp*]: **EF** p = *lfp* $(\lambda s.\ p \cup$ **EX** $s)$

definition *EG* (**EG** - [*80*] *90*)
where [*simp*]: **EG** p = *gfp* $(\lambda s.\ p \cap$ **EX** $s)$

AX, **AF** and **AG** are now defined dually in terms of **EX**, **EF** and **EG**.

definition *AX* (**AX** - [*80*] *90*)
where [*simp*]: **AX** $p = -$ **EX** $-\ p$
definition *AF* (**AF** - [*80*] *90*)
where [*simp*]: **AF** $p = -$ **EG** $-\ p$
definition *AG* (**AG** - [*80*] *90*)
where [*simp*]: **AG** $p = -$ **EF** $-\ p$

28.1 Basic fixed point properties

First of all, we use the de-Morgan property of fixed points.

lemma *lfp-gfp*: *lfp* $f = -$ *gfp* $(\lambda s::'a\ set.\ -\ (f\ (-\ s)))$
⟨*proof*⟩

lemma *lfp-gfp'*: $-$ *lfp* f = *gfp* $(\lambda s::'a\ set.\ -\ (f\ (-\ s)))$
⟨*proof*⟩

lemma *gfp-lfp'*: $-$ *gfp* f = *lfp* $(\lambda s::'a\ set.\ -\ (f\ (-\ s)))$
⟨*proof*⟩

In order to give dual fixed point representations of **AF** p and **AG** p:

lemma *AF-lfp*: **AF** p = *lfp* ($\lambda s.\ p \cup$ **AX** s)
⟨*proof*⟩

lemma *AG-gfp*: **AG** p = *gfp* ($\lambda s.\ p \cap$ **AX** s)
⟨*proof*⟩

lemma *EF-fp*: **EF** $p = p \cup$ **EX EF** p
⟨*proof*⟩

lemma *AF-fp*: **AF** $p = p \cup$ **AX AF** p
⟨*proof*⟩

lemma *EG-fp*: **EG** $p = p \cap$ **EX EG** p
⟨*proof*⟩

From the greatest fixed point definition of **AG** p, we derive as a consequence of the Knaster-Tarski theorem on the one hand that **AG** p is a fixed point of the monotonic function $\lambda s.\ p \cap$ **AX** s.

lemma *AG-fp*: **AG** $p = p \cap$ **AX AG** p
⟨*proof*⟩

This fact may be split up into two inequalities (merely using transitivity of $\subseteq$, which is an instance of the overloaded $\leq$ in Isabelle/HOL).

lemma *AG-fp-1*: **AG** $p \subseteq p$
⟨*proof*⟩

lemma *AG-fp-2*: **AG** $p \subseteq$ **AX AG** p
⟨*proof*⟩

On the other hand, we have from the Knaster-Tarski fixed point theorem that any other post-fixed point of $\lambda s.\ p \cap$ **AX** s is smaller than **AG** p. A post-fixed point is a set of states q such that $q \subseteq p \cap$ **AX** q. This leads to the following co-induction principle for **AG** p.

lemma *AG-I*: $q \subseteq p \cap$ **AX** $q \Longrightarrow q \subseteq$ **AG** p
⟨*proof*⟩

28.2 The tree induction principle

With the most basic facts available, we are now able to establish a few more interesting results, leading to the *tree induction* principle for **AG** (see below). We will use some elementary monotonicity and distributivity rules.

lemma *AX-int*: **AX** $(p \cap q) =$ **AX** $p \cap$ **AX** q ⟨*proof*⟩
lemma *AX-mono*: $p \subseteq q \Longrightarrow$ **AX** $p \subseteq$ **AX** q ⟨*proof*⟩
lemma *AG-mono*: $p \subseteq q \Longrightarrow$ **AG** $p \subseteq$ **AG** q
⟨*proof*⟩

The formula **AG** p implies **AX** p (we use substitution of $\subseteq$ with monotonicity).

lemma *AG-AX*: **AG** $p \subseteq$ **AX** p
⟨*proof*⟩

Furthermore we show idempotency of the **AG** operator. The proof is a good example of how accumulated facts may get used to feed a single rule step.

lemma *AG-AG*: **AG AG** $p =$ **AG** p
⟨*proof*⟩

We now give an alternative characterization of the **AG** operator, which describes the **AG** operator in an "operational" way by tree induction: In a state holds **AG** p iff in that state holds p, and in all reachable states s follows from the fact that p holds in s, that p also holds in all successor states of s. We use the co-induction principle *AG-I* to establish this in a purely algebraic manner.

theorem *AG-induct*: $p \cap$ **AG** $(p \to$ **AX** $p) =$ **AG** p
⟨*proof*⟩

28.3 An application of tree induction

Further interesting properties of CTL expressions may be demonstrated with the help of tree induction; here we show that **AX** and **AG** commute.

theorem *AG-AX-commute*: **AG AX** $p =$ **AX AG** p
⟨*proof*⟩

end

29 Arithmetic

theory *Arith-Examples*
imports *Main*
begin

The *arith* method is used frequently throughout the Isabelle distribution. This file merely contains some additional tests and special corner cases. Some rather technical remarks:

`Lin_Arith.simple_tac` is a very basic version of the tactic. It performs no meta-to-object-logic conversion, and only some splitting of operators. `Lin_Arith.tac` performs meta-to-object-logic conversion, full splitting of operators, and NNF normalization of the goal. The *arith* method combines them both, and tries other methods (e.g. *presburger*) as well. This is the one that you should use in your proofs!

An *arith*-based simproc is available as well (see `Lin_Arith.simproc`), which—for performance reasons—however does even less splitting than `Lin_Arith.simple_tac` at the moment (namely inequalities only). (On the other hand, it does take apart conjunctions, which `Lin_Arith.simple_tac` currently does not do.)

29.1 Splitting of Operators: *max*, *min*, *abs*, *op −*, *nat*, *op mod*, *op div*

lemma $(i::nat) <= max\ i\ j$
⟨*proof*⟩

lemma $(i::int) <= max\ i\ j$
⟨*proof*⟩

lemma $min\ i\ j <= (i::nat)$
⟨*proof*⟩

lemma $min\ i\ j <= (i::int)$
⟨*proof*⟩

lemma $min\ (i::nat)\ j <= max\ i\ j$
⟨*proof*⟩

lemma $min\ (i::int)\ j <= max\ i\ j$
⟨*proof*⟩

lemma $min\ (i::nat)\ j + max\ i\ j = i + j$
⟨*proof*⟩

lemma $min\ (i::int)\ j + max\ i\ j = i + j$
⟨*proof*⟩

lemma $(i::nat) < j ==> min\ i\ j < max\ i\ j$
⟨*proof*⟩

lemma $(i::int) < j ==> min\ i\ j < max\ i\ j$
⟨*proof*⟩

lemma $(0::int) <= |i|$
⟨*proof*⟩

lemma $(i::int) <= |i|$
⟨*proof*⟩

lemma $||i::int|| = |i|$
⟨*proof*⟩

Also testing subgoals with bound variables.

lemma $!!x.\ (x::nat) <= y ==> x - y = 0$
⟨*proof*⟩

lemma $!!x.\ (x::nat) - y = 0 ==> x <= y$
⟨*proof*⟩

lemma $!!x.\ ((x::nat) <= y) = (x - y = 0)$

⟨*proof*⟩

lemma [| (*x*::*nat*) < *y*; *d* < *1* |] ==> *x* − *y* = *d*
⟨*proof*⟩

lemma [| (*x*::*nat*) < *y*; *d* < *1* |] ==> *x* − *y* − *x* = *d* − *x*
⟨*proof*⟩

lemma (*x*::*int*) < *y* ==> *x* − *y* < *0*
⟨*proof*⟩

lemma *nat* (*i* + *j*) <= *nat i* + *nat j*
⟨*proof*⟩

lemma *i* < *j* ==> *nat* (*i* − *j*) = *0*
⟨*proof*⟩

lemma (*i*::*nat*) *mod 0* = *i*

⟨*proof*⟩

lemma (*i*::*nat*) *mod 1* = *0*

⟨*proof*⟩

lemma (*i*::*nat*) *mod 42* <= *41*
⟨*proof*⟩

lemma (*i*::*int*) *mod 0* = *i*

⟨*proof*⟩

lemma (*i*::*int*) *mod 1* = *0*

⟨*proof*⟩

lemma (*i*::*int*) *mod 42* <= *41*
⟨*proof*⟩

lemma −(*i*::*int*) ∗ *1* = *0* ==> *i* = *0*
⟨*proof*⟩

lemma [| (*0*::*int*) < |*i*|; |*i*| ∗ *1* < |*i*| ∗ *j* |] ==> *1* < |*i*| ∗ *j*
⟨*proof*⟩

29.2 Meta-Logic

lemma *x* < *Suc y* == *x* <= *y*
⟨*proof*⟩

lemma $((x::nat) == z ==> x \sim= y) ==> x \sim= y \mid z \sim= y$
⟨*proof*⟩

29.3 Various Other Examples

lemma $(x < Suc\ y) = (x <= y)$
⟨*proof*⟩

lemma $[|\ (x::nat) < y;\ y < z\ |] ==> x < z$
⟨*proof*⟩

lemma $(x::nat) < y\ \&\ y < z ==> x < z$
⟨*proof*⟩

This example involves no arithmetic at all, but is solved by preprocessing (i.e. NNF normalization) alone.

lemma $(P::bool) = Q ==> Q = P$
⟨*proof*⟩

lemma $[|\ P = (x = 0);\ (\sim P) = (y = 0)\ |] ==> min\ (x::nat)\ y = 0$
⟨*proof*⟩

lemma $[|\ P = (x = 0);\ (\sim P) = (y = 0)\ |] ==> max\ (x::nat)\ y = x + y$
⟨*proof*⟩

lemma $[|\ (x::nat) \sim= y;\ a + 2 = b;\ a < y;\ y < b;\ a < x;\ x < b\ |] ==> False$
⟨*proof*⟩

lemma $[|\ (x::nat) > y;\ y > z;\ z > x\ |] ==> False$
⟨*proof*⟩

lemma $(x::nat) - 5 > y ==> y < x$
⟨*proof*⟩

lemma $(x::nat) \sim= 0 ==> 0 < x$
⟨*proof*⟩

lemma $[|\ (x::nat) \sim= y;\ x <= y\ |] ==> x < y$
⟨*proof*⟩

lemma $[|\ (x::nat) < y;\ P\ (x - y)\ |] ==> P\ 0$
⟨*proof*⟩

lemma $(x - y) - (x::nat) = (x - x) - y$
⟨*proof*⟩

lemma $[|\ (a::nat) < b;\ c < d\ |] ==> (a - b) = (c - d)$
⟨*proof*⟩

lemma $((a::nat) - (b - (c - (d - e)))) = (a - (b - (c - (d - e))))$
⟨*proof*⟩

lemma $(n < m \;\&\; m < n') \mid (n < m \;\&\; m = n') \mid (n < n' \;\&\; n' < m) \mid$
$(n = n' \;\&\; n' < m) \mid (n = m \;\&\; m < n') \mid$
$(n' < m \;\&\; m < n) \mid (n' < m \;\&\; m = n) \mid$
$(n' < n \;\&\; n < m) \mid (n' = n \;\&\; n < m) \mid (n' = m \;\&\; m < n) \mid$
$(m < n \;\&\; n < n') \mid (m < n \;\&\; n' = n) \mid (m < n' \;\&\; n' < n) \mid$
$(m = n \;\&\; n < n') \mid (m = n' \;\&\; n' < n) \mid$
$(n' = m \;\&\; m = (n::nat))$

⟨*proof*⟩

lemma $2 * (x::nat) \sim= 1$

⟨*proof*⟩

Constants.

lemma $(0::nat) < 1$
⟨*proof*⟩

lemma $(0::int) < 1$
⟨*proof*⟩

lemma $(47::nat) + 11 < 8 * 15$
⟨*proof*⟩

lemma $(47::int) + 11 < 8 * 15$
⟨*proof*⟩

Splitting of inequalities of different type.

lemma $[\!|\ (a::nat) \sim= b;\ (i::int) \sim= j;\ a < 2;\ b < 2\ |\!] ==>$
$a + b <= nat\ (max\ |i|\ |j|)$
⟨*proof*⟩

Again, but different order.

lemma $[\!|\ (i::int) \sim= j;\ (a::nat) \sim= b;\ a < 2;\ b < 2\ |\!] ==>$
$a + b <= nat\ (max\ |i|\ |j|)$
⟨*proof*⟩

end

30 2-3 Trees

theory *Tree23*
imports *Main*
begin

This is a very direct translation of some of the functions in table.ML in the Isabelle source code. That source is due to Makarius Wenzel and Stefan Berghofer.

So far this file contains only data types and functions, but no proofs. Feel free to have a go at the latter!

Note that because of complicated patterns and mutual recursion, these function definitions take a few minutes and can also be seen as stress tests for the function definition facility.

type-synonym *key = int* — for simplicity, should be a type class

datatype *ord = LESS | EQUAL | GREATER*

definition *ord i j = (if i<j then LESS else if i=j then EQUAL else GREATER)*

```
datatype 'a tree23 =
  Empty |
  Branch2 'a tree23 key * 'a 'a tree23 |
  Branch3 'a tree23 key * 'a 'a tree23 key * 'a 'a tree23

datatype 'a growth =
  Stay 'a tree23 |
  Sprout 'a tree23 key * 'a 'a tree23

fun add :: key ⇒ 'a ⇒ 'a tree23 ⇒ 'a growth where
add key y Empty = Sprout Empty (key,y) Empty |
add key y (Branch2 left (k,x) right) =
   (case ord key k of
      LESS =>
        (case add key y left of
           Stay left' => Stay (Branch2 left' (k,x) right)
         | Sprout left1 q left2
           => Stay (Branch3 left1 q left2 (k,x) right))
    | EQUAL => Stay (Branch2 left (k,y) right)
    | GREATER =>
        (case add key y right of
           Stay right' => Stay (Branch2 left (k,x) right')
         | Sprout right1 q right2
           => Stay (Branch3 left (k,x) right1 q right2))) |
add key y (Branch3 left (k1,x1) mid (k2,x2) right) =
   (case ord key k1 of
      LESS =>
        (case add key y left of
```

```
      Stay left' => Stay (Branch3 left' (k1,x1) mid (k2,x2) right)
    | Sprout left1 q left2
      => Sprout (Branch2 left1 q left2) (k1,x1) (Branch2 mid (k2,x2) right))
  | EQUAL => Stay (Branch3 left (k1,y) mid (k2,x2) right)
  | GREATER =>
    (case ord key k2 of
      LESS =>
       (case add key y mid of
          Stay mid' => Stay (Branch3 left (k1,x1) mid' (k2,x2) right)
        | Sprout mid1 q mid2
          => Sprout (Branch2 left (k1,x1) mid1) q (Branch2 mid2 (k2,x2)
right))
    | EQUAL => Stay (Branch3 left (k1,x1) mid (k2,y) right)
    | GREATER =>
       (case add key y right of
          Stay right' => Stay (Branch3 left (k1,x1) mid (k2,x2) right')
        | Sprout right1 q right2
          => Sprout (Branch2 left (k1,x1) mid) (k2,x2) (Branch2 right1 q
right2))))

definition add0 :: key ⇒ 'a ⇒ 'a tree23 ⇒ 'a tree23 where
add0 k y t =
  (case add k y t of Stay t' => t' | Sprout l p r => Branch2 l p r)

value add0 5 e (add0 4 d (add0 3 c (add0 2 b (add0 1 a Empty))))

fun compare where
compare None (k2, -) = LESS |
compare (Some k1) (k2, -) = ord k1 k2

fun if-eq where
if-eq EQUAL x y = x |
if-eq - x y = y

fun del :: key option ⇒ 'a tree23 ⇒ ((key * 'a) * bool * 'a tree23)option
where
del (Some k) Empty = None |
del None (Branch2 Empty p Empty) = Some(p, (True, Empty)) |
del None (Branch3 Empty p Empty q Empty) = Some(p, (False, Branch2 Empty
q Empty)) |
del k (Branch2 Empty p Empty) = (case compare k p of
    EQUAL => Some(p, (True, Empty)) | - => None) |
del k (Branch3 Empty p Empty q Empty) = (case compare k p of
    EQUAL => Some(p, (False, Branch2 Empty q Empty))
  | - => (case compare k q of
      EQUAL => Some(q, (False, Branch2 Empty p Empty))
    | - => None)) |
del k (Branch2 l p r) = (case compare k p of
    LESS => (case del k l of None ⇒ None |
```

```
      Some(p', (False, l')) => Some(p', (False, Branch2 l' p r))
    | Some(p', (True, l')) => Some(p', case r of
        Branch2 rl rp rr => (True, Branch3 l' p rl rp rr)
      | Branch3 rl rp rm rq rr => (False, Branch2
          (Branch2 l' p rl) rp (Branch2 rm rq rr))))
  | or => (case del (if-eq or None k) r of None ⇒ None |
      Some(p', (False, r')) => Some(p', (False, Branch2 l (if-eq or p' p) r'))
    | Some(p', (True, r')) => Some(p', case l of
        Branch2 ll lp lr => (True, Branch3 ll lp lr (if-eq or p' p) r')
      | Branch3 ll lp lm lq lr => (False, Branch2
          (Branch2 ll lp lm) lq (Branch2 lr (if-eq or p' p) r'))))) |
del k (Branch3 l p m q r) = (case compare k q of
    LESS => (case compare k p of
      LESS => (case del k l of None ⇒ None |
        Some(p', (False, l')) => Some(p', (False, Branch3 l' p m q r))
      | Some(p', (True, l')) => Some(p', (False, case (m, r) of
          (Branch2 ml mp mr, Branch2 - - -) => Branch2 (Branch3 l' p ml mp
mr) q r
        | (Branch3 ml mp mm mq mr, -) => Branch3 (Branch2 l' p ml) mp
(Branch2 mm mq mr) q r
        | (Branch2 ml mp mr, Branch3 rl rp rm rq rr) =>
            Branch3 (Branch2 l' p ml) mp (Branch2 mr q rl) rp (Branch2 rm rq
rr))))
    | or => (case del (if-eq or None k) m of None ⇒ None |
        Some(p', (False, m')) => Some(p', (False, Branch3 l (if-eq or p' p) m' q
r))
      | Some(p', (True, m')) => Some(p', (False, case (l, r) of
          (Branch2 ll lp lr, Branch2 - - -) => Branch2 (Branch3 ll lp lr (if-eq or
p' p) m') q r
        | (Branch3 ll lp lm lq lr, -) => Branch3 (Branch2 ll lp lm) lq (Branch2 lr
(if-eq or p' p) m') q r
        | (-, Branch3 rl rp rm rq rr) => Branch3 l (if-eq or p' p) (Branch2 m' q
rl) rp (Branch2 rm rq rr)))))
  | or => (case del (if-eq or None k) r of None ⇒ None |
      Some(q', (False, r')) => Some(q', (False, Branch3 l p m (if-eq or q' q) r'))
    | Some(q', (True, r')) => Some(q', (False, case (l, m) of
        (Branch2 - - -, Branch2 ml mp mr) => Branch2 l p (Branch3 ml mp mr
(if-eq or q' q) r')
      | (-, Branch3 ml mp mm mq mr) => Branch3 l p (Branch2 ml mp mm) mq
(Branch2 mr (if-eq or q' q) r')
      | (Branch3 ll lp lm lq lr, Branch2 ml mp mr) =>
          Branch3 (Branch2 ll lp lm) lq (Branch2 lr p ml) mp (Branch2 mr (if-eq
or q' q) r')))))
```

definition *del0* :: *key* ⇒ *'a tree23* ⇒ *'a tree23* **where**
del0 k t = (*case del* (*Some k*) *t of None* ⇒ *t* | *Some*(-,(-,*t'*)) ⇒ *t'*)

Ordered trees

definition *opt-less* :: *key option* ⇒ *key option* ⇒ *bool* **where**

opt-less x y = (*case x of None* ⇒ *True* | *Some a* ⇒ (*case y of None* ⇒ *True* | *Some b* ⇒ *a* < *b*))

lemma *opt-less-simps* [*simp*]:
opt-less None y = *True*
opt-less x None = *True*
opt-less (*Some a*) (*Some b*) = (*a* < *b*)
⟨*proof*⟩

primrec *ord'* :: *key option* ⇒ *'a tree23* ⇒ *key option* ⇒ *bool* **where**
ord' x Empty y = *opt-less x y* |
ord' x (*Branch2 l p r*) *y* = (*ord' x l* (*Some* (*fst p*)) & *ord'* (*Some* (*fst p*)) *r y*) |
ord' x (*Branch3 l p m q r*) *y* = (*ord' x l* (*Some* (*fst p*)) & *ord'* (*Some* (*fst p*)) *m* (*Some* (*fst q*)) & *ord'* (*Some* (*fst q*)) *r y*)

definition *ord0* :: *'a tree23* ⇒ *bool* **where**
ord0 t = *ord' None t None*

Balanced trees

inductive *full* :: *nat* ⇒ *'a tree23* ⇒ *bool* **where**
full 0 Empty |
⟦*full n l*; *full n r*⟧ ⟹ *full* (*Suc n*) (*Branch2 l p r*) |
⟦*full n l*; *full n m*; *full n r*⟧ ⟹ *full* (*Suc n*) (*Branch3 l p m q r*)

inductive-cases *full-elims*:
full n Empty
full n (*Branch2 l p r*)
full n (*Branch3 l p m q r*)

inductive-cases *full-0-elim*: *full 0 t*
inductive-cases *full-Suc-elim*: *full* (*Suc n*) *t*

lemma *full-0-iff* [*simp*]: *full 0 t* ⟷ *t* = *Empty*
⟨*proof*⟩

lemma *full-Empty-iff* [*simp*]: *full n Empty* ⟷ *n* = *0*
⟨*proof*⟩

lemma *full-Suc-Branch2-iff* [*simp*]:
full (*Suc n*) (*Branch2 l p r*) ⟷ *full n l* ∧ *full n r*
⟨*proof*⟩

lemma *full-Suc-Branch3-iff* [*simp*]:
full (*Suc n*) (*Branch3 l p m q r*) ⟷ *full n l* ∧ *full n m* ∧ *full n r*
⟨*proof*⟩

fun *height* :: *'a tree23* ⇒ *nat* **where**
height Empty = *0* |
height (*Branch2 l - r*) = *Suc*(*max* (*height l*) (*height r*)) |

height (*Branch3 l - m - r*) = *Suc*(*max* (*height l*) (*max* (*height m*) (*height r*)))

Is a tree balanced?

fun *bal* :: *'a tree23* ⇒ *bool* **where**
bal Empty = *True* |
bal (*Branch2 l - r*) = (*bal l* & *bal r* & *height l* = *height r*) |
bal (*Branch3 l - m - r*) = (*bal l* & *bal m* & *bal r* & *height l* = *height m* & *height m* = *height r*)

lemma *full-imp-height*: *full n t* ⟹ *height t* = *n*
⟨*proof*⟩

lemma *full-imp-bal*: *full n t* ⟹ *bal t*
⟨*proof*⟩

lemma *bal-imp-full*: *bal t* ⟹ *full* (*height t*) *t*
⟨*proof*⟩

lemma *bal-iff-full*: *bal t* ⟷ (∃ *n*. *full n t*)
⟨*proof*⟩

The *add0* function either preserves the height of the tree, or increases it by one. The constructor returned by the *add* function determines which: A return value of the form *Stay t* indicates that the height will be the same. A value of the form *Sprout l p r* indicates an increase in height.

primrec *gfull* :: *nat* ⇒ *'a growth* ⇒ *bool* **where**
gfull n (*Stay t*) ⟷ *full n t* |
gfull n (*Sprout l p r*) ⟷ *full n l* ∧ *full n r*

lemma *gfull-add*: *full n t* ⟹ *gfull n* (*add k y t*)
⟨*proof*⟩

The *add0* operation preserves balance.

lemma *bal-add0*: *bal t* ⟹ *bal* (*add0 k y t*)
⟨*proof*⟩

The *add0* operation preserves order.

lemma *ord-cases*:
fixes *a b* :: *int* **obtains**
ord a b = *LESS* **and** *a* < *b* |
ord a b = *EQUAL* **and** *a* = *b* |
ord a b = *GREATER* **and** *a* > *b*
⟨*proof*⟩

definition *gtree* :: *'a growth* ⇒ *'a tree23* **where**
gtree g = (*case g of Stay t* ⇒ *t* | *Sprout l p r* ⇒ *Branch2 l p r*)

lemma *gtree-simps* [*simp*]:

gtree (Stay t) = t
gtree (Sprout l p r) = Branch2 l p r
⟨*proof*⟩

lemma *add0*: *add0 k y t = gtree (add k y t)*
⟨*proof*⟩

lemma *ord'-add0*:
⟦*ord' k1 t k2*; *opt-less k1 (Some k)*; *opt-less (Some k) k2*⟧ ⟹ *ord' k1 (add0 k y t) k2*
⟨*proof*⟩

lemma *ord0-add0*: *ord0 t* ⟹ *ord0 (add0 k y t)*
⟨*proof*⟩

The *del* function preserves balance.

lemma *del-extra-simps*:
l ≠ Empty ∨ r ≠ Empty ⟹
del k (Branch2 l p r) = (case compare k p of
LESS => (case del k l of None ⇒ None |
Some(p', (False, l')) => Some(p', (False, Branch2 l' p r))
| Some(p', (True, l')) => Some(p', case r of
Branch2 rl rp rr => (True, Branch3 l' p rl rp rr)
| Branch3 rl rp rm rq rr => (False, Branch2
(Branch2 l' p rl) rp (Branch2 rm rq rr))))
| or => (case del (if-eq or None k) r of None ⇒ None |
Some(p', (False, r')) => Some(p', (False, Branch2 l (if-eq or p' p) r'))
| Some(p', (True, r')) => Some(p', case l of
Branch2 ll lp lr => (True, Branch3 ll lp lr (if-eq or p' p) r')
| Branch3 ll lp lm lq lr => (False, Branch2
(Branch2 ll lp lm) lq (Branch2 lr (if-eq or p' p) r')))))
l ≠ Empty ∨ m ≠ Empty ∨ r ≠ Empty ⟹
del k (Branch3 l p m q r) = (case compare k q of
LESS => (case compare k p of
LESS => (case del k l of None ⇒ None |
Some(p', (False, l')) => Some(p', (False, Branch3 l' p m q r))
| Some(p', (True, l')) => Some(p', (False, case (m, r) of
(Branch2 ml mp mr, Branch2 - - -) => Branch2 (Branch3 l' p ml mp mr) q r
| (Branch3 ml mp mm mq mr, -) => Branch3 (Branch2 l' p ml) mp (Branch2 mm mq mr) q r
| (Branch2 ml mp mr, Branch3 rl rp rm rq rr) =>
Branch3 (Branch2 l' p ml) mp (Branch2 mr q rl) rp (Branch2 rm rq rr))))
| or => (case del (if-eq or None k) m of None ⇒ None |
Some(p', (False, m')) => Some(p', (False, Branch3 l (if-eq or p' p) m' q r))
| Some(p', (True, m')) => Some(p', (False, case (l, r) of
(Branch2 ll lp lr, Branch2 - - -) => Branch2 (Branch3 ll lp lr (if-eq or

```
p' p) m') q r
        | (Branch3 ll lp lm lq lr, -) => Branch3 (Branch2 ll lp lm) lq (Branch2 lr
(if-eq or p' p) m') q r
        | (-, Branch3 rl rp rm rq rr) => Branch3 l (if-eq or p' p) (Branch2 m' q
rl) rp (Branch2 rm rq rr)))))
    | or => (case del (if-eq or None k) r of None ⇒ None |
       Some(q', (False, r')) => Some(q', (False, Branch3 l p m (if-eq or q' q) r'))
     | Some(q', (True, r')) => Some(q', (False, case (l, m) of
        (Branch2 - - -, Branch2 ml mp mr) => Branch2 l p (Branch3 ml mp mr
(if-eq or q' q) r')
      | (-, Branch3 ml mp mm mq mr) => Branch3 l p (Branch2 ml mp mm) mq
(Branch2 mr (if-eq or q' q) r')
      | (Branch3 ll lp lm lq lr, Branch2 ml mp mr) =>
          Branch3 (Branch2 ll lp lm) lq (Branch2 lr p ml) mp (Branch2 mr (if-eq
or q' q) r')))))
⟨proof⟩

fun dfull where
dfull n None ⟷ True |
dfull n (Some (p, (True, t'))) ⟷ full n t' |
dfull n (Some (p, (False, t'))) ⟷ full (Suc n) t'

lemmas dfull-case-intros =
  ord.exhaust [of y dfull a (case-ord b c d y)]
  option.exhaust [of y dfull a (case-option b c y)]
  prod.exhaust [of y dfull a (case-prod b y)]
  bool.exhaust [of y dfull a (case-bool b c y)]
  tree23.exhaust [of y dfull a (Some (b, case-tree23 c d e y))]
  tree23.exhaust [of y full a (case-tree23 b c d y)]
  for a b c d e y

lemma dfull-del: full (Suc n) t ⟹ dfull n (del k t)
⟨proof⟩

lemma bal-del0: bal t ⟹ bal (del0 k t)
⟨proof⟩
```

This is a little test harness and should be commented out once the above functions have been proved correct.

```
datatype 'a act = Add int 'a | Del int

fun exec where
exec [] t = t |
exec (Add k x # as) t = exec as (add0 k x t) |
exec (Del k # as) t = exec as (del0 k t)
```

Some quick checks:

```
lemma bal-exec: bal t ⟹ bal (exec as t)
  ⟨proof⟩
```

lemma *bal*(*exec as Empty*)
⟨*proof*⟩

lemma *ord0*(*exec as Empty*)
quickcheck
⟨*proof*⟩

end

31 (Finite) multisets

theory *Multiset*
imports *Main*
begin

31.1 The type of multisets

definition *multiset* = {*f* :: *'a* ⇒ *nat*. *finite* {*x*. *f x* > *0*}}

typedef *'a multiset* = *multiset* :: (*'a* ⇒ *nat*) *set*
morphisms *count Abs-multiset*
⟨*proof*⟩

setup-lifting *type-definition-multiset*

lemma *multiset-eq-iff*: *M* = *N* ⟷ (∀ *a*. *count M a* = *count N a*)
⟨*proof*⟩

lemma *multiset-eqI*: (⋀*x*. *count A x* = *count B x*) ⟹ *A* = *B*
⟨*proof*⟩

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*: (λ*a*. *0*) ∈ *multiset*
⟨*proof*⟩

lemma *only1-in-multiset*: (λ*b*. *if b* = *a then n else 0*) ∈ *multiset*
⟨*proof*⟩

lemma *union-preserves-multiset*: *M* ∈ *multiset* ⟹ *N* ∈ *multiset* ⟹ (λ*a*. *M a* + *N a*) ∈ *multiset*
⟨*proof*⟩

lemma *diff-preserves-multiset*:
assumes *M* ∈ *multiset*
shows (λ*a*. *M a* − *N a*) ∈ *multiset*
⟨*proof*⟩

lemma *filter-preserves-multiset*:

assumes $M \in multiset$
shows $(\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
⟨*proof*⟩

lemmas *in-multiset* = *const0-in-multiset only1-in-multiset*
union-preserves-multiset diff-preserves-multiset filter-preserves-multiset

31.2 Representing multisets

Multiset enumeration

instantiation *multiset* :: (*type*) *cancel-comm-monoid-add*
begin

lift-definition *zero-multiset* :: $'a\ multiset$ **is** $\lambda a.\ 0$
⟨*proof*⟩

abbreviation *Mempty* :: $'a\ multiset$ ($\{\#\}$) **where**
Mempty ≡ *0*

lift-definition *plus-multiset* :: $'a\ multiset \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ **is** $\lambda M\ N.$ $(\lambda a.\ M\ a + N\ a)$
⟨*proof*⟩

lift-definition *minus-multiset* :: $'a\ multiset \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ **is** $\lambda\ M\ N.\ \lambda a.\ M\ a - N\ a$
⟨*proof*⟩

instance
⟨*proof*⟩

end

context
begin

qualified definition *is-empty* :: $'a\ multiset \Rightarrow bool$ **where**
[*code-abbrev*]: *is-empty A* ⟷ *A* = $\{\#\}$

end

lemma *add-mset-in-multiset*:
assumes *M*: ‹*M* ∈ *multiset*›
shows ‹$(\lambda b.\ if\ b = a\ then\ Suc\ (M\ b)\ else\ M\ b) \in multiset$›
⟨*proof*⟩

lift-definition *add-mset* :: $'a \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ **is**
$\lambda a\ M\ b.\ if\ b = a\ then\ Suc\ (M\ b)\ else\ M\ b$
⟨*proof*⟩

syntax
-multiset :: *args* ⇒ *'a multiset* ({#(-)#})
translations
{#*x*, *xs*#} == *CONST add-mset x* {#*xs*#}
{#*x*#} == *CONST add-mset x* {#}

lemma *count-empty* [*simp*]: *count* {#} *a* = *0*
⟨*proof*⟩

lemma *count-add-mset* [*simp*]:
count (*add-mset b A*) *a* = (*if b* = *a then Suc* (*count A a*) *else count A a*)
⟨*proof*⟩

lemma *count-single*: *count* {#*b*#} *a* = (*if b* = *a then 1 else 0*)
⟨*proof*⟩

lemma
add-mset-not-empty [*simp*]: ‹*add-mset a A* ≠ {#}› **and**
empty-not-add-mset [*simp*]: {#} ≠ *add-mset a A*
⟨*proof*⟩

lemma *add-mset-add-mset-same-iff* [*simp*]:
add-mset a A = *add-mset a B* ⟷ *A* = *B*
⟨*proof*⟩

lemma *add-mset-commute*:
add-mset x (*add-mset y M*) = *add-mset y* (*add-mset x M*)
⟨*proof*⟩

31.3 Basic operations

31.3.1 Conversion to set and membership

definition *set-mset* :: *'a multiset* ⇒ *'a set*
where *set-mset M* = {*x*. *count M x* > *0*}

abbreviation *Melem* :: *'a* ⇒ *'a multiset* ⇒ *bool*
where *Melem a M* ≡ *a* ∈ *set-mset M*

notation
Melem (*op* ∈#) **and**
Melem ((-/ ∈# -) [*51*, *51*] *50*)

notation (*ASCII*)
Melem (*op* :#) **and**
Melem ((-/ :# -) [*51*, *51*] *50*)

abbreviation *not-Melem* :: *'a* ⇒ *'a multiset* ⇒ *bool*
where *not-Melem a M* ≡ *a* ∉ *set-mset M*

notation
not-Melem (*op* ∉#) **and**
not-Melem ((-/ ∉# -) [*51*, *51*] *50*)

notation (*ASCII*)
not-Melem (*op* ~:#) **and**
not-Melem ((-/ ~:# -) [*51*, *51*] *50*)

context
begin

qualified abbreviation *Ball* :: *'a multiset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where *Ball M* ≡ *Set.Ball* (*set-mset M*)

qualified abbreviation *Bex* :: *'a multiset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where *Bex M* ≡ *Set.Bex* (*set-mset M*)

end

syntax
-MBall :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*∀ -∈#-./ -) [*0*, *0*, *10*] *10*)
-MBex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*∃ -∈#-./ -) [*0*, *0*, *10*] *10*)

syntax (*ASCII*)
-MBall :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*∀ -:#-./ -) [*0*, *0*, *10*] *10*)
-MBex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*∃ -:#-./ -) [*0*, *0*, *10*] *10*)

translations
∀ *x*∈#*A*. *P* ⇌ *CONST Multiset.Ball A* (λ*x*. *P*)
∃ *x*∈#*A*. *P* ⇌ *CONST Multiset.Bex A* (λ*x*. *P*)

lemma *count-eq-zero-iff*:
count M x = *0* ⟷ *x* ∉# *M*
⟨*proof*⟩

lemma *not-in-iff*:
x ∉# *M* ⟷ *count M x* = *0*
⟨*proof*⟩

lemma *count-greater-zero-iff* [*simp*]:
count M x > *0* ⟷ *x* ∈# *M*
⟨*proof*⟩

lemma *count-inI*:
assumes *count M x* = *0* ⟹ *False*
shows *x* ∈# *M*
⟨*proof*⟩

lemma *in-countE*:

assumes $x \in\# M$
obtains n **where** $count\ M\ x = Suc\ n$
⟨*proof*⟩

lemma *count-greater-eq-Suc-zero-iff* [*simp*]:
$count\ M\ x \geq Suc\ 0 \longleftrightarrow x \in\# M$
⟨*proof*⟩

lemma *count-greater-eq-one-iff* [*simp*]:
$count\ M\ x \geq 1 \longleftrightarrow x \in\# M$
⟨*proof*⟩

lemma *set-mset-empty* [*simp*]:
$set\text{-}mset\ \{\#\} = \{\}$
⟨*proof*⟩

lemma *set-mset-single*:
$set\text{-}mset\ \{\#b\#\} = \{b\}$
⟨*proof*⟩

lemma *set-mset-eq-empty-iff* [*simp*]:
$set\text{-}mset\ M = \{\} \longleftrightarrow M = \{\#\}$
⟨*proof*⟩

lemma *finite-set-mset* [*iff*]:
$finite\ (set\text{-}mset\ M)$
⟨*proof*⟩

lemma *set-mset-add-mset-insert* [*simp*]: ‹$set\text{-}mset\ (add\text{-}mset\ a\ A) = insert\ a\ (set\text{-}mset\ A)$›
⟨*proof*⟩

lemma *multiset-nonemptyE* [*elim*]:
assumes $A \neq \{\#\}$
obtains x **where** $x \in\# A$
⟨*proof*⟩

31.3.2 Union

lemma *count-union* [*simp*]:
$count\ (M + N)\ a = count\ M\ a + count\ N\ a$
⟨*proof*⟩

lemma *set-mset-union* [*simp*]:
$set\text{-}mset\ (M + N) = set\text{-}mset\ M \cup set\text{-}mset\ N$
⟨*proof*⟩

lemma *union-mset-add-mset-left* [*simp*]:
$add\text{-}mset\ a\ A + B = add\text{-}mset\ a\ (A + B)$

⟨*proof*⟩

lemma *union-mset-add-mset-right* [*simp*]:
A + *add-mset a B* = *add-mset a* (*A* + *B*)
⟨*proof*⟩

lemma *add-mset-add-single*: ‹*add-mset a A* = *A* + {#*a*#}›
⟨*proof*⟩

31.3.3 Difference

instance *multiset* :: (*type*) *comm-monoid-diff*
⟨*proof*⟩

lemma *count-diff* [*simp*]:
count (*M* − *N*) *a* = *count M a* − *count N a*
⟨*proof*⟩

lemma *add-mset-diff-bothsides*:
‹*add-mset a M* − *add-mset a A* = *M* − *A*›
⟨*proof*⟩

lemma *in-diff-count*:
a ∈# *M* − *N* ⟷ *count N a* < *count M a*
⟨*proof*⟩

lemma *count-in-diffI*:
assumes ⋀*n*. *count N x* = *n* + *count M x* ⟹ *False*
shows *x* ∈# *M* − *N*
⟨*proof*⟩

lemma *in-diff-countE*:
assumes *x* ∈# *M* − *N*
obtains *n* **where** *count M x* = *Suc n* + *count N x*
⟨*proof*⟩

lemma *in-diffD*:
assumes *a* ∈# *M* − *N*
shows *a* ∈# *M*
⟨*proof*⟩

lemma *set-mset-diff*:
set-mset (*M* − *N*) = {*a*. *count N a* < *count M a*}
⟨*proof*⟩

lemma *diff-empty* [*simp*]: *M* − {#} = *M* ∧ {#} − *M* = {#}
⟨*proof*⟩

lemma *diff-cancel*: *A* − *A* = {#}

⟨*proof*⟩

lemma *diff-union-cancelR*: $M + N - N = (M::'a\ multiset)$
⟨*proof*⟩

lemma *diff-union-cancelL*: $N + M - N = (M::'a\ multiset)$
⟨*proof*⟩

lemma *diff-right-commute*:
fixes $M\ N\ Q :: 'a\ multiset$
shows $M - N - Q = M - Q - N$
⟨*proof*⟩

lemma *diff-add*:
fixes $M\ N\ Q :: 'a\ multiset$
shows $M - (N + Q) = M - N - Q$
⟨*proof*⟩

lemma *insert-DiffM* [*simp*]: $x \in\# M \Longrightarrow add\text{-}mset\ x\ (M - \{\#x\#\}) = M$
⟨*proof*⟩

lemma *insert-DiffM2*: $x \in\# M \Longrightarrow (M - \{\#x\#\}) + \{\#x\#\} = M$
⟨*proof*⟩

lemma *diff-union-swap*: $a \neq b \Longrightarrow add\text{-}mset\ b\ (M - \{\#a\#\}) = add\text{-}mset\ b\ M - \{\#a\#\}$
⟨*proof*⟩

lemma *diff-add-mset-swap* [*simp*]: $b \notin\# A \Longrightarrow add\text{-}mset\ b\ M - A = add\text{-}mset\ b\ (M - A)$
⟨*proof*⟩

lemma *diff-union-swap2* [*simp*]: $y \in\# M \Longrightarrow add\text{-}mset\ x\ M - \{\#y\#\} = add\text{-}mset\ x\ (M - \{\#y\#\})$
⟨*proof*⟩

lemma *diff-diff-add-mset* [*simp*]: $(M::'a\ multiset) - N - P = M - (N + P)$
⟨*proof*⟩

lemma *diff-union-single-conv*:
$a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

lemma *mset-add* [*elim?*]:
assumes $a \in\# A$
obtains B **where** $A = add\text{-}mset\ a\ B$
⟨*proof*⟩

lemma *union-iff*:

$a \in\# A + B \longleftrightarrow a \in\# A \vee a \in\# B$
⟨*proof*⟩

31.3.4 Equality of multisets

lemma *single-eq-single* [*simp*]: $\{\#a\#\} = \{\#b\#\} \longleftrightarrow a = b$
⟨*proof*⟩

lemma *union-eq-empty* [*iff*]: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
⟨*proof*⟩

lemma *empty-eq-union* [*iff*]: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
⟨*proof*⟩

lemma *multi-self-add-other-not-self* [*simp*]: $M = \textit{add-mset}\ x\ M \longleftrightarrow \textit{False}$
⟨*proof*⟩

lemma *add-mset-remove-trivial* [*simp*]: ‹*add-mset x M* − {#*x*#} = *M*›
⟨*proof*⟩

lemma *diff-single-trivial*: $\neg\ x \in\# M \Longrightarrow M - \{\#x\#\} = M$
⟨*proof*⟩

lemma *diff-single-eq-union*: $x \in\# M \Longrightarrow M - \{\#x\#\} = N \longleftrightarrow M = \textit{add-mset}\ x\ N$
⟨*proof*⟩

lemma *union-single-eq-diff*: $\textit{add-mset}\ x\ M = N \Longrightarrow M = N - \{\#x\#\}$
⟨*proof*⟩

lemma *union-single-eq-member*: $\textit{add-mset}\ x\ M = N \Longrightarrow x \in\# N$
⟨*proof*⟩

lemma *add-mset-remove-trivial-If*:
$\textit{add-mset}\ a\ (N - \{\#a\#\}) = (\textit{if}\ a \in\# N\ \textit{then}\ N\ \textit{else}\ \textit{add-mset}\ a\ N)$
⟨*proof*⟩

lemma *add-mset-remove-trivial-eq*: ‹*N* = *add-mset a* (*N* − {#*a*#}) ⟷ *a* ∈# *N*›
⟨*proof*⟩

lemma *union-is-single*:
$M + N = \{\#a\#\} \longleftrightarrow M = \{\#a\#\} \wedge N = \{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\}$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *single-is-union*: $\{\#a\#\} = M + N \longleftrightarrow \{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N$
⟨*proof*⟩

lemma *add-eq-conv-diff*:
add-mset a M = add-mset b N ⟷ M = N ∧ a = b ∨ M = add-mset b (N − {#a#}) ∧ N = add-mset a (M − {#b#})
(is *?lhs ⟷ ?rhs*)

⟨*proof*⟩

lemma *add-mset-eq-single* [*iff*]: *add-mset b M = {#a#} ⟷ b = a ∧ M = {#}*
⟨*proof*⟩

lemma *single-eq-add-mset* [*iff*]: *{#a#} = add-mset b M ⟷ b = a ∧ M = {#}*
⟨*proof*⟩

lemma *insert-noteq-member*:
assumes *BC*: *add-mset b B = add-mset c C*
and *bnotc*: *b ≠ c*
shows *c ∈# B*
⟨*proof*⟩

lemma *add-eq-conv-ex*:
(add-mset a M = add-mset b N) =
(M = N ∧ a = b ∨ (∃ K. M = add-mset b K ∧ N = add-mset a K))
⟨*proof*⟩

lemma *multi-member-split*: *x ∈# M ⟹ ∃ A. M = add-mset x A*
⟨*proof*⟩

lemma *multiset-add-sub-el-shuffle*:
assumes *c ∈# B*
and *b ≠ c*
shows *add-mset b (B − {#c#}) = add-mset b B − {#c#}*
⟨*proof*⟩

lemma *add-mset-eq-singleton-iff* [*iff*]:
add-mset x M = {#y#} ⟷ M = {#} ∧ x = y
⟨*proof*⟩

31.3.5 Pointwise ordering induced by count

definition *subseteq-mset* :: *'a multiset ⇒ 'a multiset ⇒ bool* (**infix** *⊆# 50*)
where *A ⊆# B = (∀ a. count A a ≤ count B a)*

definition *subset-mset* :: *'a multiset ⇒ 'a multiset ⇒ bool* (**infix** *⊂# 50*)
where *A ⊂# B = (A ⊆# B ∧ A ≠ B)*

abbreviation (*input*) *supseteq-mset* :: *'a multiset ⇒ 'a multiset ⇒ bool* (**infix** *⊇# 50*)
where *supseteq-mset A B ≡ B ⊆# A*

abbreviation (*input*) *supset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ⊃# *50*)
where *supset-mset A B* ≡ *B* ⊂# *A*

notation (*input*)
subseteq-mset (**infix** ≤# *50*) **and**
supseteq-mset (**infix** ≥# *50*)

notation (*ASCII*)
subseteq-mset (**infix** <=# *50*) **and**
subset-mset (**infix** <# *50*) **and**
supseteq-mset (**infix** >=# *50*) **and**
supset-mset (**infix** ># *50*)

interpretation *subset-mset*: *ordered-ab-semigroup-add-imp-le op* + *op* − *op* ⊆# *op* ⊂#
⟨*proof*⟩

interpretation *subset-mset*: *ordered-ab-semigroup-monoid-add-imp-le op* + *0 op* − *op* ≤# *op* <#
⟨*proof*⟩

lemma *mset-subset-eqI*:
(⋀*a*. *count A a* ≤ *count B a*) ⟹ *A* ⊆# *B*
⟨*proof*⟩

lemma *mset-subset-eq-count*:
A ⊆# *B* ⟹ *count A a* ≤ *count B a*
⟨*proof*⟩

lemma *mset-subset-eq-exists-conv*: (*A*::*'a multiset*) ⊆# *B* ⟷ (∃ *C*. *B* = *A* + *C*)
⟨*proof*⟩

interpretation *subset-mset*: *ordered-cancel-comm-monoid-diff op* + *0 op* ≤# *op* <# *op* −
⟨*proof*⟩

declare *subset-mset.add-diff-assoc*[*simp*] *subset-mset.add-diff-assoc2*[*simp*]

lemma *mset-subset-eq-mono-add-right-cancel*: (*A*::*'a multiset*) + *C* ⊆# *B* + *C* ⟷ *A* ⊆# *B*
⟨*proof*⟩

lemma *mset-subset-eq-mono-add-left-cancel*: *C* + (*A*::*'a multiset*) ⊆# *C* + *B* ⟷ *A* ⊆# *B*
⟨*proof*⟩

lemma *mset-subset-eq-mono-add*: (*A*::*'a multiset*) ⊆# *B* ⟹ *C* ⊆# *D* ⟹ *A* +

$C \subseteq\# B + D$
⟨*proof*⟩

lemma *mset-subset-eq-add-left*: $(A::'a\ multiset) \subseteq\# A + B$
⟨*proof*⟩

lemma *mset-subset-eq-add-right*: $B \subseteq\# (A::'a\ multiset) + B$
⟨*proof*⟩

lemma *single-subset-iff* [*simp*]:
$\{\#a\#\} \subseteq\# M \longleftrightarrow a \in\# M$
⟨*proof*⟩

lemma *mset-subset-eq-single*: $a \in\# B \Longrightarrow \{\#a\#\} \subseteq\# B$
⟨*proof*⟩

lemma *mset-subset-eq-add-mset-cancel*: ‹*add-mset a A* $\subseteq\#$ *add-mset a B* $\longleftrightarrow$ *A* $\subseteq\#$ *B*›
⟨*proof*⟩

lemma *multiset-diff-union-assoc*:
fixes *A B C D* :: *'a multiset*
shows $C \subseteq\# B \Longrightarrow A + B - C = A + (B - C)$
⟨*proof*⟩

lemma *mset-subset-eq-multiset-union-diff-commute*:
fixes *A B C D* :: *'a multiset*
shows $B \subseteq\# A \Longrightarrow A - B + C = A + C - B$
⟨*proof*⟩

lemma *diff-subset-eq-self* [*simp*]:
$(M::'a\ multiset) - N \subseteq\# M$
⟨*proof*⟩

lemma *mset-subset-eqD*:
assumes $A \subseteq\# B$ **and** $x \in\# A$
shows $x \in\# B$
⟨*proof*⟩

lemma *mset-subsetD*:
$A \subset\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
⟨*proof*⟩

lemma *set-mset-mono*:
$A \subseteq\# B \Longrightarrow$ *set-mset A* $\subseteq$ *set-mset B*
⟨*proof*⟩

lemma *mset-subset-eq-insertD*:
add-mset x A $\subseteq\# B \Longrightarrow x \in\# B \wedge A \subset\# B$

⟨*proof*⟩

lemma *mset-subset-insertD*:
add-mset x A ⊂# *B* ⟹ *x* ∈# *B* ∧ *A* ⊂# *B*
⟨*proof*⟩

lemma *mset-subset-of-empty*[*simp*]: *A* ⊂# {#} ⟷ *False*
⟨*proof*⟩

lemma *empty-subset-add-mset*[*simp*]: {#} <# *add-mset x M*
⟨*proof*⟩

lemma *empty-le*: {#} ⊆# *A*
⟨*proof*⟩

lemma *insert-subset-eq-iff*:
add-mset a A ⊆# *B* ⟷ *a* ∈# *B* ∧ *A* ⊆# *B* − {#*a*#}
⟨*proof*⟩

lemma *insert-union-subset-iff*:
add-mset a A ⊂# *B* ⟷ *a* ∈# *B* ∧ *A* ⊂# *B* − {#*a*#}
⟨*proof*⟩

lemma *subset-eq-diff-conv*:
A − *C* ⊆# *B* ⟷ *A* ⊆# *B* + *C*
⟨*proof*⟩

lemma *multi-psub-of-add-self* [*simp*]: *A* ⊂# *add-mset x A*
⟨*proof*⟩

lemma *multi-psub-self*: *A* ⊂# *A* = *False*
⟨*proof*⟩

lemma *mset-subset-add-mset* [*simp*]: *add-mset x N* ⊂# *add-mset x M* ⟷ *N* ⊂# *M*
⟨*proof*⟩

lemma *mset-subset-diff-self*: *c* ∈# *B* ⟹ *B* − {#*c*#} ⊂# *B*
⟨*proof*⟩

lemma *Diff-eq-empty-iff-mset*: *A* − *B* = {#} ⟷ *A* ⊆# *B*
⟨*proof*⟩

lemma *add-mset-subseteq-single-iff*[*iff*]: *add-mset a M* ⊆# {#*b*#} ⟷ *M* = {#} ∧ *a* = *b*
⟨*proof*⟩

31.3.6 Intersection and bounded union

definition *inf-subset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* (**infixl** ∩# *70*) **where**
multiset-inter-def: *inf-subset-mset A B* = *A* − (*A* − *B*)

interpretation *subset-mset*: *semilattice-inf inf-subset-mset op* ⊆# *op* ⊂#
⟨*proof*⟩

definition *sup-subset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset*(**infixl** ∪# *70*)
where *sup-subset-mset A B* = *A* + (*B* − *A*) — FIXME irregular fact name

interpretation *subset-mset*: *semilattice-sup sup-subset-mset op* ⊆# *op* ⊂#
⟨*proof*⟩

interpretation *subset-mset*: *bounded-lattice-bot op* ∩# *op* ⊆# *op* ⊂#
op ∪# {#}
⟨*proof*⟩

31.3.7 Additional intersection facts

lemma *multiset-inter-count* [*simp*]:
fixes *A B* :: *'a multiset*
shows *count* (*A* ∩# *B*) *x* = *min* (*count A x*) (*count B x*)
⟨*proof*⟩

lemma *set-mset-inter* [*simp*]:
set-mset (*A* ∩# *B*) = *set-mset A* ∩ *set-mset B*
⟨*proof*⟩

lemma *diff-intersect-left-idem* [*simp*]:
M − *M* ∩# *N* = *M* − *N*
⟨*proof*⟩

lemma *diff-intersect-right-idem* [*simp*]:
M − *N* ∩# *M* = *M* − *N*
⟨*proof*⟩

lemma *multiset-inter-single*[*simp*]: *a* ≠ *b* ⟹ {#*a*#} ∩# {#*b*#} = {#}
⟨*proof*⟩

lemma *multiset-union-diff-commute*:
assumes *B* ∩# *C* = {#}
shows *A* + *B* − *C* = *A* − *C* + *B*
⟨*proof*⟩

lemma *disjunct-not-in*:
A ∩# *B* = {#} ⟷ (∀ *a*. *a* ∉# *A* ∨ *a* ∉# *B*) (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *inter-mset-empty-distrib-right*: $A \cap\# (B + C) = \{\#\} \longleftrightarrow A \cap\# B = \{\#\} \wedge A \cap\# C = \{\#\}$
⟨*proof*⟩

lemma *inter-mset-empty-distrib-left*: $(A + B) \cap\# C = \{\#\} \longleftrightarrow A \cap\# C = \{\#\} \wedge B \cap\# C = \{\#\}$
⟨*proof*⟩

lemma *add-mset-inter-add-mset*[*simp*]:
$add\text{-}mset\ a\ A \cap\# add\text{-}mset\ a\ B = add\text{-}mset\ a\ (A \cap\# B)$
⟨*proof*⟩

lemma *add-mset-disjoint* [*simp*]:
$add\text{-}mset\ a\ A \cap\# B = \{\#\} \longleftrightarrow a \notin\# B \wedge A \cap\# B = \{\#\}$
$\{\#\} = add\text{-}mset\ a\ A \cap\# B \longleftrightarrow a \notin\# B \wedge \{\#\} = A \cap\# B$
⟨*proof*⟩

lemma *disjoint-add-mset* [*simp*]:
$B \cap\# add\text{-}mset\ a\ A = \{\#\} \longleftrightarrow a \notin\# B \wedge B \cap\# A = \{\#\}$
$\{\#\} = A \cap\# add\text{-}mset\ b\ B \longleftrightarrow b \notin\# A \wedge \{\#\} = A \cap\# B$
⟨*proof*⟩

lemma *inter-add-left1*: $\neg\ x \in\# N \Longrightarrow (add\text{-}mset\ x\ M) \cap\# N = M \cap\# N$
⟨*proof*⟩

lemma *inter-add-left2*: $x \in\# N \Longrightarrow (add\text{-}mset\ x\ M) \cap\# N = add\text{-}mset\ x\ (M \cap\# (N - \{\#x\#\}))$
⟨*proof*⟩

lemma *inter-add-right1*: $\neg\ x \in\# N \Longrightarrow N \cap\# (add\text{-}mset\ x\ M) = N \cap\# M$
⟨*proof*⟩

lemma *inter-add-right2*: $x \in\# N \Longrightarrow N \cap\# (add\text{-}mset\ x\ M) = add\text{-}mset\ x\ ((N - \{\#x\#\}) \cap\# M)$
⟨*proof*⟩

lemma *disjunct-set-mset-diff*:
assumes $M \cap\# N = \{\#\}$
shows $set\text{-}mset\ (M - N) = set\text{-}mset\ M$
⟨*proof*⟩

lemma *at-most-one-mset-mset-diff*:
assumes $a \notin\# M - \{\#a\#\}$
shows $set\text{-}mset\ (M - \{\#a\#\}) = set\text{-}mset\ M - \{a\}$
⟨*proof*⟩

lemma *more-than-one-mset-mset-diff*:
assumes $a \in\# M - \{\#a\#\}$

shows *set-mset* $(M - \{\#a\#\}) = \textit{set-mset}\ M$
⟨*proof*⟩

lemma *inter-iff*:
$a \in\# A \cap\# B \longleftrightarrow a \in\# A \wedge a \in\# B$
⟨*proof*⟩

lemma *inter-union-distrib-left*:
$A \cap\# B + C = (A + C) \cap\# (B + C)$
⟨*proof*⟩

lemma *inter-union-distrib-right*:
$C + A \cap\# B = (C + A) \cap\# (C + B)$
⟨*proof*⟩

lemma *inter-subset-eq-union*:
$A \cap\# B \subseteq\# A + B$
⟨*proof*⟩

31.3.8 Additional bounded union facts

lemma *sup-subset-mset-count* [*simp*]: — FIXME irregular fact name
count $(A \cup\# B)\ x = \textit{max}\ (\textit{count}\ A\ x)\ (\textit{count}\ B\ x)$
⟨*proof*⟩

lemma *set-mset-sup* [*simp*]:
set-mset $(A \cup\# B) = \textit{set-mset}\ A \cup \textit{set-mset}\ B$
⟨*proof*⟩

lemma *sup-union-left1* [*simp*]: $\neg\ x \in\# N \Longrightarrow (\textit{add-mset}\ x\ M) \cup\# N = \textit{add-mset}\ x\ (M \cup\# N)$
⟨*proof*⟩

lemma *sup-union-left2*: $x \in\# N \Longrightarrow (\textit{add-mset}\ x\ M) \cup\# N = \textit{add-mset}\ x\ (M \cup\# (N - \{\#x\#\}))$
⟨*proof*⟩

lemma *sup-union-right1* [*simp*]: $\neg\ x \in\# N \Longrightarrow N \cup\# (\textit{add-mset}\ x\ M) = \textit{add-mset}\ x\ (N \cup\# M)$
⟨*proof*⟩

lemma *sup-union-right2*: $x \in\# N \Longrightarrow N \cup\# (\textit{add-mset}\ x\ M) = \textit{add-mset}\ x\ ((N - \{\#x\#\}) \cup\# M)$
⟨*proof*⟩

lemma *sup-union-distrib-left*:
$A \cup\# B + C = (A + C) \cup\# (B + C)$
⟨*proof*⟩

lemma *union-sup-distrib-right*:
$C + A \cup\# B = (C + A) \cup\# (C + B)$
⟨*proof*⟩

lemma *union-diff-inter-eq-sup*:
$A + B - A \cap\# B = A \cup\# B$
⟨*proof*⟩

lemma *union-diff-sup-eq-inter*:
$A + B - A \cup\# B = A \cap\# B$
⟨*proof*⟩

lemma *add-mset-union*:
‹*add-mset a A* ∪# *add-mset a B* = *add-mset a* (*A* ∪# *B*)›
⟨*proof*⟩

31.3.9 Subset is an order

interpretation *subset-mset*: *order op* ⊆# *op* ⊂# ⟨*proof*⟩

31.4 Replicate and repeat operations

definition *replicate-mset* :: *nat* ⇒ *'a* ⇒ *'a multiset* **where**
replicate-mset n x = (*add-mset x* ^^ *n*) {#}

lemma *replicate-mset-0*[*simp*]: *replicate-mset 0 x* = {#}
⟨*proof*⟩

lemma *replicate-mset-Suc* [*simp*]: *replicate-mset* (*Suc n*) *x* = *add-mset x* (*replicate-mset n x*)
⟨*proof*⟩

lemma *count-replicate-mset*[*simp*]: *count* (*replicate-mset n x*) *y* = (*if y* = *x then n else 0*)
⟨*proof*⟩

fun *repeat-mset* :: *nat* ⇒ *'a multiset* ⇒ *'a multiset* **where**
repeat-mset 0 - = {#} |
repeat-mset (*Suc n*) *A* = *A* + *repeat-mset n A*

lemma *count-repeat-mset* [*simp*]: *count* (*repeat-mset i A*) *a* = *i* ∗ *count A a*
⟨*proof*⟩

lemma *repeat-mset-right* [*simp*]: *repeat-mset a* (*repeat-mset b A*) = *repeat-mset* (*a* ∗ *b*) *A*
⟨*proof*⟩

lemma *left-diff-repeat-mset-distrib'*: ‹*repeat-mset* (*i* − *j*) *u* = *repeat-mset i u* − *repeat-mset j u*›
⟨*proof*⟩

lemma *left-add-mult-distrib-mset*:
repeat-mset i u + (repeat-mset j u + k) = repeat-mset (i+j) u + k
⟨*proof*⟩

lemma *repeat-mset-distrib*:
repeat-mset (m + n) A = repeat-mset m A + repeat-mset n A
⟨*proof*⟩

lemma *repeat-mset-distrib2*[*simp*]:
repeat-mset n (A + B) = repeat-mset n A + repeat-mset n B
⟨*proof*⟩

lemma *repeat-mset-replicate-mset*[*simp*]:
repeat-mset n {#a#} = replicate-mset n a
⟨*proof*⟩

lemma *repeat-mset-distrib-add-mset*[*simp*]:
repeat-mset n (add-mset a A) = replicate-mset n a + repeat-mset n A
⟨*proof*⟩

lemma *repeat-mset-empty*[*simp*]: *repeat-mset n {#} = {#}*
⟨*proof*⟩

31.4.1 Simprocs

lemma *mset-diff-add-eq1*:
j ≤ (i::nat) ⟹ ((repeat-mset i u + m) − (repeat-mset j u + n)) = ((repeat-mset (i−j) u + m) − n)
⟨*proof*⟩

lemma *mset-diff-add-eq2*:
i ≤ (j::nat) ⟹ ((repeat-mset i u + m) − (repeat-mset j u + n)) = (m − (repeat-mset (j−i) u + n))
⟨*proof*⟩

lemma *mset-eq-add-iff1*:
j ≤ (i::nat) ⟹ (repeat-mset i u + m = repeat-mset j u + n) = (repeat-mset (i−j) u + m = n)
⟨*proof*⟩

lemma *mset-eq-add-iff2*:
i ≤ (j::nat) ⟹ (repeat-mset i u + m = repeat-mset j u + n) = (m = repeat-mset (j−i) u + n)
⟨*proof*⟩

lemma *mset-subseteq-add-iff1*:
j ≤ (i::nat) ⟹ (repeat-mset i u + m ⊆# repeat-mset j u + n) = (repeat-mset (i−j) u + m ⊆# n)

⟨*proof*⟩

lemma *mset-subseteq-add-iff2*:
i ≤ (*j*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊆# *repeat-mset j u* + *n*) = (*m* ⊆# *repeat-mset* (*j*−*i*) *u* + *n*)
⟨*proof*⟩

lemma *mset-subset-add-iff1*:
j ≤ (*i*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊂# *repeat-mset j u* + *n*) = (*repeat-mset* (*i*−*j*) *u* + *m* ⊂# *n*)
⟨*proof*⟩

lemma *mset-subset-add-iff2*:
i ≤ (*j*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊂# *repeat-mset j u* + *n*) = (*m* ⊂# *repeat-mset* (*j*−*i*) *u* + *n*)
⟨*proof*⟩

⟨*ML*⟩

31.4.2 Conditionally complete lattice

instantiation *multiset* :: (*type*) *Inf*
begin

lift-definition *Inf-multiset* :: *'a multiset set* ⇒ *'a multiset* **is**
λ*A i*. *if A* = {} *then 0 else Inf* ((λ*f*. *f i*) ‘ *A*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *Inf-multiset-empty*: *Inf* {} = {#}
⟨*proof*⟩

lemma *count-Inf-multiset-nonempty*: *A* ≠ {} ⟹ *count* (*Inf A*) *x* = *Inf* ((λ*X*. *count X x*) ‘ *A*)
⟨*proof*⟩

instantiation *multiset* :: (*type*) *Sup*
begin

definition *Sup-multiset* :: *'a multiset set* ⇒ *'a multiset* **where**
Sup-multiset A = (*if A* ≠ {} ∧ *subset-mset.bdd-above A then*
Abs-multiset (λ*i*. *Sup* ((λ*X*. *count X i*) ‘ *A*)) *else* {#})

lemma *Sup-multiset-empty*: *Sup* {} = {#}
⟨*proof*⟩

lemma *Sup-multiset-unbounded*: $\neg$*subset-mset.bdd-above* $A \Longrightarrow$ *Sup* $A = \{\#\}$
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *bdd-above-multiset-imp-bdd-above-count*:
assumes *subset-mset.bdd-above* (A :: $'a$ *multiset set*)
shows *bdd-above* ((λX. *count* X x) ‘ A)
⟨*proof*⟩

lemma *bdd-above-multiset-imp-finite-support*:
assumes $A \neq \{\}$ *subset-mset.bdd-above* (A :: $'a$ *multiset set*)
shows *finite* ($\bigcup X{\in}A.$ $\{x.$ *count* X $x > 0\}$)
⟨*proof*⟩

lemma *Sup-multiset-in-multiset*:
assumes $A \neq \{\}$ *subset-mset.bdd-above* A
shows (λi. *SUP* X:A. *count* X i) $\in$ *multiset*
⟨*proof*⟩

lemma *count-Sup-multiset-nonempty*:
assumes $A \neq \{\}$ *subset-mset.bdd-above* A
shows *count* (*Sup* A) x = (*SUP* X:A. *count* X x)
⟨*proof*⟩

interpretation *subset-mset*: *conditionally-complete-lattice Inf Sup op* $\cap\#$ *op* $\subseteq\#$ *op* $\subset\#$ *op* $\cup\#$
⟨*proof*⟩

lemma *set-mset-Inf*:
assumes $A \neq \{\}$
shows *set-mset* (*Inf* A) = ($\bigcap X{\in}A.$ *set-mset* X)
⟨*proof*⟩

lemma *in-Inf-multiset-iff*:
assumes $A \neq \{\}$
shows $x \in\#$ *Inf* $A \longleftrightarrow$ ($\forall\, X{\in}A.$ $x \in\#$ X)
⟨*proof*⟩

lemma *in-Inf-multisetD*: $x \in\#$ *Inf* $A \Longrightarrow X \in A \Longrightarrow x \in\#$ X
⟨*proof*⟩

lemma *set-mset-Sup*:
assumes *subset-mset.bdd-above* A

shows *set-mset* (*Sup A*) = ($\bigcup X{\in}A.$ *set-mset X*)
⟨*proof*⟩

lemma *in-Sup-multiset-iff*:
assumes *subset-mset.bdd-above A*
shows $x \in\# \mathit{Sup}\ A \longleftrightarrow (\exists X{\in}A.\ x \in\# X)$
⟨*proof*⟩

lemma *in-Sup-multisetD*:
assumes $x \in\# \mathit{Sup}\ A$
shows $\exists X{\in}A.\ x \in\# X$
⟨*proof*⟩

interpretation *subset-mset*: *distrib-lattice op* $\cap\#$ *op* $\subseteq\#$ *op* $\subset\#$ *op* $\cup\#$
⟨*proof*⟩

31.4.3 Filter (with comprehension syntax)

Multiset comprehension

lift-definition *filter-mset* :: ('*a* ⇒ *bool*) ⇒ '*a multiset* ⇒ '*a multiset*
is $\lambda P\ M.\ \lambda x.$ *if P x then M x else 0*
⟨*proof*⟩

syntax (*ASCII*)
-MCollect :: *pttrn* ⇒ '*a multiset* ⇒ *bool* ⇒ '*a multiset* ((*1*{#- :# -./ -#}))
syntax
-MCollect :: *pttrn* ⇒ '*a multiset* ⇒ *bool* ⇒ '*a multiset* ((*1*{#- ∈# -./ -#}))
translations
{#*x* ∈# *M*. *P*#} == *CONST filter-mset* (λ*x*. *P*) *M*

lemma *count-filter-mset* [*simp*]:
count (*filter-mset P M*) *a* = (*if P a then count M a else 0*)
⟨*proof*⟩

lemma *set-mset-filter* [*simp*]:
set-mset (*filter-mset P M*) = {*a* ∈ *set-mset M*. *P a*}
⟨*proof*⟩

lemma *filter-empty-mset* [*simp*]: *filter-mset P* {#} = {#}
⟨*proof*⟩

lemma *filter-single-mset*: *filter-mset P* {#*x*#} = (*if P x then* {#*x*#} *else* {#})
⟨*proof*⟩

lemma *filter-union-mset* [*simp*]: *filter-mset P* (*M* + *N*) = *filter-mset P M* + *filter-mset P N*
⟨*proof*⟩

lemma *filter-diff-mset* [*simp*]: *filter-mset P* (*M* − *N*) = *filter-mset P M* − *filter-mset*

P N
⟨*proof*⟩

lemma *filter-inter-mset* [*simp*]: *filter-mset P* (*M* ∩# *N*) = *filter-mset P M* ∩# *filter-mset P N*
⟨*proof*⟩

lemma *filter-sup-mset*[*simp*]: *filter-mset P* (*A* ∪# *B*) = *filter-mset P A* ∪# *filter-mset P B*
⟨*proof*⟩

lemma *filter-mset-add-mset* [*simp*]:
filter-mset P (*add-mset x A*) =
(*if P x then add-mset x* (*filter-mset P A*) *else filter-mset P A*)
⟨*proof*⟩

lemma *multiset-filter-subset*[*simp*]: *filter-mset f M* ⊆# *M*
⟨*proof*⟩

lemma *multiset-filter-mono*:
assumes *A* ⊆# *B*
shows *filter-mset f A* ⊆# *filter-mset f B*
⟨*proof*⟩

lemma *filter-mset-eq-conv*:
filter-mset P M = *N* ⟷ *N* ⊆# *M* ∧ (∀ *b*∈#*N*. *P b*) ∧ (∀ *a*∈#*M* − *N*. ¬ *P a*)
(**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *filter-filter-mset*: *filter-mset P* (*filter-mset Q M*) = {#*x* ∈# *M*. *Q x* ∧ *P x*#}
⟨*proof*⟩

lemma
filter-mset-True[*simp*]: {#*y* ∈# *M*. *True*#} = *M* **and**
filter-mset-False[*simp*]: {#*y* ∈# *M*. *False*#} = {#}
⟨*proof*⟩

31.4.4 Size

definition *wcount* **where** *wcount f M* = (λ*x*. *count M x* ∗ *Suc* (*f x*))

lemma *wcount-union*: *wcount f* (*M* + *N*) *a* = *wcount f M a* + *wcount f N a*
⟨*proof*⟩

lemma *wcount-add-mset*:
wcount f (*add-mset x M*) *a* = (*if x* = *a then Suc* (*f a*) *else 0*) + *wcount f M a*
⟨*proof*⟩

definition *size-multiset* :: $('a \Rightarrow nat) \Rightarrow 'a\ multiset \Rightarrow nat$ **where**
size-multiset f M = *sum* (*wcount f M*) (*set-mset M*)

lemmas *size-multiset-eq* = *size-multiset-def*[*unfolded wcount-def*]

instantiation *multiset* :: (*type*) *size*
begin

definition *size-multiset* **where**
size-multiset-overloaded-def: *size-multiset* = *Multiset.size-multiset* (λ-. *0*)
instance ⟨*proof*⟩

end

lemmas *size-multiset-overloaded-eq* =
size-multiset-overloaded-def[*THEN fun-cong*, *unfolded size-multiset-eq*, *simplified*]

lemma *size-multiset-empty* [*simp*]: *size-multiset f* {#} = *0*
⟨*proof*⟩

lemma *size-empty* [*simp*]: *size* {#} = *0*
⟨*proof*⟩

lemma *size-multiset-single* : *size-multiset f* {#*b*#} = *Suc* (*f b*)
⟨*proof*⟩

lemma *size-single*: *size* {#*b*#} = *1*
⟨*proof*⟩

lemma *sum-wcount-Int*:
finite A ⟹ *sum* (*wcount f N*) (*A* ∩ *set-mset N*) = *sum* (*wcount f N*) *A*
⟨*proof*⟩

lemma *size-multiset-union* [*simp*]:
size-multiset f (*M* + *N*::'*a multiset*) = *size-multiset f M* + *size-multiset f N*
⟨*proof*⟩

lemma *size-multiset-add-mset* [*simp*]:
size-multiset f (*add-mset a M*) = *Suc* (*f a*) + *size-multiset f M*
⟨*proof*⟩

lemma *size-add-mset* [*simp*]: *size* (*add-mset a A*) = *Suc* (*size A*)
⟨*proof*⟩

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
⟨*proof*⟩

lemma *size-multiset-eq-0-iff-empty* [*iff*]:
size-multiset f M = *0* ⟷ *M* = {#}

⟨*proof*⟩

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
⟨*proof*⟩

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)
⟨*proof*⟩

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ⟹ ∃ *a*. *a* ∈# *M*
⟨*proof*⟩

lemma *size-eq-Suc-imp-eq-union*:
assumes *size M* = *Suc n*
shows ∃ *a N*. *M* = *add-mset a N*
⟨*proof*⟩

lemma *size-mset-mono*:
fixes *A B* :: *'a multiset*
assumes *A* ⊆# *B*
shows *size A* ≤ *size B*
⟨*proof*⟩

lemma *size-filter-mset-lesseq*[*simp*]: *size* (*filter-mset f M*) ≤ *size M*
⟨*proof*⟩

lemma *size-Diff-submset*:
M ⊆# *M'* ⟹ *size* (*M'* − *M*) = *size M'* − *size*(*M*::*'a multiset*)
⟨*proof*⟩

31.5 Induction and case splits

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
assumes *empty*: *P* {#}
assumes *add*: ⋀*x M*. *P M* ⟹ *P* (*add-mset x M*)
shows *P M*
⟨*proof*⟩

lemma *multi-nonempty-split*: *M* ≠ {#} ⟹ ∃ *A a*. *M* = *add-mset a A*
⟨*proof*⟩

lemma *multiset-cases* [*cases type*]:
obtains (*empty*) *M* = {#}
| (*add*) *x N* **where** *M* = *add-mset x N*
⟨*proof*⟩

lemma *multi-drop-mem-not-eq*: *c* ∈# *B* ⟹ *B* − {#*c*#} ≠ *B*
⟨*proof*⟩

lemma *multiset-partition*: *M* = {# *x*∈#*M*. *P x* #} + {# *x*∈#*M*. ¬ *P x* #}

⟨*proof*⟩

lemma *mset-subset-size*: $(A::'a\ multiset) \subset\# B \Longrightarrow size\ A < size\ B$
⟨*proof*⟩

lemma *size-1-singleton-mset*: $size\ M = 1 \Longrightarrow \exists\, a.\ M = \{\#a\#\}$
⟨*proof*⟩

31.5.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

lemma *wf-subset-mset-rel*: *wf* $\{(M, N :: 'a\ multiset).\ M \subset\# N\}$
⟨*proof*⟩

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall (A::'a\ multiset).\ A \subset\# B \longrightarrow P\ A \Longrightarrow P\ B$
shows $P\ B$
⟨*proof*⟩

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \subseteq\# A$
and *empty*: $P\ \{\#\}$
and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (add\text{-}mset\ a\ F)$
shows $P\ F$
⟨*proof*⟩

31.6 The fold combinator

definition *fold-mset* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b$
where
$fold\text{-}mset\ f\ s\ M = Finite\text{-}Set.fold\ (\lambda x.\ f\ x\ \hat{}\hat{}\ count\ M\ x)\ s\ (set\text{-}mset\ M)$

lemma *fold-mset-empty* [*simp*]: $fold\text{-}mset\ f\ s\ \{\#\} = s$
⟨*proof*⟩

context *comp-fun-commute*
begin

lemma *fold-mset-add-mset* [*simp*]: $fold\text{-}mset\ f\ s\ (add\text{-}mset\ x\ M) = f\ x\ (fold\text{-}mset\ f\ s\ M)$
⟨*proof*⟩

corollary *fold-mset-single*: $fold\text{-}mset\ f\ s\ \{\#x\#\} = f\ x\ s$
⟨*proof*⟩

lemma *fold-mset-fun-left-comm*: $f\ x\ (fold\text{-}mset\ f\ s\ M) = fold\text{-}mset\ f\ (f\ x\ s)\ M$
⟨*proof*⟩

lemma *fold-mset-union* [*simp*]: *fold-mset f s* (*M* + *N*) = *fold-mset f* (*fold-mset f s M*) *N*
⟨*proof*⟩

lemma *fold-mset-fusion*:
assumes *comp-fun-commute g*
and ∗: ⋀*x y*. *h* (*g x y*) = *f x* (*h y*)
shows *h* (*fold-mset g w A*) = *fold-mset f* (*h w*) *A*
⟨*proof*⟩

end

lemma *union-fold-mset-add-mset*: *A* + *B* = *fold-mset add-mset A B*
⟨*proof*⟩

A note on code generation: When defining some function containing a subterm *fold-mset F*, code generation is not automatic. When interpreting locale *left-commutative* with *F*, the would be code thms for *fold-mset* become thms like *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.

31.7 Image

definition *image-mset* :: ('*a* ⇒ '*b*) ⇒ '*a multiset* ⇒ '*b multiset* **where**
image-mset f = *fold-mset* (*add-mset* ∘ *f*) {#}

lemma *comp-fun-commute-mset-image*: *comp-fun-commute* (*add-mset* ∘ *f*)
⟨*proof*⟩

lemma *image-mset-empty* [*simp*]: *image-mset f* {#} = {#}
⟨*proof*⟩

lemma *image-mset-single*: *image-mset f* {#*x*#} = {#*f x*#}
⟨*proof*⟩

lemma *image-mset-union* [*simp*]: *image-mset f* (*M* + *N*) = *image-mset f M* + *image-mset f N*
⟨*proof*⟩

corollary *image-mset-add-mset* [*simp*]:
image-mset f (*add-mset a M*) = *add-mset* (*f a*) (*image-mset f M*)
⟨*proof*⟩

lemma *set-image-mset* [*simp*]: *set-mset* (*image-mset f M*) = *image f* (*set-mset M*)
⟨*proof*⟩

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*

⟨*proof*⟩

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} ⟷ *M* = {#}
⟨*proof*⟩

lemma *image-mset-If*:
image-mset (λ*x*. *if P x then f x else g x*) *A* =
image-mset f (*filter-mset P A*) + *image-mset g* (*filter-mset* (λ*x*. ¬*P x*) *A*)
⟨*proof*⟩

lemma *image-mset-Diff*:
assumes *B* ⊆# *A*
shows *image-mset f* (*A* − *B*) = *image-mset f A* − *image-mset f B*
⟨*proof*⟩

lemma *count-image-mset*:
count (*image-mset f A*) *x* = (∑ *y*∈*f* −‘ {*x*} ∩ *set-mset A*. *count A y*)
⟨*proof*⟩

lemma *image-mset-subseteq-mono*: *A* ⊆# *B* ⟹ *image-mset f A* ⊆# *image-mset f B*
⟨*proof*⟩

syntax (*ASCII*)
-*comprehension-mset* :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset* (({#-/. - :# -#}))
syntax
-*comprehension-mset* :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset* (({#-/. - ∈# -#}))
translations
{#*e*. *x* ∈# *M*#} ⇌ *CONST image-mset* (λ*x*. *e*) *M*

syntax (*ASCII*)
-*comprehension-mset'* :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset* (({#-/ | - :# -./ -#}))
syntax
-*comprehension-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*#}#}.

lemma *in-image-mset*: *y* ∈# {#*f x*. *x* ∈# *M*#} ⟷ *y* ∈ *f* ‘ *set-mset M*
⟨*proof*⟩

functor *image-mset*: *image-mset*
⟨*proof*⟩

declare

image-mset.id [*simp*]
image-mset.identity [*simp*]

lemma *image-mset-id*[*simp*]: *image-mset id x* = *x*
⟨*proof*⟩

lemma *image-mset-cong*: (⋀*x*. *x* ∈# *M* ⟹ *f x* = *g x*) ⟹ {#*f x*. *x* ∈# *M*#}
= {#*g x*. *x* ∈# *M*#}
⟨*proof*⟩

lemma *image-mset-cong-pair*:
(∀ *x y*. (*x*, *y*) ∈# *M* ⟶ *f x y* = *g x y*) ⟹ {#*f x y*. (*x*, *y*) ∈# *M*#} = {#*g x*
y. (*x*, *y*) ∈# *M*#}
⟨*proof*⟩

31.8 Further conversions

primrec *mset* :: *'a list* ⇒ *'a multiset* **where**
mset [] = {#} |
mset (*a* # *x*) = *add-mset a* (*mset x*)

lemma *in-multiset-in-set*:
x ∈# *mset xs* ⟷ *x* ∈ *set xs*
⟨*proof*⟩

lemma *count-mset*:
count (*mset xs*) *x* = *length* (*filter* (λ*y*. *x* = *y*) *xs*)
⟨*proof*⟩

lemma *mset-zero-iff*[*simp*]: (*mset x* = {#}) = (*x* = [])
⟨*proof*⟩

lemma *mset-zero-iff-right*[*simp*]: ({#} = *mset x*) = (*x* = [])
⟨*proof*⟩

lemma *mset-single-iff*[*iff*]: *mset xs* = {#*x*#} ⟷ *xs* = [*x*]
⟨*proof*⟩

lemma *mset-single-iff-right*[*iff*]: {#*x*#} = *mset xs* ⟷ *xs* = [*x*]
⟨*proof*⟩

lemma *set-mset-mset*[*simp*]: *set-mset* (*mset xs*) = *set xs*
⟨*proof*⟩

lemma *set-mset-comp-mset* [*simp*]: *set-mset* ∘ *mset* = *set*
⟨*proof*⟩

lemma *size-mset* [*simp*]: *size* (*mset xs*) = *length xs*
⟨*proof*⟩

lemma *mset-append* [*simp*]: *mset* (*xs* @ *ys*) = *mset xs* + *mset ys*
⟨*proof*⟩

lemma *mset-filter*: *mset* (*filter P xs*) = {#*x* ∈# *mset xs*. *P x* #}
⟨*proof*⟩

lemma *mset-rev* [*simp*]:
mset (*rev xs*) = *mset xs*
⟨*proof*⟩

lemma *surj-mset*: *surj mset*
⟨*proof*⟩

lemma *distinct-count-atmost-1*:
distinct x = (∀ *a*. *count* (*mset x*) *a* = (*if a* ∈ *set x then 1 else 0*))
⟨*proof*⟩

lemma *mset-eq-setD*:
assumes *mset xs* = *mset ys*
shows *set xs* = *set ys*
⟨*proof*⟩

lemma *set-eq-iff-mset-eq-distinct*:
distinct x ⟹ *distinct y* ⟹
(*set x* = *set y*) = (*mset x* = *mset y*)
⟨*proof*⟩

lemma *set-eq-iff-mset-remdups-eq*:
(*set x* = *set y*) = (*mset* (*remdups x*) = *mset* (*remdups y*))
⟨*proof*⟩

lemma *mset-compl-union* [*simp*]: *mset* [*x*←*xs*. *P x*] + *mset* [*x*←*xs*. ¬*P x*] = *mset xs*
⟨*proof*⟩

lemma *nth-mem-mset*: *i* < *length ls* ⟹ (*ls* ! *i*) ∈# *mset ls*
⟨*proof*⟩

lemma *mset-remove1*[*simp*]: *mset* (*remove1 a xs*) = *mset xs* − {#*a*#}
⟨*proof*⟩

lemma *mset-eq-length*:
assumes *mset xs* = *mset ys*
shows *length xs* = *length ys*
⟨*proof*⟩

lemma *mset-eq-length-filter*:
assumes *mset xs* = *mset ys*

shows *length* (*filter* ($\lambda x.\ z = x$) *xs*) = *length* (*filter* ($\lambda y.\ z = y$) *ys*)
⟨*proof*⟩

lemma *fold-multiset-equiv*:
assumes *f*: $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow f\ x \circ f\ y = f\ y \circ f\ x$
and *equiv*: *mset xs* = *mset ys*
shows *List.fold f xs* = *List.fold f ys*
⟨*proof*⟩

lemma *mset-insort* [*simp*]: *mset* (*insort x xs*) = *add-mset x* (*mset xs*)
⟨*proof*⟩

lemma *mset-map*[*simp*]: *mset* (*map f xs*) = *image-mset f* (*mset xs*)
⟨*proof*⟩

global-interpretation *mset-set*: *folding add-mset* {#}
defines *mset-set* = *folding.F add-mset* {#}
⟨*proof*⟩

lemma *count-mset-set* [*simp*]:
finite A $\Longrightarrow$ $x \in A \Longrightarrow$ *count* (*mset-set A*) *x* = *1* (**is** *PROP ?P*)
$\neg$ *finite A* $\Longrightarrow$ *count* (*mset-set A*) *x* = *0* (**is** *PROP ?Q*)
$x \notin A \Longrightarrow$ *count* (*mset-set A*) *x* = *0* (**is** *PROP ?R*)
⟨*proof*⟩

lemma *elem-mset-set*[*simp*, *intro*]: *finite A* $\Longrightarrow$ $x \in\#$ *mset-set A* $\longleftrightarrow$ $x \in A$
⟨*proof*⟩

lemma *mset-set-Union*:
finite A $\Longrightarrow$ *finite B* $\Longrightarrow$ $A \cap B = \{\} \Longrightarrow$ *mset-set* ($A \cup B$) = *mset-set A* + *mset-set B*
⟨*proof*⟩

lemma *filter-mset-mset-set* [*simp*]:
finite A $\Longrightarrow$ *filter-mset P* (*mset-set A*) = *mset-set* $\{x \in A.\ P\ x\}$
⟨*proof*⟩

lemma *mset-set-Diff*:
assumes *finite A* $B \subseteq A$
shows *mset-set* ($A - B$) = *mset-set A* − *mset-set B*
⟨*proof*⟩

lemma *mset-set-set*: *distinct xs* $\Longrightarrow$ *mset-set* (*set xs*) = *mset xs*
⟨*proof*⟩

context *linorder*
begin

definition *sorted-list-of-multiset* :: *'a multiset* ⇒ *'a list*

where
sorted-list-of-multiset M = *fold-mset insort* [] *M*

lemma *sorted-list-of-multiset-empty* [*simp*]:
sorted-list-of-multiset {#} = []
⟨*proof*⟩

lemma *sorted-list-of-multiset-singleton* [*simp*]:
sorted-list-of-multiset {#*x*#} = [*x*]
⟨*proof*⟩

lemma *sorted-list-of-multiset-insert* [*simp*]:
sorted-list-of-multiset (*add-mset x M*) = *List.insort x* (*sorted-list-of-multiset M*)
⟨*proof*⟩

end

lemma *mset-sorted-list-of-multiset* [*simp*]:
mset (*sorted-list-of-multiset M*) = *M*
⟨*proof*⟩

lemma *sorted-list-of-multiset-mset* [*simp*]:
sorted-list-of-multiset (*mset xs*) = *sort xs*
⟨*proof*⟩

lemma *finite-set-mset-mset-set*[*simp*]:
finite A ⟹ *set-mset* (*mset-set A*) = *A*
⟨*proof*⟩

lemma *mset-set-empty-iff*: *mset-set A* = {#} ⟷ *A* = {} ∨ *infinite A*
⟨*proof*⟩

lemma *infinite-set-mset-mset-set*:
¬ *finite A* ⟹ *set-mset* (*mset-set A*) = {}
⟨*proof*⟩

lemma *set-sorted-list-of-multiset* [*simp*]:
set (*sorted-list-of-multiset M*) = *set-mset M*
⟨*proof*⟩

lemma *sorted-list-of-mset-set* [*simp*]:
sorted-list-of-multiset (*mset-set A*) = *sorted-list-of-set A*
⟨*proof*⟩

lemma *mset-upt* [*simp*]: *mset* [*m*..<*n*] = *mset-set* {*m*..<*n*}
⟨*proof*⟩

lemma *image-mset-map-of*:
distinct (*map fst xs*) ⟹ {#*the* (*map-of xs i*). *i* ∈# *mset* (*map fst xs*)#} = *mset*

(*map snd xs*)
⟨*proof*⟩

lemma *image-mset-mset-set*:
assumes *inj-on f A*
shows *image-mset f* (*mset-set A*) = *mset-set* (*f* ‘ *A*)
⟨*proof*⟩

31.9 More properties of the replicate and repeat operations

lemma *in-replicate-mset*[*simp*]: *x* ∈# *replicate-mset n y* ⟷ *n* > *0* ∧ *x* = *y*
⟨*proof*⟩

lemma *set-mset-replicate-mset-subset*[*simp*]: *set-mset* (*replicate-mset n x*) = (*if n* = *0 then* {} *else* {*x*})
⟨*proof*⟩

lemma *size-replicate-mset*[*simp*]: *size* (*replicate-mset n M*) = *n*
⟨*proof*⟩

lemma *count-le-replicate-mset-subset-eq*: *n* ≤ *count M x* ⟷ *replicate-mset n x* ⊆# *M*
⟨*proof*⟩

lemma *filter-eq-replicate-mset*: {#*y* ∈# *D*. *y* = *x*#} = *replicate-mset* (*count D x*) *x*
⟨*proof*⟩

lemma *replicate-count-mset-eq-filter-eq*:
replicate (*count* (*mset xs*) *k*) *k* = *filter* (*HOL.eq k*) *xs*
⟨*proof*⟩

lemma *replicate-mset-eq-empty-iff* [*simp*]:
replicate-mset n a = {#} ⟷ *n* = *0*
⟨*proof*⟩

lemma *replicate-mset-eq-iff*:
replicate-mset m a = *replicate-mset n b* ⟷
m = *0* ∧ *n* = *0* ∨ *m* = *n* ∧ *a* = *b*
⟨*proof*⟩

lemma *repeat-mset-cancel1*: *repeat-mset a A* = *repeat-mset a B* ⟷ *A* = *B* ∨ *a* = *0*
⟨*proof*⟩

lemma *repeat-mset-cancel2*: *repeat-mset a A* = *repeat-mset b A* ⟷ *a* = *b* ∨ *A* = {#}
⟨*proof*⟩

lemma *repeat-mset-eq-empty-iff*: *repeat-mset n A* = {#} ⟷ *n* = *0* ∨ *A* = {#}
⟨*proof*⟩

lemma *image-replicate-mset* [*simp*]:
image-mset f (*replicate-mset n a*) = *replicate-mset n* (*f a*)
⟨*proof*⟩

31.10 Big operators

locale *comm-monoid-mset* = *comm-monoid*
begin

interpretation *comp-fun-commute f*
⟨*proof*⟩

interpretation *comp?*: *comp-fun-commute f* ∘ *g*
⟨*proof*⟩

context
begin

definition *F* :: *'a multiset* ⇒ *'a*
where *eq-fold*: *F M* = *fold-mset f* **1** *M*

lemma *empty* [*simp*]: *F* {#} = **1**
⟨*proof*⟩

lemma *singleton* [*simp*]: *F* {#*x*#} = *x*
⟨*proof*⟩

lemma *union* [*simp*]: *F* (*M* + *N*) = *F M* ∗ *F N*
⟨*proof*⟩

lemma *add-mset* [*simp*]: *F* (*add-mset x N*) = *x* ∗ *F N*
⟨*proof*⟩

lemma *insert* [*simp*]:
shows *F* (*image-mset g* (*add-mset x A*)) = *g x* ∗ *F* (*image-mset g A*)
⟨*proof*⟩

lemma *remove*:
assumes *x* ∈# *A*
shows *F A* = *x* ∗ *F* (*A* − {#*x*#})
⟨*proof*⟩

lemma *neutral*:
∀ *x*∈#*A*. *x* = **1** ⟹ *F A* = **1**
⟨*proof*⟩

lemma *neutral-const* [*simp*]:
F (*image-mset* (λ-. **1**) *A*) = **1**
⟨*proof*⟩ **lemma** *F-image-mset-product*:
F {#*g x j* ∗ *F* {#*g i j*. *i* ∈# *A*#}. *j* ∈# *B*#} =
F (*image-mset* (*g x*) *B*) ∗ *F* {#*F* {#*g i j*. *i* ∈# *A*#}. *j* ∈# *B*#}
⟨*proof*⟩

lemma *commute*:
F (*image-mset* (λ*i*. *F* (*image-mset* (*g i*) *B*)) *A*) =
F (*image-mset* (λ*j*. *F* (*image-mset* (λ*i*. *g i j*) *A*)) *B*)
⟨*proof*⟩

lemma *distrib*: *F* (*image-mset* (λ*x*. *g x* ∗ *h x*) *A*) = *F* (*image-mset g A*) ∗ *F* (*image-mset h A*)
⟨*proof*⟩

lemma *union-disjoint*:
A ∩# *B* = {#} ⟹ *F* (*image-mset g* (*A* ∪# *B*)) = *F* (*image-mset g A*) ∗ *F* (*image-mset g B*)
⟨*proof*⟩

end
end

lemma *comp-fun-commute-plus-mset*[*simp*]: *comp-fun-commute* (*op* + :: *'a multi-set* ⇒ - ⇒ -)
⟨*proof*⟩

declare *comp-fun-commute.fold-mset-add-mset*[*OF comp-fun-commute-plus-mset*, *simp*]

lemma *in-mset-fold-plus-iff*[*iff*]: *x* ∈# *fold-mset* (*op* +) *M NN* ⟷ *x* ∈# *M* ∨ (∃ *N*. *N* ∈# *NN* ∧ *x* ∈# *N*)
⟨*proof*⟩

context *comm-monoid-add*
begin

sublocale *sum-mset*: *comm-monoid-mset plus 0*
defines *sum-mset* = *sum-mset.F* ⟨*proof*⟩

lemma (**in** *semiring-1*) *sum-mset-replicate-mset* [*simp*]:
sum-mset (*replicate-mset n a*) = *of-nat n* ∗ *a*
⟨*proof*⟩

lemma *sum-unfold-sum-mset*:
sum f A = *sum-mset* (*image-mset f* (*mset-set A*))
⟨*proof*⟩

lemma *sum-mset-delta*: *sum-mset* (*image-mset* (λ*x*. *if x* = *y then c else 0*) *A*) = *c* ∗ *count A y*
⟨*proof*⟩

lemma *sum-mset-delta'*: *sum-mset* (*image-mset* (λ*x*. *if y* = *x then c else 0*) *A*) = *c* ∗ *count A y*
⟨*proof*⟩

end

lemma *of-nat-sum-mset* [*simp*]:
of-nat (*sum-mset M*) = *sum-mset* (*image-mset of-nat M*)
⟨*proof*⟩

lemma *sum-mset-0-iff* [*simp*]:
sum-mset M = (*0*::*'a*::*canonically-ordered-monoid-add*)
⟷ (∀ *x* ∈ *set-mset M*. *x* = *0*)
⟨*proof*⟩

lemma *sum-mset-diff*:
fixes *M N* :: (*'a* :: *ordered-cancel-comm-monoid-diff*) *multiset*
shows *N* ⊆# *M* ⟹ *sum-mset* (*M* − *N*) = *sum-mset M* − *sum-mset N*
⟨*proof*⟩

lemma *size-eq-sum-mset*: *size M* = *sum-mset* (*image-mset* (λ-. *1*) *M*)
⟨*proof*⟩

lemma *size-mset-set* [*simp*]: *size* (*mset-set A*) = *card A*
⟨*proof*⟩

lemma *sum-mset-sum-list*: *sum-mset* (*mset xs*) = *sum-list xs*
⟨*proof*⟩

syntax (*ASCII*)
-sum-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* (*(3SUM -:#-. -) [0, 51, 10] 10*)
syntax
-sum-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* (*(3*∑ *-∈#-. -) [0, 51, 10] 10*)
translations
∑ *i* ∈# *A*. *b* ⇌ *CONST sum-mset* (*CONST image-mset* (λ*i*. *b*) *A*)

lemma *sum-mset-distrib-left*:
fixes *f* :: *'a* ⇒ *'b*::*semiring-0*
shows *c* ∗ (∑ *x* ∈# *M*. *f x*) = (∑ *x* ∈# *M*. *c* ∗ *f*(*x*))
⟨*proof*⟩

lemma *sum-mset-distrib-right*:

fixes $f :: 'a \Rightarrow 'b$::*semiring-0*
shows $(\sum b \in\# B.\ f\ b) * a = (\sum b \in\# B.\ f\ b * a)$
⟨*proof*⟩

lemma *sum-mset-constant* [*simp*]:
fixes $y :: 'b$::*semiring-1*
shows $‹(\sum x \in\# A.\ y) = of\text{-}nat\ (size\ A) * y›$
⟨*proof*⟩

lemma (**in** *ordered-comm-monoid-add*) *sum-mset-mono*:
assumes $\bigwedge i.\ i \in\# K \Longrightarrow f\ i \leq g\ i$
shows $sum\text{-}mset\ (image\text{-}mset\ f\ K) \leq sum\text{-}mset\ (image\text{-}mset\ g\ K)$
⟨*proof*⟩

lemma *sum-mset-product*:
fixes $f :: 'a$::{*comm-monoid-add*,*times*} $\Rightarrow 'b$::*semiring-0*
shows $(\sum i \in\# A.\ f\ i) * (\sum i \in\# B.\ g\ i) = (\sum i \in\# A.\ \sum j \in\# B.\ f\ i * g\ j)$
⟨*proof*⟩

abbreviation *Union-mset* :: *'a multiset multiset* $\Rightarrow$ *'a multiset* $(\bigcup\#\text{-}\ [900]\ 900)$
where $\bigcup\# MM \equiv sum\text{-}mset\ MM$ — FIXME ambiguous notation – could likewise refer to $\bigsqcup\#$

lemma *set-mset-Union-mset*[*simp*]: $set\text{-}mset\ (\bigcup\# MM) = (\bigcup M \in set\text{-}mset\ MM.\ set\text{-}mset\ M)$
⟨*proof*⟩

lemma *in-Union-mset-iff*[*iff*]: $x \in\# \bigcup\# MM \longleftrightarrow (\exists M.\ M \in\# MM \wedge x \in\# M)$
⟨*proof*⟩

lemma *count-sum*:
$count\ (sum\ f\ A)\ x = sum\ (\lambda a.\ count\ (f\ a)\ x)\ A$
⟨*proof*⟩

lemma *sum-eq-empty-iff*:
assumes *finite A*
shows $sum\ f\ A = \{\#\} \longleftrightarrow (\forall a \in A.\ f\ a = \{\#\})$
⟨*proof*⟩

lemma *Union-mset-empty-conv*[*simp*]: $\bigcup\# M = \{\#\} \longleftrightarrow (\forall i \in\# M.\ i = \{\#\})$
⟨*proof*⟩

context *comm-monoid-mult*
begin

sublocale *prod-mset*: *comm-monoid-mset times 1*
defines *prod-mset* = *prod-mset.F* ⟨*proof*⟩

lemma *prod-mset-empty*:
prod-mset {#} = *1*
⟨*proof*⟩

lemma *prod-mset-singleton*:
prod-mset {#*x*#} = *x*
⟨*proof*⟩

lemma *prod-mset-Un*:
prod-mset (*A* + *B*) = *prod-mset A* ∗ *prod-mset B*
⟨*proof*⟩

lemma *prod-mset-replicate-mset* [*simp*]:
prod-mset (*replicate-mset n a*) = *a* ^ *n*
⟨*proof*⟩

lemma *prod-unfold-prod-mset*:
prod f A = *prod-mset* (*image-mset f* (*mset-set A*))
⟨*proof*⟩

lemma *prod-mset-multiplicity*:
prod-mset M = *prod* (λ*x*. *x* ^ *count M x*) (*set-mset M*)
⟨*proof*⟩

lemma *prod-mset-delta*: *prod-mset* (*image-mset* (λ*x*. *if x* = *y then c else 1*) *A*) = *c* ^ *count A y*
⟨*proof*⟩

lemma *prod-mset-delta′*: *prod-mset* (*image-mset* (λ*x*. *if y* = *x then c else 1*) *A*) = *c* ^ *count A y*
⟨*proof*⟩

end

syntax (*ASCII*)
-prod-mset-image :: *pttrn* ⇒ *′b set* ⇒ *′a* ⇒ *′a*::*comm-monoid-mult* ((*3PROD -*:#*-. -*) [*0*, *51*, *10*] *10*)
syntax
-prod-mset-image :: *pttrn* ⇒ *′b set* ⇒ *′a* ⇒ *′a*::*comm-monoid-mult* ((*3*∏ *-*∈#*-. -*) [*0*, *51*, *10*] *10*)
translations
∏ *i* ∈# *A*. *b* ⇌ *CONST prod-mset* (*CONST image-mset* (λ*i*. *b*) *A*)

lemma (**in** *comm-monoid-mult*) *prod-mset-subset-imp-dvd*:
assumes *A* ⊆# *B*
shows *prod-mset A dvd prod-mset B*
⟨*proof*⟩

lemma (**in** *comm-monoid-mult*) *dvd-prod-mset*:

assumes $x \in\# A$
shows x *dvd* *prod-mset* A
⟨*proof*⟩

lemma (**in** *semidom*) *prod-mset-zero-iff* [*iff*]:
$prod\text{-}mset\ A = 0 \longleftrightarrow 0 \in\# A$
⟨*proof*⟩

lemma (**in** *semidom-divide*) *prod-mset-diff*:
assumes $B \subseteq\# A$ **and** $0 \notin\# B$
shows $prod\text{-}mset\ (A - B) = prod\text{-}mset\ A\ div\ prod\text{-}mset\ B$
⟨*proof*⟩

lemma (**in** *semidom-divide*) *prod-mset-minus*:
assumes $a \in\# A$ **and** $a \neq 0$
shows $prod\text{-}mset\ (A - \{\#a\#\}) = prod\text{-}mset\ A\ div\ a$
⟨*proof*⟩

lemma (**in** *algebraic-semidom*) *is-unit-prod-mset-iff*:
$is\text{-}unit\ (prod\text{-}mset\ A) \longleftrightarrow (\forall x \in\# A.\ is\text{-}unit\ x)$
⟨*proof*⟩

lemma (**in** *normalization-semidom*) *normalize-prod-mset*:
$normalize\ (prod\text{-}mset\ A) = prod\text{-}mset\ (image\text{-}mset\ normalize\ A)$
⟨*proof*⟩

lemma (**in** *normalization-semidom*) *normalized-prod-msetI*:
assumes $\bigwedge a.\ a \in\# A \Longrightarrow normalize\ a = a$
shows $normalize\ (prod\text{-}mset\ A) = prod\text{-}mset\ A$
⟨*proof*⟩

lemma *prod-mset-prod-list*: $prod\text{-}mset\ (mset\ xs) = prod\text{-}list\ xs$
⟨*proof*⟩

31.11 Alternative representations

31.11.1 Lists

context *linorder*
begin

lemma *mset-insort* [*simp*]:
$mset\ (insort\text{-}key\ k\ x\ xs) = add\text{-}mset\ x\ (mset\ xs)$
⟨*proof*⟩

lemma *mset-sort* [*simp*]:
$mset\ (sort\text{-}key\ k\ xs) = mset\ xs$
⟨*proof*⟩

This lemma shows which properties suffice to show that a function f with f

$xs = ys$ behaves like sort.

lemma *properties-for-sort-key*:
  assumes *mset ys = mset xs*
    and $\bigwedge k.\ k \in set\ ys \Longrightarrow filter\ (\lambda x.\ f\ k = f\ x)\ ys = filter\ (\lambda x.\ f\ k = f\ x)\ xs$
    and *sorted (map f ys)*
  shows *sort-key f xs = ys*
  ⟨*proof*⟩

lemma *properties-for-sort*:
  assumes *multiset*: *mset ys = mset xs*
    and *sorted ys*
  shows *sort xs = ys*
⟨*proof*⟩

lemma *sort-key-inj-key-eq*:
  assumes *mset-equal*: *mset xs = mset ys*
    and *inj-on f (set xs)*
    and *sorted (map f ys)*
  shows *sort-key f xs = ys*
⟨*proof*⟩

lemma *sort-key-eq-sort-key*:
  assumes *mset xs = mset ys*
    and *inj-on f (set xs)*
  shows *sort-key f xs = sort-key f ys*
  ⟨*proof*⟩

lemma *sort-key-by-quicksort*:
  sort-key f xs = sort-key f [x←xs. f x < f (xs ! (length xs div 2))]
    @ [x←xs. f x = f (xs ! (length xs div 2))]
    @ sort-key f [x←xs. f x > f (xs ! (length xs div 2))] (**is** *sort-key f ?lhs = ?rhs*)
⟨*proof*⟩

lemma *sort-by-quicksort*:
  sort xs = sort [x←xs. x < xs ! (length xs div 2)]
    @ [x←xs. x = xs ! (length xs div 2)]
    @ sort [x←xs. x > xs ! (length xs div 2)] (**is** *sort ?lhs = ?rhs*)
  ⟨*proof*⟩

A stable parametrized quicksort

definition *part* :: $('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'b\ list \times 'b\ list \times 'b\ list$ **where**
  part f pivot xs = ([x ← xs. f x < pivot], [x ← xs. f x = pivot], [x ← xs. pivot < f x])

lemma *part-code* [*code*]:
  part f pivot [] = ([], [], [])
  part f pivot (x # xs) = (let (lts, eqs, gts) = part f pivot xs; x' = f x in
    if x' < pivot then (x # lts, eqs, gts)
    else if x' > pivot then (lts, eqs, x # gts)

else (*lts*, *x* # *eqs*, *gts*))
⟨*proof*⟩

lemma *sort-key-by-quicksort-code* [*code*]:
sort-key f xs =
(*case xs of*
[] ⇒ []
| [*x*] ⇒ *xs*
| [*x*, *y*] ⇒ (*if f x* ≤ *f y then xs else* [*y*, *x*])
| - ⇒
let (*lts*, *eqs*, *gts*) = *part f* (*f* (*xs* ! (*length xs div 2*))) *xs*
in sort-key f lts @ *eqs* @ *sort-key f gts*)
⟨*proof*⟩

end

hide-const (**open**) *part*

lemma *mset-remdups-subset-eq*: *mset* (*remdups xs*) ⊆# *mset xs*
⟨*proof*⟩

lemma *mset-update*:
i < *length ls* ⟹ *mset* (*ls*[*i* := *v*]) = *add-mset v* (*mset ls* − {#*ls* ! *i*#})
⟨*proof*⟩

lemma *mset-swap*:
i < *length ls* ⟹ *j* < *length ls* ⟹
mset (*ls*[*j* := *ls* ! *i*, *i* := *ls* ! *j*]) = *mset ls*
⟨*proof*⟩

31.12 The multiset order

31.12.1 Well-foundedness

definition *mult1* :: ('*a* × '*a*) *set* ⇒ ('*a multiset* × '*a multiset*) *set* **where**
mult1 r = {(*N*, *M*). ∃ *a M0 K*. *M* = *add-mset a M0* ∧ *N* = *M0* + *K* ∧
(∀ *b*. *b* ∈# *K* ⟶ (*b*, *a*) ∈ *r*)}

definition *mult* :: ('*a* × '*a*) *set* ⇒ ('*a multiset* × '*a multiset*) *set* **where**
mult r = (*mult1 r*)$^+$

lemma *mult1I*:
assumes *M* = *add-mset a M0* **and** *N* = *M0* + *K* **and** ⋀*b*. *b* ∈# *K* ⟹ (*b*, *a*) ∈ *r*
shows (*N*, *M*) ∈ *mult1 r*
⟨*proof*⟩

lemma *mult1E*:
assumes (*N*, *M*) ∈ *mult1 r*
obtains *a M0 K* **where** *M* = *add-mset a M0 N* = *M0* + *K* ⋀*b*. *b* ∈# *K* ⟹

$(b, a) \in r$
⟨*proof*⟩

lemma *mono-mult1*:
assumes $r \subseteq r'$ **shows** *mult1* $r \subseteq$ *mult1* r'
⟨*proof*⟩

lemma *mono-mult*:
assumes $r \subseteq r'$ **shows** *mult* $r \subseteq$ *mult* r'
⟨*proof*⟩

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin$ *mult1* r
⟨*proof*⟩

lemma *less-add*:
assumes *mult1*: $(N,$ *add-mset a M0*$) \in$ *mult1* r
shows
$(\exists M.\ (M,\ M0) \in$ *mult1* $r \wedge N =$ *add-mset a* $M) \vee$
$(\exists K.\ (\forall b.\ b \in\# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$
⟨*proof*⟩

lemma *all-accessible*:
assumes *wf* r
shows $\forall M.\ M \in$ *Wellfounded.acc* (*mult1* r)
⟨*proof*⟩

theorem *wf-mult1*: *wf* $r \Longrightarrow$ *wf* (*mult1* r)
⟨*proof*⟩

theorem *wf-mult*: *wf* $r \Longrightarrow$ *wf* (*mult* r)
⟨*proof*⟩

31.12.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:
assumes
trans: *trans* r **and**
MN: $(M, N) \in$ *mult* r
shows $\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k \in$ *set-mset* $K.\ \exists j \in$ *set-mset* $J.\ (k, j) \in r)$
⟨*proof*⟩

lemma *one-step-implies-mult*:
assumes
$J \neq \{\#\}$ **and**
$\forall k \in$ *set-mset* $K.\ \exists j \in$ *set-mset* $J.\ (k, j) \in r$
shows $(I + K, I + J) \in$ *mult* r
⟨*proof*⟩

31.13 The multiset extension is cancellative for multiset union

lemma *mult-cancel*:
assumes *trans s* **and** *irrefl s*
shows $(X + Z,\ Y + Z) \in \mathit{mult}\ s \longleftrightarrow (X,\ Y) \in \mathit{mult}\ s$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemma *mult-cancel-max*:
assumes *trans s* **and** *irrefl s*
shows $(X,\ Y) \in \mathit{mult}\ s \longleftrightarrow (X - X \cap\# Y,\ Y - X \cap\# Y) \in \mathit{mult}\ s$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

31.14 Quasi-executable version of the multiset extension

Predicate variants of *mult* and the reflexive closure of *mult*, which are executable whenever the given predicate P is. Together with the standard code equations for *op* $\cap\#$ and *op* $-$ this should yield quadratic (with respect to calls to P) implementations of *multp* and *multeqp*.

definition *multp* :: $('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{multiset} \Rightarrow 'a\ \mathit{multiset} \Rightarrow \mathit{bool}$ **where**
multp P N M =
($\mathit{let}\ Z = M \cap\# N;\ X = M - Z\ \mathit{in}$
$X \neq \{\#\} \wedge (\mathit{let}\ Y = N - Z\ \mathit{in}\ (\forall y \in \mathit{set\text{-}mset}\ Y.\ \exists x \in \mathit{set\text{-}mset}\ X.\ P\ y\ x)))$

definition *multeqp* :: $('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{multiset} \Rightarrow 'a\ \mathit{multiset} \Rightarrow \mathit{bool}$ **where**
multeqp P N M =
($\mathit{let}\ Z = M \cap\# N;\ X = M - Z;\ Y = N - Z\ \mathit{in}$
$(\forall y \in \mathit{set\text{-}mset}\ Y.\ \exists x \in \mathit{set\text{-}mset}\ X.\ P\ y\ x)$)

lemma *multp-iff*:
assumes *irrefl R* **and** *trans R* **and** [*simp*]: $\bigwedge x\ y.\ P\ x\ y \longleftrightarrow (x,\ y) \in R$
shows *multp P N M* $\longleftrightarrow (N,\ M) \in \mathit{mult}\ R$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemma *multeqp-iff*:
assumes *irrefl R* **and** *trans R* **and** $\bigwedge x\ y.\ P\ x\ y \longleftrightarrow (x,\ y) \in R$
shows *multeqp P N M* $\longleftrightarrow (N,\ M) \in (\mathit{mult}\ R)^{=}$
⟨*proof*⟩

31.14.1 Partial-order properties

lemma (**in** *preorder*) *mult1-lessE*:
assumes $(N,\ M) \in \mathit{mult1}\ \{(a,\ b).\ a < b\}$
obtains *a M0 K* **where** $M = \mathit{add\text{-}mset}\ a\ M0\ N = M0 + K$
$a \notin\# K\ \bigwedge b.\ b \in\# K \Longrightarrow b < a$
⟨*proof*⟩

instantiation *multiset* :: (*preorder*) *order*
begin

definition *less-multiset* :: *'a multiset ⇒ 'a multiset ⇒ bool*
  where $M' < M \longleftrightarrow (M', M) \in mult\ \{(x', x).\ x' < x\}$

definition *less-eq-multiset* :: *'a multiset ⇒ 'a multiset ⇒ bool*
  where *less-eq-multiset* $M'\ M \longleftrightarrow M' < M \vee M' = M$

instance
⟨*proof*⟩
end — FIXME avoid junk stemming from type class interpretation

lemma *mset-le-irrefl* [*elim*!]:
  fixes M :: *'a*::*preorder multiset*
  shows $M < M \Longrightarrow R$
  ⟨*proof*⟩

31.14.2 Monotonicity of multiset union

lemma *mult1-union*: $(B, D) \in mult1\ r \Longrightarrow (C + B, C + D) \in mult1\ r$
  ⟨*proof*⟩

lemma *union-le-mono2*: $B < D \Longrightarrow C + B < C + (D::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *union-le-mono1*: $B < D \Longrightarrow B + C < D + (C::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *union-less-mono*:
  fixes $A\ B\ C\ D$ :: *'a*::*preorder multiset*
  shows $A < C \Longrightarrow B < D \Longrightarrow A + B < C + D$
  ⟨*proof*⟩

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-add*
begin
instance
  ⟨*proof*⟩
end

31.14.3 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in\# XS \Longrightarrow x \in\# \{\#\ y\ \#\} + XS$
  and *multi-member-this*: $x \in\# \{\#\ x\ \#\} + XS$
  and *multi-member-last*: $x \in\# \{\#\ x\ \#\}$
  ⟨*proof*⟩

definition *ms-strict* = *mult pair-less*
definition *ms-weak* = *ms-strict* ∪ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
⟨*proof*⟩

lemma *smsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-strict*
⟨*proof*⟩

lemma *wmsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ *A* = {#} ∧ *B* = {#}
⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-weak*
⟨*proof*⟩

inductive *pw-leq*
where
pw-leq-empty: *pw-leq* {#} {#}
| *pw-leq-step*: ⟦(*x*,*y*) ∈ *pair-leq*; *pw-leq X Y* ⟧ ⟹ *pw-leq* ({#*x*#} + *X*) ({#*y*#} + *Y*)

lemma *pw-leq-lstep*:
(*x*, *y*) ∈ *pair-leq* ⟹ *pw-leq* {#*x*#} {#*y*#}
⟨*proof*⟩

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows ∃ *A B Z*. *X* = *A* + *Z* ∧ *Y* = *B* + *Z* ∧ ((*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#}))
⟨*proof*⟩

lemma
assumes *pwleq*: *pw-leq Z Z′*
shows *ms-strictI*: (*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z′* + *B*) ∈ *ms-strict*
and *ms-weakI1*: (*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z′* + *B*) ∈ *ms-weak*
and *ms-weakI2*: (*Z* + {#}, *Z′* + {#}) ∈ *ms-weak*
⟨*proof*⟩

lemma *empty-neutral*: {#} + *x* = *x x* + {#} = *x*
and *nonempty-plus*: {# *x* #} + *rs* ≠ {#}
and *nonempty-single*: {# *x* #} ≠ {#}
⟨*proof*⟩

⟨*ML*⟩

31.15 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-eq-iff* [*symmetric*]

lemma *union-commute*: *M* + *N* = *N* + (*M*::′*a multiset*)
⟨*proof*⟩

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
⟨*proof*⟩

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
⟨*proof*⟩

lemmas *union-ac* = *union-assoc union-commute union-lcomm add-mset-commute*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ multiset)$
⟨*proof*⟩

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ multiset)$
⟨*proof*⟩

lemma *multi-union-self-other-eq*: $(A::'a\ multiset) + X = A + Y \Longrightarrow X = Y$
⟨*proof*⟩

lemma *mset-subset-trans*: $(M::'a\ multiset) \subset\# K \Longrightarrow K \subset\# N \Longrightarrow M \subset\# N$
⟨*proof*⟩

lemma *multiset-inter-commute*: $A \cap\# B = B \cap\# A$
⟨*proof*⟩

lemma *multiset-inter-assoc*: $A \cap\# (B \cap\# C) = A \cap\# B \cap\# C$
⟨*proof*⟩

lemma *multiset-inter-left-commute*: $A \cap\# (B \cap\# C) = B \cap\# (A \cap\# C)$
⟨*proof*⟩

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *mset-le-not-refl*: $\neg M < (M::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *mset-le-trans*: $K < M \Longrightarrow M < N \Longrightarrow K < (N::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *mset-le-not-sym*: $M < N \Longrightarrow \neg N < (M::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *mset-le-asym*: $M < N \Longrightarrow (\neg P \Longrightarrow N < (M::'a::preorder\ multiset)) \Longrightarrow P$
⟨*proof*⟩

⟨*ML*⟩

31.16 Naive implementation using lists

code-datatype *mset*

lemma [*code*]: {#} = *mset* []
⟨*proof*⟩

lemma [*code*]: *add-mset x* (*mset xs*) = *mset* (*x* # *xs*)
⟨*proof*⟩

lemma [*code*]: *Multiset.is-empty* (*mset xs*) ⟷ *List.null xs*
⟨*proof*⟩

lemma *union-code* [*code*]: *mset xs* + *mset ys* = *mset* (*xs* @ *ys*)
⟨*proof*⟩

lemma [*code*]: *image-mset f* (*mset xs*) = *mset* (*map f xs*)
⟨*proof*⟩

lemma [*code*]: *filter-mset f* (*mset xs*) = *mset* (*filter f xs*)
⟨*proof*⟩

lemma [*code*]: *mset xs* − *mset ys* = *mset* (*fold remove1 ys xs*)
⟨*proof*⟩

lemma [*code*]:
mset xs ∩# *mset ys* =
mset (*snd* (*fold* (λ*x* (*ys*, *zs*).
if x ∈ *set ys then* (*remove1 x ys*, *x* # *zs*) *else* (*ys*, *zs*)) *xs* (*ys*, []))))
⟨*proof*⟩

lemma [*code*]:
mset xs ∪# *mset ys* =
mset (*case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, []))))
⟨*proof*⟩

declare *in-multiset-in-set* [*code-unfold*]

lemma [*code*]: *count* (*mset xs*) *x* = *fold* (λ*y*. *if x* = *y then Suc else id*) *xs 0*
⟨*proof*⟩

declare *set-mset-mset* [*code*]

declare *sorted-list-of-multiset-mset* [*code*]

lemma [*code*]: — not very efficient, but representation-ignorant!
mset-set A = *mset* (*sorted-list-of-set A*)
⟨*proof*⟩

declare *size-mset* [*code*]

fun *subset-eq-mset-impl* :: *'a list* ⇒ *'a list* ⇒ *bool option* **where**
  subset-eq-mset-impl [] *ys* = *Some* (*ys* ≠ [])
| *subset-eq-mset-impl* (*Cons x xs*) *ys* = (*case List.extract* (*op* = *x*) *ys of*
    None ⇒ *None*
  | *Some* (*ys1*,-,*ys2*) ⇒ *subset-eq-mset-impl xs* (*ys1* @ *ys2*))

lemma *subset-eq-mset-impl*: (*subset-eq-mset-impl xs ys* = *None* ⟷ ¬ *mset xs* ⊆# *mset ys*) ∧
  (*subset-eq-mset-impl xs ys* = *Some True* ⟷ *mset xs* ⊂# *mset ys*) ∧
  (*subset-eq-mset-impl xs ys* = *Some False* ⟶ *mset xs* = *mset ys*)
⟨*proof*⟩

lemma [*code*]: *mset xs* ⊆# *mset ys* ⟷ *subset-eq-mset-impl xs ys* ≠ *None*
  ⟨*proof*⟩

lemma [*code*]: *mset xs* ⊂# *mset ys* ⟷ *subset-eq-mset-impl xs ys* = *Some True*
  ⟨*proof*⟩

instantiation *multiset* :: (*equal*) *equal*
begin

definition
  [*code del*]: *HOL.equal A* (*B* :: *'a multiset*) ⟷ *A* = *B*
lemma [*code*]: *HOL.equal* (*mset xs*) (*mset ys*) ⟷ *subset-eq-mset-impl xs ys* = *Some False*
  ⟨*proof*⟩

instance
  ⟨*proof*⟩

end

lemma [*code*]: *sum-mset* (*mset xs*) = *sum-list xs*
  ⟨*proof*⟩

lemma [*code*]: *prod-mset* (*mset xs*) = *fold times xs 1*
⟨*proof*⟩

Exercise for the casual reader: add implementations for *op* ≤ and *op* < (multiset order).

Quickcheck generators

definition (**in** *term-syntax*)
  msetify :: *'a*::*typerep list* × (*unit* ⇒ *Code-Evaluation.term*)
    ⇒ *'a multiset* × (*unit* ⇒ *Code-Evaluation.term*) **where**
  [*code-unfold*]: *msetify xs* = *Code-Evaluation.valtermify mset* {·} *xs*

notation *fcomp* (**infixl** ∘> *60*)

notation *scomp* (**infixl** $\circ\!\!\rightarrow$ *60*)

instantiation *multiset* :: (*random*) *random*
begin

definition
Quickcheck-Random.random i = *Quickcheck-Random.random i* $\circ\!\!\rightarrow$ (λ*xs*. *Pair* (*msetify xs*))

instance ⟨*proof*⟩

end

no-notation *fcomp* (**infixl** $\circ\!\!>$ *60*)
no-notation *scomp* (**infixl** $\circ\!\!\rightarrow$ *60*)

instantiation *multiset* :: (*full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive-multiset* :: ('*a multiset* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
full-exhaustive-multiset f i = *Quickcheck-Exhaustive.full-exhaustive* (λ*xs*. *f* (*msetify xs*)) *i*

instance ⟨*proof*⟩

end

hide-const (**open**) *msetify*

31.17 BNF setup

definition *rel-mset* **where**
rel-mset R X Y ⟷ (∃*xs ys*. *mset xs* = *X* ∧ *mset ys* = *Y* ∧ *list-all2 R xs ys*)

lemma *mset-zip-take-Cons-drop-twice*:
assumes *length xs* = *length ys j* ≤ *length xs*
shows *mset* (*zip* (*take j xs* @ *x* # *drop j xs*) (*take j ys* @ *y* # *drop j ys*)) =
add-mset (*x*,*y*) (*mset* (*zip xs ys*))
⟨*proof*⟩

lemma *ex-mset-zip-left*:
assumes *length xs* = *length ys mset xs'* = *mset xs*
shows ∃*ys'*. *length ys'* = *length xs'* ∧ *mset* (*zip xs' ys'*) = *mset* (*zip xs ys*)
⟨*proof*⟩

lemma *list-all2-reorder-left-invariance*:
assumes *rel*: *list-all2 R xs ys* **and** *ms-x*: *mset xs'* = *mset xs*

shows $\exists ys'.\ list\text{-}all2\ R\ xs'\ ys' \wedge mset\ ys' = mset\ ys$
⟨*proof*⟩

lemma *ex-mset*: $\exists xs.\ mset\ xs = X$
⟨*proof*⟩

inductive *pred-mset* :: $('a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow bool$
where
pred-mset P {#}
| ⟦*P a*; *pred-mset P M*⟧ ⟹ *pred-mset P* (*add-mset a M*)

bnf $'a\ multiset$
map: *image-mset*
sets: *set-mset*
bd: *natLeq*
wits: {#}
rel: *rel-mset*
pred: *pred-mset*
⟨*proof*⟩

inductive *rel-mset'*
where
Zero[*intro*]: *rel-mset' R* {#} {#}
| *Plus*[*intro*]: ⟦*R a b*; *rel-mset' R M N*⟧ ⟹ *rel-mset' R* (*add-mset a M*) (*add-mset b N*)

lemma *rel-mset-Zero*: *rel-mset R* {#} {#}
⟨*proof*⟩

declare *multiset.count*[*simp*]
declare *Abs-multiset-inverse*[*simp*]
declare *multiset.count-inverse*[*simp*]
declare *union-preserves-multiset*[*simp*]

lemma *rel-mset-Plus*:
assumes *ab*: *R a b*
and *MN*: *rel-mset R M N*
shows *rel-mset R* (*add-mset a M*) (*add-mset b N*)
⟨*proof*⟩

lemma *rel-mset'-imp-rel-mset*: *rel-mset' R M N* ⟹ *rel-mset R M N*
⟨*proof*⟩

lemma *rel-mset-size*: *rel-mset R M N* ⟹ *size M* = *size N*
⟨*proof*⟩

lemma *multiset-induct2*[*case-names empty addL addR*]:
assumes *empty*: *P* {#} {#}
and *addL*: $\bigwedge a\ M\ N.\ P\ M\ N \Longrightarrow P\ (add\text{-}mset\ a\ M)\ N$

and *addR*: $\bigwedge a\ M\ N.\ P\ M\ N \Longrightarrow P\ M\ (\mathit{add\text{-}mset}\ a\ N)$
shows $P\ M\ N$
⟨*proof*⟩

lemma *multiset-induct2-size*[*consumes 1*, *case-names empty add*]:
assumes *c*: $\mathit{size}\ M = \mathit{size}\ N$
and *empty*: $P\ \{\#\}\ \{\#\}$
and *add*: $\bigwedge a\ b\ M\ N\ a\ b.\ P\ M\ N \Longrightarrow P\ (\mathit{add\text{-}mset}\ a\ M)\ (\mathit{add\text{-}mset}\ b\ N)$
shows $P\ M\ N$
⟨*proof*⟩

lemma *msed-map-invL*:
assumes $\mathit{image\text{-}mset}\ f\ (\mathit{add\text{-}mset}\ a\ M) = N$
shows $\exists N1.\ N = \mathit{add\text{-}mset}\ (f\ a)\ N1 \wedge \mathit{image\text{-}mset}\ f\ M = N1$
⟨*proof*⟩

lemma *msed-map-invR*:
assumes $\mathit{image\text{-}mset}\ f\ M = \mathit{add\text{-}mset}\ b\ N$
shows $\exists M1\ a.\ M = \mathit{add\text{-}mset}\ a\ M1 \wedge f\ a = b \wedge \mathit{image\text{-}mset}\ f\ M1 = N$
⟨*proof*⟩

lemma *msed-rel-invL*:
assumes $\mathit{rel\text{-}mset}\ R\ (\mathit{add\text{-}mset}\ a\ M)\ N$
shows $\exists N1\ b.\ N = \mathit{add\text{-}mset}\ b\ N1 \wedge R\ a\ b \wedge \mathit{rel\text{-}mset}\ R\ M\ N1$
⟨*proof*⟩

lemma *msed-rel-invR*:
assumes $\mathit{rel\text{-}mset}\ R\ M\ (\mathit{add\text{-}mset}\ b\ N)$
shows $\exists M1\ a.\ M = \mathit{add\text{-}mset}\ a\ M1 \wedge R\ a\ b \wedge \mathit{rel\text{-}mset}\ R\ M1\ N$
⟨*proof*⟩

lemma *rel-mset-imp-rel-mset′*:
assumes $\mathit{rel\text{-}mset}\ R\ M\ N$
shows $\mathit{rel\text{-}mset'}\ R\ M\ N$
⟨*proof*⟩

lemma *rel-mset-rel-mset′*: $\mathit{rel\text{-}mset}\ R\ M\ N = \mathit{rel\text{-}mset'}\ R\ M\ N$
⟨*proof*⟩

The main end product for *rel-mset*: inductive characterization:

lemmas *rel-mset-induct*[*case-names empty add*, *induct pred*: *rel-mset*] =
rel-mset′.induct[*unfolded rel-mset-rel-mset′*[*symmetric*]]

31.18 Size setup

lemma *multiset-size-o-map*:
$\mathit{size\text{-}multiset}\ g \circ \mathit{image\text{-}mset}\ f = \mathit{size\text{-}multiset}\ (g \circ f)$
⟨*proof*⟩

⟨*ML*⟩

hide-const (**open**) *wcount*

end

32 Bubblesort

theory *Bubblesort*
imports ~~/*src*/*HOL*/*Library*/*Multiset*
begin

This is *a* version of bubblesort.

context *linorder*
begin

fun *bubble-min* **where**
bubble-min [] = [] |
bubble-min [*x*] = [*x*] |
bubble-min (*x*#*xs*) =
  (*case bubble-min xs of y*#*ys* ⇒ *if x*>*y then y*#*x*#*ys else x*#*y*#*ys*)

lemma *size-bubble-min*: *size*(*bubble-min xs*) = *size xs*
⟨*proof*⟩

lemma *bubble-min-eq-Nil-iff* [*simp*]: *bubble-min xs* = [] ⟷ *xs* = []
⟨*proof*⟩

lemma *bubble-minD-size*: *bubble-min* (*xs*) = *ys* ⟹ *size xs* = *size ys*
⟨*proof*⟩

function (*sequential*) *bubblesort* **where**
bubblesort [] = [] |
bubblesort [*x*] = [*x*] |
bubblesort xs = (*case bubble-min xs of y*#*ys* ⇒ *y* # *bubblesort ys*)
⟨*proof*⟩

termination
⟨*proof*⟩

lemma *mset-bubble-min*: *mset* (*bubble-min xs*) = *mset xs*
⟨*proof*⟩

lemma *bubble-minD-mset*:
  bubble-min (*xs*) = *ys* ⟹ *mset xs* = *mset ys*
⟨*proof*⟩

lemma *mset-bubblesort*:
  mset (*bubblesort xs*) = *mset xs*

⟨*proof*⟩

lemma *set-bubblesort*: *set* (*bubblesort xs*) = *set xs*
⟨*proof*⟩

lemma *bubble-min-min*: *bubble-min xs* = *y*#*ys* ⟹ *z* ∈ *set ys* ⟹ *y* ≤ *z*
⟨*proof*⟩

lemma *sorted-bubblesort*: *sorted*(*bubblesort xs*)
⟨*proof*⟩

end

end

33 Merge Sort

theory *MergeSort*
imports ~~/*src*/*HOL*/*Library*/*Multiset*
begin

context *linorder*
begin

fun *merge* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
  merge (*x*#*xs*) (*y*#*ys*) =
      (*if x* ≤ *y then x* # *merge xs* (*y*#*ys*) *else y* # *merge* (*x*#*xs*) *ys*)
| *merge xs* [] = *xs*
| *merge* [] *ys* = *ys*

lemma *mset-merge* [*simp*]:
  mset (*merge xs ys*) = *mset xs* + *mset ys*
  ⟨*proof*⟩

lemma *set-merge* [*simp*]:
  set (*merge xs ys*) = *set xs* ∪ *set ys*
  ⟨*proof*⟩

lemma *sorted-merge* [*simp*]:
  sorted (*merge xs ys*) ⟷ *sorted xs* ∧ *sorted ys*
  ⟨*proof*⟩

fun *msort* :: *'a list* ⇒ *'a list*
where
  msort [] = []
| *msort* [*x*] = [*x*]
| *msort xs* = *merge* (*msort* (*take* (*size xs div 2*) *xs*))
                (*msort* (*drop* (*size xs div 2*) *xs*))

lemma *sorted-msort*:
sorted (*msort xs*)
⟨*proof*⟩

lemma *mset-msort*:
mset (*msort xs*) = *mset xs*
⟨*proof*⟩

theorem *msort-sort*:
sort = *msort*
⟨*proof*⟩

end

end

34 A lemma for Lagrange's theorem

theory *Lagrange* **imports** *Main* **begin**

This theory only contains a single theorem, which is a lemma in Lagrange's proof that every natural number is the sum of 4 squares. Its sole purpose is to demonstrate ordered rewriting for commutative rings.

The enterprising reader might consider proving all of Lagrange's theorem.

definition *sq* :: *'a*::*times* => *'a* **where** *sq x* == *x***x*

The following lemma essentially shows that every natural number is the sum of four squares, provided all prime numbers are. However, this is an abstract theorem about commutative rings. It has, a priori, nothing to do with nat.

lemma *Lagrange-lemma*: **fixes** *x1* :: *'a*::*comm-ring* **shows**
(*sq x1* + *sq x2* + *sq x3* + *sq x4*) * (*sq y1* + *sq y2* + *sq y3* + *sq y4*) =
sq (*x1***y1* − *x2***y2* − *x3***y3* − *x4***y4*) +
sq (*x1***y2* + *x2***y1* + *x3***y4* − *x4***y3*) +
sq (*x1***y3* − *x2***y4* + *x3***y1* + *x4***y2*) +
sq (*x1***y4* + *x2***y3* − *x3***y2* + *x4***y1*)
⟨*proof*⟩

A challenge by John Harrison. Takes about 12s on a 1.6GHz machine.

lemma fixes *p1* :: *'a*::*comm-ring* **shows**
(*sq p1* + *sq q1* + *sq r1* + *sq s1* + *sq t1* + *sq u1* + *sq v1* + *sq w1*) *
(*sq p2* + *sq q2* + *sq r2* + *sq s2* + *sq t2* + *sq u2* + *sq v2* + *sq w2*)
= *sq* (*p1***p2* − *q1***q2* − *r1***r2* − *s1***s2* − *t1***t2* − *u1***u2* − *v1***v2* − *w1***w2*)
+
sq (*p1***q2* + *q1***p2* + *r1***s2* − *s1***r2* + *t1***u2* − *u1***t2* − *v1***w2* + *w1***v2*)
+
sq (*p1***r2* − *q1***s2* + *r1***p2* + *s1***q2* + *t1***v2* + *u1***w2* − *v1***t2* − *w1***u2*)
+

$sq\ (p1*s2 + q1*r2 - r1*q2 + s1*p2 + t1*w2 - u1*v2 + v1*u2 - w1*t2)$
+
$sq\ (p1*t2 - q1*u2 - r1*v2 - s1*w2 + t1*p2 + u1*q2 + v1*r2 + w1*s2)$
+
$sq\ (p1*u2 + q1*t2 - r1*w2 + s1*v2 - t1*q2 + u1*p2 - v1*s2 + w1*r2)$
+
$sq\ (p1*v2 + q1*w2 + r1*t2 - s1*u2 - t1*r2 + u1*s2 + v1*p2 - w1*q2)$
+
$sq\ (p1*w2 - q1*v2 + r1*u2 + s1*t2 - t1*s2 - u1*r2 + v1*q2 + w1*p2)$
⟨*proof*⟩

end

35 Groebner Basis Examples

theory *Groebner-Examples*
imports ~~/*src*/*HOL*/*Groebner-Basis*
begin

35.1 Basic examples

lemma
fixes $x :: int$
shows $x \hat{} 3 = x \hat{} 3$
⟨*proof*⟩

lemma
fixes $x :: int$
shows $(x - (-2)) \hat{} 5 = x \hat{} 5 + (10 * x \hat{} 4 + (40 * x \hat{} 3 + (80 * x^2 + (80 * x + 32))))$
⟨*proof*⟩

schematic-goal
fixes $x :: int$
shows $(x - (-2)) \hat{} 5 * (y - 78) \hat{} 8 = ?X$
⟨*proof*⟩

lemma $((-3) \hat{} (Suc\ (Suc\ (Suc\ 0)))) == (X::'a::\{comm\text{-}ring\text{-}1\})$
⟨*proof*⟩

lemma $((x::int) + y) \hat{} 3 - 1 = (x - z) \hat{} 2 - 10 \Longrightarrow x = z + 3 \Longrightarrow x = - y$
⟨*proof*⟩

lemma $(4::nat) + 4 = 3 + 5$
⟨*proof*⟩

lemma $(4::int) + 0 = 4$
⟨*proof*⟩

lemma
assumes $a * x^2 + b * x + c = (0::int)$ **and** $d * x^2 + e * x + f = 0$
shows $d^2 * c^2 - 2 * d * c * a * f + a^2 * f^2 - e * d * b * c - e * b * a * f +$
$a * e^2 * c + f * d * b^2 = 0$
⟨*proof*⟩

lemma $(x::int)\hat{}3 - x\hat{}2 - 5*x - 3 = 0 \longleftrightarrow (x = 3 \vee x = -1)$
⟨*proof*⟩

theorem $x*(x^2 - x - 5) - 3 = (0::int) \longleftrightarrow (x = 3 \vee x = -1)$
⟨*proof*⟩

lemma
fixes $x::'a::idom$
shows $x^2*y = x^2 \;\&\; x*y^2 = y^2 \longleftrightarrow x = 1 \;\&\; y = 1 \mid x = 0 \;\&\; y = 0$
⟨*proof*⟩

35.2 Lemmas for Lagrange's theorem

definition
$sq :: 'a::times => 'a$ **where**
$sq\ x == x*x$

lemma
fixes $x1 :: 'a::\{idom\}$
shows
$(sq\ x1 + sq\ x2 + sq\ x3 + sq\ x4) * (sq\ y1 + sq\ y2 + sq\ y3 + sq\ y4) =$
$sq\ (x1*y1 - x2*y2 - x3*y3 - x4*y4) +$
$sq\ (x1*y2 + x2*y1 + x3*y4 - x4*y3) +$
$sq\ (x1*y3 - x2*y4 + x3*y1 + x4*y2) +$
$sq\ (x1*y4 + x2*y3 - x3*y2 + x4*y1)$
⟨*proof*⟩

lemma
fixes $p1 :: 'a::\{idom\}$
shows
$(sq\ p1 + sq\ q1 + sq\ r1 + sq\ s1 + sq\ t1 + sq\ u1 + sq\ v1 + sq\ w1) *$
$(sq\ p2 + sq\ q2 + sq\ r2 + sq\ s2 + sq\ t2 + sq\ u2 + sq\ v2 + sq\ w2)$
$= sq\ (p1*p2 - q1*q2 - r1*r2 - s1*s2 - t1*t2 - u1*u2 - v1*v2 - w1*w2)$
$+$
$sq\ (p1*q2 + q1*p2 + r1*s2 - s1*r2 + t1*u2 - u1*t2 - v1*w2 + w1*v2)$
$+$
$sq\ (p1*r2 - q1*s2 + r1*p2 + s1*q2 + t1*v2 + u1*w2 - v1*t2 - w1*u2)$
$+$
$sq\ (p1*s2 + q1*r2 - r1*q2 + s1*p2 + t1*w2 - u1*v2 + v1*u2 - w1*t2)$
$+$
$sq\ (p1*t2 - q1*u2 - r1*v2 - s1*w2 + t1*p2 + u1*q2 + v1*r2 + w1*s2)$
$+$
$sq\ (p1*u2 + q1*t2 - r1*w2 + s1*v2 - t1*q2 + u1*p2 - v1*s2 + w1*r2)$

+
    $sq\ (p1*v2 + q1*w2 + r1*t2 - s1*u2 - t1*r2 + u1*s2 + v1*p2 - w1*q2)$
+
    $sq\ (p1*w2 - q1*v2 + r1*u2 + s1*t2 - t1*s2 - u1*r2 + v1*q2 + w1*p2)$
  ⟨*proof*⟩

35.3 Colinearity is invariant by rotation

type-synonym *point* = *int* × *int*

definition *collinear* ::*point* ⇒ *point* ⇒ *point* ⇒ *bool* **where**
  collinear ≡ λ(*Ax*,*Ay*) (*Bx*,*By*) (*Cx*,*Cy*).
    ((*Ax* − *Bx*) ∗ (*By* − *Cy*) = (*Ay* − *By*) ∗ (*Bx* − *Cx*))

lemma *collinear-inv-rotation*:
  assumes *collinear* (*Ax*, *Ay*) (*Bx*, *By*) (*Cx*, *Cy*) **and** $c^2 + s^2 = 1$
  shows *collinear* (*Ax* ∗ *c* − *Ay* ∗ *s*, *Ay* ∗ *c* + *Ax* ∗ *s*)
    (*Bx* ∗ *c* − *By* ∗ *s*, *By* ∗ *c* + *Bx* ∗ *s*) (*Cx* ∗ *c* − *Cy* ∗ *s*, *Cy* ∗ *c* + *Cx* ∗ *s*)
  ⟨*proof*⟩

lemma *EX* (*d*::*int*). *a*∗*y* − *a*∗*x* = *n*∗*d* ⟹ *EX u v*. *a*∗*u* + *n*∗*v* = 1 ⟹ *EX e*. *y* − *x* = *n*∗*e*
  ⟨*proof*⟩

end

36 Substitution and Unification

theory *Unification*
imports *Main*
begin

Implements Manna & Waldinger's formalization, with Paulson's simplifications, and some new simplifications by Slind and Krauss.

Z Manna & R Waldinger, Deductive Synthesis of the Unification Algorithm. SCP 1 (1981), 5-48

L C Paulson, Verifying the Unification Algorithm in LCF. SCP 5 (1985), 143-170

K Slind, Reasoning about Terminating Functional Programs, Ph.D. thesis, TUM, 1999, Sect. 5.8

A Krauss, Partial and Nested Recursive Function Definitions in Higher-Order Logic, JAR 44(4):303-336, 2010. Sect. 6.3

36.1 Terms

Binary trees with leaves that are constants or variables.

datatype *'a trm* =
  Var 'a
| *Const 'a*
| *Comb 'a trm 'a trm* (**infix** · *60*)

primrec *vars-of* :: *'a trm* ⇒ *'a set*
where
  vars-of (*Var v*) = {*v*}
| *vars-of* (*Const c*) = {}
| *vars-of* (*M* · *N*) = *vars-of M* ∪ *vars-of N*

fun *occs* :: *'a trm* ⇒ *'a trm* ⇒ *bool* (**infixl** ≺ *54*)
where
  u ≺ *Var v* ⟷ *False*
| *u* ≺ *Const c* ⟷ *False*
| *u* ≺ *M* · *N* ⟷ *u* = *M* ∨ *u* = *N* ∨ *u* ≺ *M* ∨ *u* ≺ *N*

lemma *finite-vars-of*[*intro*]: *finite* (*vars-of t*)
  ⟨*proof*⟩

lemma *vars-iff-occseq*: *x* ∈ *vars-of t* ⟷ *Var x* ≺ *t* ∨ *Var x* = *t*
  ⟨*proof*⟩

lemma *occs-vars-subset*: *M* ≺ *N* ⟹ *vars-of M* ⊆ *vars-of N*
  ⟨*proof*⟩

36.2 Substitutions

type-synonym *'a subst* = (*'a* × *'a trm*) *list*

fun *assoc* :: *'a* ⇒ *'b* ⇒ (*'a* × *'b*) *list* ⇒ *'b*
where
  assoc x d [] = *d*
| *assoc x d* ((*p*,*q*)#*t*) = (*if x* = *p then q else assoc x d t*)

primrec *subst* :: *'a trm* ⇒ *'a subst* ⇒ *'a trm* (**infixl** ◁ *55*)
where
  (*Var v*) ◁ *s* = *assoc v* (*Var v*) *s*
| (*Const c*) ◁ *s* = (*Const c*)
| (*M* · *N*) ◁ *s* = (*M* ◁ *s*) · (*N* ◁ *s*)

definition *subst-eq* (**infixr** �doteq *52*)
where
  s1 �doteq *s2* ⟷ (∀ *t*. *t* ◁ *s1* = *t* ◁ *s2*)

fun *comp* :: *'a subst* ⇒ *'a subst* ⇒ *'a subst* (**infixl** ◊ *56*)
where
  [] ◊ *bl* = *bl*

| $((a,b) \# al) \diamondsuit bl = (a,\ b \lhd bl) \# (al \diamondsuit bl)$

lemma *subst-Nil*[*simp*]: $t \lhd [] = t$
⟨*proof*⟩

lemma *subst-mono*: $t \prec u \Longrightarrow t \lhd s \prec u \lhd s$
⟨*proof*⟩

lemma *agreement*: $(t \lhd r = t \lhd s) \longleftrightarrow (\forall v \in \mathit{vars\text{-}of}\ t.\ \mathit{Var}\ v \lhd r = \mathit{Var}\ v \lhd s)$
⟨*proof*⟩

lemma *repl-invariance*: $v \notin \mathit{vars\text{-}of}\ t \Longrightarrow t \lhd (v,u) \# s = t \lhd s$
⟨*proof*⟩

lemma *remove-var*: $v \notin \mathit{vars\text{-}of}\ s \Longrightarrow v \notin \mathit{vars\text{-}of}\ (t \lhd [(v,\ s)])$
⟨*proof*⟩

lemma *subst-refl*[*iff*]: $s \doteq s$
⟨*proof*⟩

lemma *subst-sym*[*sym*]: $[\![s1 \doteq s2]\!] \Longrightarrow s2 \doteq s1$
⟨*proof*⟩

lemma *subst-trans*[*trans*]: $[\![s1 \doteq s2;\ s2 \doteq s3]\!] \Longrightarrow s1 \doteq s3$
⟨*proof*⟩

lemma *subst-no-occs*: $\neg\ \mathit{Var}\ v \prec t \Longrightarrow \mathit{Var}\ v \neq t \Longrightarrow t \lhd [(v,s)] = t$
⟨*proof*⟩

lemma *comp-Nil*[*simp*]: $\sigma \diamondsuit [] = \sigma$
⟨*proof*⟩

lemma *subst-comp*[*simp*]: $t \lhd (r \diamondsuit s) = t \lhd r \lhd s$
⟨*proof*⟩

lemma *subst-eq-intro*[*intro*]: $(\bigwedge t.\ t \lhd \sigma = t \lhd \vartheta) \Longrightarrow \sigma \doteq \vartheta$
⟨*proof*⟩

lemma *subst-eq-dest*[*dest*]: $s1 \doteq s2 \Longrightarrow t \lhd s1 = t \lhd s2$
⟨*proof*⟩

lemma *comp-assoc*: $(a \diamondsuit b) \diamondsuit c \doteq a \diamondsuit (b \diamondsuit c)$
⟨*proof*⟩

lemma *subst-cong*: $[\![\sigma \doteq \sigma';\ \vartheta \doteq \vartheta']\!] \Longrightarrow (\sigma \diamondsuit \vartheta) \doteq (\sigma' \diamondsuit \vartheta')$
⟨*proof*⟩

lemma *var-self*: $[(v,\ Var\ v)] \doteq []$
⟨*proof*⟩

lemma *var-same*[*simp*]: $[(v,\ t)] \doteq [] \longleftrightarrow t = Var\ v$
⟨*proof*⟩

36.3 Unifiers and Most General Unifiers

definition *Unifier* :: *'a subst* ⇒ *'a trm* ⇒ *'a trm* ⇒ *bool*
where $Unifier\ \sigma\ t\ u \longleftrightarrow (t \lhd \sigma = u \lhd \sigma)$

definition *MGU* :: *'a subst* ⇒ *'a trm* ⇒ *'a trm* ⇒ *bool* **where**
$MGU\ \sigma\ t\ u \longleftrightarrow$
$Unifier\ \sigma\ t\ u \wedge (\forall \vartheta.\ Unifier\ \vartheta\ t\ u \longrightarrow (\exists \gamma.\ \vartheta \doteq \sigma \Diamond \gamma))$

lemma *MGUI*[*intro*]:
$[\![t \lhd \sigma = u \lhd \sigma;\ \bigwedge \vartheta.\ t \lhd \vartheta = u \lhd \vartheta \Longrightarrow \exists \gamma.\ \vartheta \doteq \sigma \Diamond \gamma]\!]$
$\Longrightarrow MGU\ \sigma\ t\ u$
⟨*proof*⟩

lemma *MGU-sym*[*sym*]:
$MGU\ \sigma\ s\ t \Longrightarrow MGU\ \sigma\ t\ s$
⟨*proof*⟩

lemma *MGU-is-Unifier*: $MGU\ \sigma\ t\ u \Longrightarrow Unifier\ \sigma\ t\ u$
⟨*proof*⟩

lemma *MGU-Var*:
assumes $\neg\ Var\ v \prec t$
shows $MGU\ [(v,t)]\ (Var\ v)\ t$
⟨*proof*⟩

lemma *MGU-Const*: $MGU\ []\ (Const\ c)\ (Const\ d) \longleftrightarrow c = d$
⟨*proof*⟩

36.4 The unification algorithm

function *unify* :: *'a trm* ⇒ *'a trm* ⇒ *'a subst option*
where
$unify\ (Const\ c)\ (M \cdot N) = None$
| $unify\ (M \cdot N)\ (Const\ c) = None$
| $unify\ (Const\ c)\ (Var\ v) = Some\ [(v,\ Const\ c)]$
| $unify\ (M \cdot N)\ (Var\ v) = (if\ Var\ v \prec M \cdot N$
$then\ None$
$else\ Some\ [(v,\ M \cdot N)])$
| $unify\ (Var\ v)\ M = (if\ Var\ v \prec M$
$then\ None$
$else\ Some\ [(v,\ M)])$
| $unify\ (Const\ c)\ (Const\ d) = (if\ c{=}d\ then\ Some\ []\ else\ None)$
| $unify\ (M \cdot N)\ (M' \cdot N') = (case\ unify\ M\ M'\ of$

None ⇒ *None* |
Some ϑ ⇒ (*case unify* (*N* ◁ ϑ) (*N′* ◁ ϑ)
of None ⇒ *None* |
Some σ ⇒ *Some* (ϑ ◇ σ)))

⟨*proof*⟩

36.5 Properties used in termination proof

Elimination of variables by a substitution:

definition
elim σ *v* ≡ ∀ *t*. *v* ∉ *vars-of* (*t* ◁ σ)

lemma *elim-intro*[*intro*]: (⋀*t*. *v* ∉ *vars-of* (*t* ◁ σ)) ⟹ *elim* σ *v*
⟨*proof*⟩

lemma *elim-dest*[*dest*]: *elim* σ *v* ⟹ *v* ∉ *vars-of* (*t* ◁ σ)
⟨*proof*⟩

lemma *elim-eq*: σ ≐ ϑ ⟹ *elim* σ *x* = *elim* ϑ *x*
⟨*proof*⟩

lemma *occs-elim*: ¬ *Var v* ≺ *t*
⟹ *elim* [(*v*,*t*)] *v* ∨ [(*v*,*t*)] ≐ []
⟨*proof*⟩

The result of a unification never introduces new variables:

declare *unify.psimps*[*simp*]

lemma *unify-vars*:
assumes *unify-dom* (*M*, *N*)
assumes *unify M N* = *Some* σ
shows *vars-of* (*t* ◁ σ) ⊆ *vars-of M* ∪ *vars-of N* ∪ *vars-of t*
(**is** *?P M N* σ *t*)
⟨*proof*⟩

The result of a unification is either the identity substitution or it eliminates a variable from one of the terms:

lemma *unify-eliminates*:
assumes *unify-dom* (*M*, *N*)
assumes *unify M N* = *Some* σ
shows (∃ *v*∈*vars-of M* ∪ *vars-of N*. *elim* σ *v*) ∨ σ ≐ []
(**is** *?P M N* σ)
⟨*proof*⟩

declare *unify.psimps*[*simp del*]

36.6 Termination proof

termination *unify*

⟨*proof*⟩

36.7 Unification returns a Most General Unifier

lemma *unify-computes-MGU*:
unify M N = Some σ ⟹ MGU σ M N
⟨*proof*⟩

36.8 Unification returns Idempotent Substitution

definition *Idem* :: *'a subst ⇒ bool*
where *Idem s ⟷ (s ◇ s) ≐ s*

lemma *Idem-Nil* [*iff*]: *Idem []*
⟨*proof*⟩

lemma *Var-Idem*:
assumes ~ *(Var v ≺ t)* **shows** *Idem* [*(v,t)*]
⟨*proof*⟩

lemma *Unifier-Idem-subst*:
Idem(r) ⟹ Unifier s (t ◁ r) (u ◁ r) ⟹
Unifier (r ◇ s) (t ◁ r) (u ◁ r)
⟨*proof*⟩

lemma *Idem-comp*:
Idem r ⟹ Unifier s (t ◁ r) (u ◁ r) ⟹
(!!q. Unifier q (t ◁ r) (u ◁ r) ⟹ s ◇ q ≐ q) ⟹
Idem (r ◇ s)
⟨*proof*⟩

theorem *unify-gives-Idem*:
unify M N = Some σ ⟹ Idem σ
⟨*proof*⟩

end

37 Primitive Recursive Functions

theory *Primrec* **imports** *Main* **begin**

Proof adopted from

Nora Szasz, A Machine Checked Proof that Ackermann's Function is not Primitive Recursive, In: Huet & Plotkin, eds., Logical Environments (CUP, 1993), 317-338.

See also E. Mendelson, Introduction to Mathematical Logic. (Van Nostrand, 1964), page 250, exercise 11.

37.1 Ackermann's Function

fun *ack* :: *nat* => *nat* => *nat* **where**
ack 0 n = *Suc n* |
ack (*Suc m*) *0* = *ack m 1* |
ack (*Suc m*) (*Suc n*) = *ack m* (*ack* (*Suc m*) *n*)

PROPERTY A 4

lemma *less-ack2* [*iff*]: *j* < *ack i j*
⟨*proof*⟩

PROPERTY A 5-, the single-step lemma

lemma *ack-less-ack-Suc2* [*iff*]: *ack i j* < *ack i* (*Suc j*)
⟨*proof*⟩

PROPERTY A 5, monotonicity for <

lemma *ack-less-mono2*: *j* < *k* ==> *ack i j* < *ack i k*
⟨*proof*⟩

PROPERTY A 5', monotonicity for ≤

lemma *ack-le-mono2*: *j* ≤ *k* ==> *ack i j* ≤ *ack i k*
⟨*proof*⟩

PROPERTY A 6

lemma *ack2-le-ack1* [*iff*]: *ack i* (*Suc j*) ≤ *ack* (*Suc i*) *j*
⟨*proof*⟩

PROPERTY A 7-, the single-step lemma

lemma *ack-less-ack-Suc1* [*iff*]: *ack i j* < *ack* (*Suc i*) *j*
⟨*proof*⟩

PROPERTY A 4'? Extra lemma needed for *CONSTANT* case, constant functions

lemma *less-ack1* [*iff*]: *i* < *ack i j*
⟨*proof*⟩

PROPERTY A 8

lemma *ack-1* [*simp*]: *ack* (*Suc 0*) *j* = *j* + *2*
⟨*proof*⟩

PROPERTY A 9. The unary *1* and *2* in *ack* is essential for the rewriting.

lemma *ack-2* [*simp*]: *ack* (*Suc* (*Suc 0*)) *j* = *2* * *j* + *3*
⟨*proof*⟩

PROPERTY A 7, monotonicity for < [not clear why *ack-1* is now needed first!]

lemma *ack-less-mono1-aux*: *ack i k* < *ack* (*Suc* (*i* + *i'*)) *k*

⟨*proof*⟩

lemma *ack-less-mono1*: $i < j \Longrightarrow ack\ i\ k < ack\ j\ k$
⟨*proof*⟩

PROPERTY A 7', monotonicity for $\leq$

lemma *ack-le-mono1*: $i \leq j \Longrightarrow ack\ i\ k \leq ack\ j\ k$
⟨*proof*⟩

PROPERTY A 10

lemma *ack-nest-bound*: $ack\ i1\ (ack\ i2\ j) < ack\ (2 + (i1 + i2))\ j$
⟨*proof*⟩

PROPERTY A 11

lemma *ack-add-bound*: $ack\ i1\ j + ack\ i2\ j < ack\ (4 + (i1 + i2))\ j$
⟨*proof*⟩

PROPERTY A 12. Article uses existential quantifier but the ALF proof used $k + 4$. Quantified version must be nested $\exists\, k'.\ \forall\, i\ j.$ ...

lemma *ack-add-bound2*: $i < ack\ k\ j \Longrightarrow i + j < ack\ (4 + k)\ j$
⟨*proof*⟩

37.2 Primitive Recursive Functions

primrec *hd0* :: *nat list* => *nat* **where**
hd0 [] = *0* |
hd0 (*m* # *ms*) = *m*

Inductive definition of the set of primitive recursive functions of type *nat list* ⇒ *nat*.

definition *SC* :: *nat list* => *nat* **where**
SC l = *Suc* (*hd0 l*)

definition *CONSTANT* :: *nat* => *nat list* => *nat* **where**
CONSTANT k l = *k*

definition *PROJ* :: *nat* => *nat list* => *nat* **where**
PROJ i l = *hd0* (*drop i l*)

definition
COMP :: (*nat list* => *nat*) => (*nat list* => *nat*) *list* => *nat list* => *nat*
where *COMP g fs l* = *g* (*map* (λ*f*. *f l*) *fs*)

definition *PREC* :: (*nat list* => *nat*) => (*nat list* => *nat*) => *nat list* => *nat*
where
PREC f g l =
(*case l of*
[] => *0*

| $x \# l' \Rightarrow$ *rec-nat* $(f\ l')\ (\lambda y\ r.\ g\ (r \# y \# l'))\ x$)
— Note that g is applied first to *PREC* $f\ g\ y$ and then to y!

inductive *PRIMREC* :: (*nat list* => *nat*) => *bool* **where**
SC: *PRIMREC SC* |
CONSTANT: *PRIMREC* (*CONSTANT k*) |
PROJ: *PRIMREC* (*PROJ i*) |
COMP: *PRIMREC g* ==> $\forall f \in$ *set fs*. *PRIMREC f* ==> *PRIMREC* (*COMP g fs*) |
PREC: *PRIMREC f* ==> *PRIMREC g* ==> *PRIMREC* (*PREC f g*)

Useful special cases of evaluation

lemma *SC* [*simp*]: *SC* $(x \# l) =$ *Suc x*
⟨*proof*⟩

lemma *CONSTANT* [*simp*]: *CONSTANT k l* = *k*
⟨*proof*⟩

lemma *PROJ-0* [*simp*]: *PROJ 0* $(x \# l) = x$
⟨*proof*⟩

lemma *COMP-1* [*simp*]: *COMP g* [*f*] *l* = *g* [*f l*]
⟨*proof*⟩

lemma *PREC-0* [*simp*]: *PREC f g* $(0 \# l) = f\ l$
⟨*proof*⟩

lemma *PREC-Suc* [*simp*]: *PREC f g* $(Suc\ x \# l) = g\ (PREC\ f\ g\ (x \# l) \# x \# l)$
⟨*proof*⟩

MAIN RESULT

lemma *SC-case*: *SC l* < *ack 1* (*sum-list l*)
⟨*proof*⟩

lemma *CONSTANT-case*: *CONSTANT k l* < *ack k* (*sum-list l*)
⟨*proof*⟩

lemma *PROJ-case*: *PROJ i l* < *ack 0* (*sum-list l*)
⟨*proof*⟩

COMP case

lemma *COMP-map-aux*: $\forall f \in$ *set fs*. *PRIMREC f* $\wedge$ $(\exists kf.\ \forall l.\ f\ l <$ *ack kf* (*sum-list l*))
==> $\exists k.\ \forall l.$ *sum-list* (*map* $(\lambda f.\ f\ l)$ *fs*) < *ack k* (*sum-list l*)
⟨*proof*⟩

lemma *COMP-case*:
$\forall l.$ *g l* < *ack kg* (*sum-list l*) ==>

$\forall f \in set\ fs.\ PRIMREC\ f \wedge (\exists kf.\ \forall l.\ f\ l < ack\ kf\ (sum\text{-}list\ l))$
$\Longrightarrow \exists k.\ \forall l.\ COMP\ g\ fs\ \ l < ack\ k\ (sum\text{-}list\ l)$
⟨*proof*⟩

PREC case

lemma *PREC-case-aux*:
$\forall l.\ f\ l + sum\text{-}list\ l < ack\ kf\ (sum\text{-}list\ l) \Longrightarrow$
$\forall l.\ g\ l + sum\text{-}list\ l < ack\ kg\ (sum\text{-}list\ l) \Longrightarrow$
$PREC\ f\ g\ l + sum\text{-}list\ l < ack\ (Suc\ (kf + kg))\ (sum\text{-}list\ l)$
⟨*proof*⟩

lemma *PREC-case*:
$\forall l.\ f\ l < ack\ kf\ (sum\text{-}list\ l) \Longrightarrow$
$\forall l.\ g\ l < ack\ kg\ (sum\text{-}list\ l) \Longrightarrow$
$\exists k.\ \forall l.\ PREC\ f\ g\ l < ack\ k\ (sum\text{-}list\ l)$
⟨*proof*⟩

lemma *ack-bounds-PRIMREC*: $PRIMREC\ f \Longrightarrow \exists k.\ \forall l.\ f\ l < ack\ k\ (sum\text{-}list\ l)$
⟨*proof*⟩

theorem *ack-not-PRIMREC*:
$\neg\ PRIMREC\ (\lambda l.\ case\ l\ of\ [] \Rightarrow 0 \mid x \# l' \Rightarrow ack\ x\ x)$
⟨*proof*⟩

end

38 The Full Theorem of Tarski

theory *Tarski*
imports *Main* ~~/*src*/*HOL*/*Library*/*FuncSet*
begin

Minimal version of lattice theory plus the full theorem of Tarski: The fixed-points of a complete lattice themselves form a complete lattice.

Illustrates first-class theories, using the Sigma representation of structures. Tidied and converted to Isar by lcp.

record *'a potype* =
pset :: *'a set*
order :: *('a * 'a) set*

definition
monotone :: *['a => 'a, 'a set, ('a *'a)set] => bool* **where**
monotone f A r = $(\forall x{\in}A.\ \forall y{\in}A.\ (x, y): r \longrightarrow ((f\ x), (f\ y)) : r)$

definition
least :: *['a => bool, 'a potype] => 'a* **where**
least P po = (*SOME x. x: pset po & P x &*

$(\forall\, y \in pset\ po.\ P\ y \longrightarrow (x,y)\colon order\ po))$

definition
greatest :: $['a \Rightarrow bool,\ 'a\ potype] \Rightarrow 'a$ **where**
greatest P po = $(SOME\ x.\ x\colon pset\ po\ \&\ P\ x\ \&$
$(\forall\, y \in pset\ po.\ P\ y \longrightarrow (y,x)\colon order\ po))$

definition
lub :: $['a\ set,\ 'a\ potype] \Rightarrow 'a$ **where**
lub S po = $least\ (\%x.\ \forall\, y{\in}S.\ (y,x)\colon order\ po)\ po$

definition
glb :: $['a\ set,\ 'a\ potype] \Rightarrow 'a$ **where**
glb S po = $greatest\ (\%x.\ \forall\, y{\in}S.\ (x,y)\colon order\ po)\ po$

definition
isLub :: $['a\ set,\ 'a\ potype,\ 'a] \Rightarrow bool$ **where**
isLub S po = $(\%L.\ (L\colon pset\ po\ \&\ (\forall\, y{\in}S.\ (y,L)\colon order\ po)\ \&$
$(\forall\, z{\in}pset\ po.\ (\forall\, y{\in}S.\ (y,z)\colon order\ po) \longrightarrow (L,z)\colon order\ po)))$

definition
isGlb :: $['a\ set,\ 'a\ potype,\ 'a] \Rightarrow bool$ **where**
isGlb S po = $(\%G.\ (G\colon pset\ po\ \&\ (\forall\, y{\in}S.\ (G,y)\colon order\ po)\ \&$
$(\forall\, z \in pset\ po.\ (\forall\, y{\in}S.\ (z,y)\colon order\ po) \longrightarrow (z,G)\colon order\ po)))$

definition
fix :: $[('a \Rightarrow 'a),\ 'a\ set] \Rightarrow 'a\ set$ **where**
fix f A = $\{x.\ x\colon A\ \&\ f\ x = x\}$

definition
interval :: $[('a{*}'a)\ set, 'a,\ 'a\ ] \Rightarrow 'a\ set$ **where**
interval r a b = $\{x.\ (a,x)\colon r\ \&\ (x,b)\colon r\}$

definition
Bot :: $'a\ potype \Rightarrow 'a$ **where**
Bot po = $least\ (\%x.\ True)\ po$

definition
Top :: $'a\ potype \Rightarrow 'a$ **where**
Top po = $greatest\ (\%x.\ True)\ po$

definition
PartialOrder :: $('a\ potype)\ set$ **where**
PartialOrder = $\{P.\ refl\text{-}on\ (pset\ P)\ (order\ P)\ \&\ antisym\ (order\ P)\ \&$
$trans\ (order\ P)\}$

definition
CompleteLattice :: $('a\ potype)\ set$ **where**

```
  CompleteLattice = {cl. cl: PartialOrder &
                        (∀ S. S ⊆ pset cl --> (∃ L. isLub S cl L)) &
                        (∀ S. S ⊆ pset cl --> (∃ G. isGlb S cl G))}

definition
  CLF-set :: ('a potype * ('a => 'a)) set where
  CLF-set = (SIGMA cl: CompleteLattice.
          {f. f: pset cl → pset cl & monotone f (pset cl) (order cl)})

definition
  induced :: ['a set, ('a * 'a) set] => ('a *'a)set where
  induced A r = {(a,b). a : A & b: A & (a,b): r}


definition
  sublattice :: ('a potype * 'a set)set where
  sublattice =
     (SIGMA cl: CompleteLattice.
        {S. S ⊆ pset cl &
         (| pset = S, order = induced S (order cl) |): CompleteLattice})

abbreviation
  sublat :: ['a set, 'a potype] => bool  (- <<= - [51,50]50) where
  S <<= cl == S : sublattice `` {cl}

definition
  dual :: 'a potype => 'a potype where
  dual po = (| pset = pset po, order = converse (order po) |)

locale S =
  fixes cl :: 'a potype
    and A  :: 'a set
    and r  :: ('a * 'a) set
  defines A-def: A == pset cl
     and  r-def: r == order cl

locale PO = S +
  assumes cl-po:  cl : PartialOrder

locale CL = S +
  assumes cl-co:  cl : CompleteLattice

sublocale CL < po?: PO
⟨proof⟩

locale CLF = S +
  fixes f :: 'a => 'a
    and P :: 'a set
  assumes f-cl:  (cl,f) : CLF-set
```

defines *P-def*: $P == fix\ f\ A$

sublocale $CLF < cl?: CL$
⟨*proof*⟩

locale $Tarski = CLF +$
fixes $Y \quad :: 'a\ set$
and $intY1 :: 'a\ set$
and $v \quad :: 'a$
assumes
Y-ss: $Y \subseteq P$
defines
intY1-def: $intY1 == interval\ r\ (lub\ Y\ cl)\ (Top\ cl)$
and *v-def*: $v == glb\ \{x.\ ((\%x{:}\ intY1.\ f\ x)\ x,\ x){:}\ induced\ intY1\ r\ \&$
$x{:}\ intY1\}$
$(|\ pset{=}intY1,\ order{=}induced\ intY1\ r|)$

38.1 Partial Order

lemma (**in** *PO*) *dual*:
$PO\ (dual\ cl)$
⟨*proof*⟩

lemma (**in** *PO*) *PO-imp-refl-on* [*simp*]: $refl\text{-}on\ A\ r$
⟨*proof*⟩

lemma (**in** *PO*) *PO-imp-sym* [*simp*]: $antisym\ r$
⟨*proof*⟩

lemma (**in** *PO*) *PO-imp-trans* [*simp*]: $trans\ r$
⟨*proof*⟩

lemma (**in** *PO*) *reflE*: $x \in A ==> (x, x) \in r$
⟨*proof*⟩

lemma (**in** *PO*) *antisymE*: $[|\ (a, b) \in r; (b, a) \in r\ |] ==> a = b$
⟨*proof*⟩

lemma (**in** *PO*) *transE*: $[|\ (a, b) \in r; (b, c) \in r|] ==> (a,c) \in r$
⟨*proof*⟩

lemma (**in** *PO*) *monotoneE*:
$[|\ monotone\ f\ A\ r;\ \ x \in A; y \in A; (x, y) \in r\ |] ==> (f\ x, f\ y) \in r$
⟨*proof*⟩

lemma (**in** *PO*) *po-subset-po*:
$S \subseteq A ==> (|\ pset = S, order = induced\ S\ r\ |) \in PartialOrder$
⟨*proof*⟩

lemma (**in** *PO*) *indE*: [| $(x, y) \in$ *induced S r*; $S \subseteq A$ |] ==> $(x, y) \in r$
⟨*proof*⟩

lemma (**in** *PO*) *indI*: [| $(x, y) \in r$; $x \in S$; $y \in S$ |] ==> $(x, y) \in$ *induced S r*
⟨*proof*⟩

lemma (**in** *CL*) *CL-imp-ex-isLub*: $S \subseteq A$ ==> $\exists L.$ *isLub S cl L*
⟨*proof*⟩

declare (**in** *CL*) *cl-co* [*simp*]

lemma *isLub-lub*: $(\exists L.$ *isLub S cl L*$) =$ *isLub S cl* (*lub S cl*)
⟨*proof*⟩

lemma *isGlb-glb*: $(\exists G.$ *isGlb S cl G*$) =$ *isGlb S cl* (*glb S cl*)
⟨*proof*⟩

lemma *isGlb-dual-isLub*: *isGlb S cl* = *isLub S* (*dual cl*)
⟨*proof*⟩

lemma *isLub-dual-isGlb*: *isLub S cl* = *isGlb S* (*dual cl*)
⟨*proof*⟩

lemma (**in** *PO*) *dualPO*: *dual cl* ∈ *PartialOrder*
⟨*proof*⟩

lemma *Rdual*:
$\forall S.$ ($S \subseteq A$ −−>($\exists L.$ *isLub S* (| *pset* = *A*, *order* = *r*|) *L*))
==> $\forall S.$ ($S \subseteq A$ −−> ($\exists G.$ *isGlb S* (| *pset* = *A*, *order* = *r*|) *G*))
⟨*proof*⟩

lemma *lub-dual-glb*: *lub S cl* = *glb S* (*dual cl*)
⟨*proof*⟩

lemma *glb-dual-lub*: *glb S cl* = *lub S* (*dual cl*)
⟨*proof*⟩

lemma *CL-subset-PO*: *CompleteLattice* ⊆ *PartialOrder*
⟨*proof*⟩

lemmas *CL-imp-PO* = *CL-subset-PO* [*THEN subsetD*]

lemma (**in** *CL*) *CO-refl-on*: *refl-on A r*
⟨*proof*⟩

lemma (**in** *CL*) *CO-antisym*: *antisym r*
⟨*proof*⟩

lemma (**in** *CL*) *CO-trans*: *trans r*
⟨*proof*⟩

lemma *CompleteLatticeI*:
[| *po* ∈ *PartialOrder*; (∀ *S*. *S* ⊆ *pset po* −−> (∃ *L*. *isLub S po L*));
(∀ *S*. *S* ⊆ *pset po* −−> (∃ *G*. *isGlb S po G*))|]
==> *po* ∈ *CompleteLattice*
⟨*proof*⟩

lemma (**in** *CL*) *CL-dualCL*: *dual cl* ∈ *CompleteLattice*
⟨*proof*⟩

lemma (**in** *PO*) *dualA-iff*: *pset* (*dual cl*) = *pset cl*
⟨*proof*⟩

lemma (**in** *PO*) *dualr-iff*: ((*x*, *y*) ∈ (*order*(*dual cl*))) = ((*y*, *x*) ∈ *order cl*)
⟨*proof*⟩

lemma (**in** *PO*) *monotone-dual*:
monotone f (*pset cl*) (*order cl*)
==> *monotone f* (*pset* (*dual cl*)) (*order*(*dual cl*))
⟨*proof*⟩

lemma (**in** *PO*) *interval-dual*:
[| *x* ∈ *A*; *y* ∈ *A*|] ==> *interval r x y* = *interval* (*order*(*dual cl*)) *y x*
⟨*proof*⟩

lemma (**in** *PO*) *trans*:
(*x*, *y*) ∈ *r* ⟹ (*y*, *z*) ∈ *r* ⟹ (*x*, *z*) ∈ *r*
⟨*proof*⟩

lemma (**in** *PO*) *interval-not-empty*:
interval r a b ≠ {} ==> (*a*, *b*) ∈ *r*
⟨*proof*⟩

lemma (**in** *PO*) *interval-imp-mem*: *x* ∈ *interval r a b* ==> (*a*, *x*) ∈ *r*
⟨*proof*⟩

lemma (**in** *PO*) *left-in-interval*:
[| *a* ∈ *A*; *b* ∈ *A*; *interval r a b* ≠ {} |] ==> *a* ∈ *interval r a b*
⟨*proof*⟩

lemma (**in** *PO*) *right-in-interval*:
[| *a* ∈ *A*; *b* ∈ *A*; *interval r a b* ≠ {} |] ==> *b* ∈ *interval r a b*
⟨*proof*⟩

38.2 sublattice

lemma (**in** *PO*) *sublattice-imp-CL*:
$S <<= cl \implies (| pset = S, order = induced\ S\ r |) \in CompleteLattice$
⟨*proof*⟩

lemma (**in** *CL*) *sublatticeI*:
$[| S \subseteq A; (| pset = S, order = induced\ S\ r |) \in CompleteLattice |] \implies S <<= cl$
⟨*proof*⟩

lemma (**in** *CL*) *dual*:
$CL\ (dual\ cl)$
⟨*proof*⟩

38.3 lub

lemma (**in** *CL*) *lub-unique*: $[| S \subseteq A; isLub\ S\ cl\ x; isLub\ S\ cl\ L|] \implies x = L$
⟨*proof*⟩

lemma (**in** *CL*) *lub-upper*: $[|S \subseteq A; x \in S|] \implies (x, lub\ S\ cl) \in r$
⟨*proof*⟩

lemma (**in** *CL*) *lub-least*:
$[| S \subseteq A; L \in A; \forall x \in S.\ (x,L) \in r\ |] \implies (lub\ S\ cl, L) \in r$
⟨*proof*⟩

lemma (**in** *CL*) *lub-in-lattice*: $S \subseteq A \implies lub\ S\ cl \in A$
⟨*proof*⟩

lemma (**in** *CL*) *lubI*:
$[| S \subseteq A; L \in A; \forall x \in S.\ (x,L) \in r;$
$\forall z \in A.\ (\forall y \in S.\ (y,z) \in r) \longrightarrow (L,z) \in r\ |] \implies L = lub\ S\ cl$
⟨*proof*⟩

lemma (**in** *CL*) *lubIa*: $[| S \subseteq A; isLub\ S\ cl\ L\ |] \implies L = lub\ S\ cl$
⟨*proof*⟩

lemma (**in** *CL*) *isLub-in-lattice*: $isLub\ S\ cl\ L \implies L \in A$
⟨*proof*⟩

lemma (**in** *CL*) *isLub-upper*: $[|isLub\ S\ cl\ L; y \in S|] \implies (y, L) \in r$
⟨*proof*⟩

lemma (**in** *CL*) *isLub-least*:
$[| isLub\ S\ cl\ L; z \in A; \forall y \in S.\ (y, z) \in r|] \implies (L, z) \in r$
⟨*proof*⟩

lemma (**in** *CL*) *isLubI*:
$[| L \in A; \forall y \in S.\ (y, L) \in r;$

$(\forall z \in A.\ (\forall y \in S.\ (y,\ z){:}r) \longrightarrow (L,\ z) \in r)|] \Longrightarrow isLub\ S\ cl\ L$
⟨*proof*⟩

38.4 glb

lemma (**in** *CL*) *glb-in-lattice*: $S \subseteq A \Longrightarrow glb\ S\ cl \in A$
⟨*proof*⟩

lemma (**in** *CL*) *glb-lower*: $[|S \subseteq A;\ x \in S|] \Longrightarrow (glb\ S\ cl,\ x) \in r$
⟨*proof*⟩

Reduce the sublattice property by using substructural properties; abandoned see *Tarski-4.ML*.

lemma (**in** *CLF*) [*simp*]:
f: $pset\ cl \to pset\ cl\ \&\ monotone\ f\ (pset\ cl)\ (order\ cl)$
⟨*proof*⟩

declare (**in** *CLF*) *f-cl* [*simp*]

lemma (**in** *CLF*) *f-in-funcset*: $f \in A \to A$
⟨*proof*⟩

lemma (**in** *CLF*) *monotone-f*: $monotone\ f\ A\ r$
⟨*proof*⟩

lemma (**in** *CLF*) *CLF-dual*: $(dual\ cl,\ f) \in CLF\text{-}set$
⟨*proof*⟩

lemma (**in** *CLF*) *dual*:
$CLF\ (dual\ cl)\ f$
⟨*proof*⟩

38.5 fixed points

lemma *fix-subset*: $fix\ f\ A \subseteq A$
⟨*proof*⟩

lemma *fix-imp-eq*: $x \in fix\ f\ A \Longrightarrow f\ x = x$
⟨*proof*⟩

lemma *fixf-subset*:
$[|\ A \subseteq B;\ x \in fix\ (\%y{:}\ A.\ f\ y)\ A\ |] \Longrightarrow x \in fix\ f\ B$
⟨*proof*⟩

38.6 lemmas for Tarski, lub

lemma (**in** *CLF*) *lubH-le-flubH*:
$H = \{x.\ (x,\ f\ x) \in r\ \&\ x \in A\} \Longrightarrow (lub\ H\ cl,\ f\ (lub\ H\ cl)) \in r$

⟨*proof*⟩

lemma (**in** *CLF*) *flubH-le-lubH*:
[| $H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\}$ |] ==> $(f\ (lub\ H\ cl), lub\ H\ cl) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *lubH-is-fixp*:
$H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\}$ ==> $lub\ H\ cl \in fix\ f\ A$
⟨*proof*⟩

lemma (**in** *CLF*) *fix-in-H*:
[| $H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\};\ x \in P$ |] ==> $x \in H$
⟨*proof*⟩

lemma (**in** *CLF*) *fixf-le-lubH*:
$H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\}$ ==> $\forall x \in fix\ f\ A.\ (x, lub\ H\ cl) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *lubH-least-fixf*:
$H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\}$
==> $\forall L.\ (\forall y \in fix\ f\ A.\ (y,L) \in r) \longrightarrow (lub\ H\ cl, L) \in r$
⟨*proof*⟩

38.7 Tarski fixpoint theorem 1, first part

lemma (**in** *CLF*) *T-thm-1-lub*: $lub\ P\ cl = lub\ \{x.\ (x, f\ x) \in r\ \&\ x \in A\}\ cl$
⟨*proof*⟩

lemma (**in** *CLF*) *glbH-is-fixp*: $H = \{x.\ (f\ x, x) \in r\ \&\ x \in A\}$ ==> $glb\ H\ cl \in P$
— Tarski for glb
⟨*proof*⟩

lemma (**in** *CLF*) *T-thm-1-glb*: $glb\ P\ cl = glb\ \{x.\ (f\ x, x) \in r\ \&\ x \in A\}\ cl$
⟨*proof*⟩

38.8 interval

lemma (**in** *CLF*) *rel-imp-elem*: $(x, y) \in r$ ==> $x \in A$
⟨*proof*⟩

lemma (**in** *CLF*) *interval-subset*: [| $a \in A$; $b \in A$ |] ==> $interval\ r\ a\ b \subseteq A$
⟨*proof*⟩

lemma (**in** *CLF*) *intervalI*:
[| $(a, x) \in r$; $(x, b) \in r$ |] ==> $x \in interval\ r\ a\ b$
⟨*proof*⟩

lemma (**in** *CLF*) *interval-lemma1*:
[| $S \subseteq interval\ r\ a\ b$; $x \in S$ |] ==> $(a, x) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *interval-lemma2*:
[| $S \subseteq$ *interval r a b*; $x \in S$ |] ==> $(x, b) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *a-less-lub*:
[| $S \subseteq A$; $S \neq \{\}$;
$\forall x \in S.\ (a,x) \in r$; $\forall y \in S.\ (y, L) \in r$ |] ==> $(a,L) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *glb-less-b*:
[| $S \subseteq A$; $S \neq \{\}$;
$\forall x \in S.\ (x,b) \in r$; $\forall y \in S.\ (G, y) \in r$ |] ==> $(G,b) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *S-intv-cl*:
[| $a \in A$; $b \in A$; $S \subseteq$ *interval r a b* |]==> $S \subseteq A$
⟨*proof*⟩

lemma (**in** *CLF*) *L-in-interval*:
[| $a \in A$; $b \in A$; $S \subseteq$ *interval r a b*;
$S \neq \{\}$; *isLub S cl L*; *interval r a b* $\neq \{\}$ |] ==> $L \in$ *interval r a b*
⟨*proof*⟩

lemma (**in** *CLF*) *G-in-interval*:
[| $a \in A$; $b \in A$; *interval r a b* $\neq \{\}$; $S \subseteq$ *interval r a b*; *isGlb S cl G*;
$S \neq \{\}$ |] ==> $G \in$ *interval r a b*
⟨*proof*⟩

lemma (**in** *CLF*) *intervalPO*:
[| $a \in A$; $b \in A$; *interval r a b* $\neq \{\}$ |]
==> (| *pset* = *interval r a b*, *order* = *induced* (*interval r a b*) *r* |)
$\in$ *PartialOrder*
⟨*proof*⟩

lemma (**in** *CLF*) *intv-CL-lub*:
[| $a \in A$; $b \in A$; *interval r a b* $\neq \{\}$ |]
==> $\forall S.\ S \subseteq$ *interval r a b* −−>
($\exists L.$ *isLub S* (| *pset* = *interval r a b*,
order = *induced* (*interval r a b*) *r* |) *L*)
⟨*proof*⟩

lemmas (**in** *CLF*) *intv-CL-glb* = *intv-CL-lub* [*THEN Rdual*]

lemma (**in** *CLF*) *interval-is-sublattice*:
[| $a \in A$; $b \in A$; *interval r a b* $\neq \{\}$ |]
==> *interval r a b* <<= *cl*
⟨*proof*⟩

lemmas (**in** *CLF*) *interv-is-compl-latt* =
interval-is-sublattice [*THEN sublattice-imp-CL*]

38.9 Top and Bottom

lemma (**in** *CLF*) *Top-dual-Bot*: *Top cl* = *Bot* (*dual cl*)
⟨*proof*⟩

lemma (**in** *CLF*) *Bot-dual-Top*: *Bot cl* = *Top* (*dual cl*)
⟨*proof*⟩

lemma (**in** *CLF*) *Bot-in-lattice*: *Bot cl* ∈ *A*
⟨*proof*⟩

lemma (**in** *CLF*) *Top-in-lattice*: *Top cl* ∈ *A*
⟨*proof*⟩

lemma (**in** *CLF*) *Top-prop*: *x* ∈ *A* ==> (*x*, *Top cl*) ∈ *r*
⟨*proof*⟩

lemma (**in** *CLF*) *Bot-prop*: *x* ∈ *A* ==> (*Bot cl*, *x*) ∈ *r*
⟨*proof*⟩

lemma (**in** *CLF*) *Top-intv-not-empty*: *x* ∈ *A* ==> *interval r x* (*Top cl*) ≠ {}
⟨*proof*⟩

lemma (**in** *CLF*) *Bot-intv-not-empty*: *x* ∈ *A* ==> *interval r* (*Bot cl*) *x* ≠ {}
⟨*proof*⟩

38.10 fixed points form a partial order

lemma (**in** *CLF*) *fixf-po*: (| *pset* = *P*, *order* = *induced P r*|) ∈ *PartialOrder*
⟨*proof*⟩

lemma (**in** *Tarski*) *Y-subset-A*: *Y* ⊆ *A*
⟨*proof*⟩

lemma (**in** *Tarski*) *lubY-in-A*: *lub Y cl* ∈ *A*
⟨*proof*⟩

lemma (**in** *Tarski*) *lubY-le-flubY*: (*lub Y cl*, *f* (*lub Y cl*)) ∈ *r*
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-subset*: *intY1* ⊆ *A*
⟨*proof*⟩

lemmas (**in** *Tarski*) *intY1-elem* = *intY1-subset* [*THEN subsetD*]

lemma (**in** *Tarski*) *intY1-f-closed*: *x* ∈ *intY1* ⟹ *f x* ∈ *intY1*
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-mono*:
monotone (%*x*: *intY1*. *f x*) *intY1* (*induced intY1 r*)
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-is-cl*:
(| *pset* = *intY1*, *order* = *induced intY1 r* |) ∈ *CompleteLattice*
⟨*proof*⟩

lemma (**in** *Tarski*) *v-in-P*: *v* ∈ *P*
⟨*proof*⟩

lemma (**in** *Tarski*) *z-in-interval*:
[| *z* ∈ *P*; ∀ *y*∈*Y*. (*y*, *z*) ∈ *induced P r* |] ==> *z* ∈ *intY1*
⟨*proof*⟩

lemma (**in** *Tarski*) *f'z-in-int-rel*: [| *z* ∈ *P*; ∀ *y*∈*Y*. (*y*, *z*) ∈ *induced P r* |]
==> ((%*x*: *intY1*. *f x*) *z*, *z*) ∈ *induced intY1 r*
⟨*proof*⟩

lemma (**in** *Tarski*) *tarski-full-lemma*:
∃ *L*. *isLub Y* (| *pset* = *P*, *order* = *induced P r* |) *L*
⟨*proof*⟩

lemma *CompleteLatticeI-simp*:
[| (| *pset* = *A*, *order* = *r* |) ∈ *PartialOrder*;
∀ *S*. *S* ⊆ *A* −−> (∃ *L*. *isLub S* (| *pset* = *A*, *order* = *r* |) *L*) |]
==> (| *pset* = *A*, *order* = *r* |) ∈ *CompleteLattice*
⟨*proof*⟩

theorem (**in** *CLF*) *Tarski-full*:
(| *pset* = *P*, *order* = *induced P r*|) ∈ *CompleteLattice*
⟨*proof*⟩

end

39 Classical Predicate Calculus Problems

theory *Classical* **imports** *Main* **begin**

39.1 Traditional Classical Reasoner

The machine "griffon" mentioned below is a 2.5GHz Power Mac G5.

Taken from *FOL/Classical.thy*. When porting examples from first-order logic, beware of the precedence of = versus ↔.

lemma (*P* −−> *Q* | *R*) −−> (*P*−−>*Q*) | (*P*−−>*R*)
⟨*proof*⟩

If and only if

lemma $(P{=}Q) = (Q = (P{::}bool))$
⟨*proof*⟩

lemma $\sim (P = (\sim P))$
⟨*proof*⟩

Sample problems from F. J. Pelletier, Seventy-Five Problems for Testing Automatic Theorem Provers, J. Automated Reasoning 2 (1986), 191-216. Errata, JAR 4 (1988), 236-236.

The hardest problems – judging by experience with several theorem provers, including matrix ones – are 34 and 43.

39.1.1 Pelletier's examples

1

lemma $(P{-\!-}{>}Q) \;=\; (\sim Q \;{-\!-}{>}\; \sim P)$
⟨*proof*⟩

2

lemma $(\sim\, \sim\, P) =\; P$
⟨*proof*⟩

3

lemma $\sim(P{-\!-}{>}Q) \;{-\!-}{>}\; (Q{-\!-}{>}P)$
⟨*proof*⟩

4

lemma $(\sim P{-\!-}{>}Q) \;=\; (\sim Q \;{-\!-}{>}\; P)$
⟨*proof*⟩

5

lemma $((P|Q){-\!-}{>}(P|R)) \;{-\!-}{>}\; (P|(Q{-\!-}{>}R))$
⟨*proof*⟩

6

lemma $P \mid \sim P$
⟨*proof*⟩

7

lemma $P \mid \sim \sim \sim P$
⟨*proof*⟩

8. Peirce's law

lemma $((P{-\!-}{>}Q) \;{-\!-}{>}\; P) \;{-\!-}{>}\; P$

⟨*proof*⟩

9

lemma ((*P*|*Q*) & (~*P*|*Q*) & (*P*| ~*Q*)) −−> ~ (~*P* | ~*Q*)
⟨*proof*⟩

10

lemma (*Q*−−>*R*) & (*R*−−>*P*&*Q*) & (*P*−−>*Q*|*R*) −−> (*P*=*Q*)
⟨*proof*⟩

11. Proved in each direction (incorrectly, says Pelletier!!)

lemma *P*=(*P*::*bool*)
⟨*proof*⟩

12. "Dijkstra's law"

lemma ((*P* = *Q*) = *R*) = (*P* = (*Q* = *R*))
⟨*proof*⟩

13. Distributive law

lemma (*P* | (*Q* & *R*)) = ((*P* | *Q*) & (*P* | *R*))
⟨*proof*⟩

14

lemma (*P* = *Q*) = ((*Q* | ~*P*) & (~*Q*|*P*))
⟨*proof*⟩

15

lemma (*P* −−> *Q*) = (~*P* | *Q*)
⟨*proof*⟩

16

lemma (*P*−−>*Q*) | (*Q*−−>*P*)
⟨*proof*⟩

17

lemma ((*P* & (*Q*−−>*R*))−−>*S*) = ((~*P* | *Q* | *S*) & (~*P* | ~*R* | *S*))
⟨*proof*⟩

39.1.2 Classical Logic: examples with quantifiers

lemma (∀ *x*. *P*(*x*) & *Q*(*x*)) = ((∀ *x*. *P*(*x*)) & (∀ *x*. *Q*(*x*)))
⟨*proof*⟩

lemma (∃ *x*. *P*−−>*Q*(*x*)) = (*P* −−> (∃ *x*. *Q*(*x*)))
⟨*proof*⟩

lemma (∃ *x*. *P*(*x*)−−>*Q*) = ((∀ *x*. *P*(*x*)) −−> *Q*)

⟨*proof*⟩

lemma $((\forall x.\ P(x)) \mid Q) \ = \ (\forall x.\ P(x) \mid Q)$
⟨*proof*⟩

From Wishnu Prasetya

lemma $(\forall s.\ q(s) \dashrightarrow r(s)) \ \& \ \sim r(s) \ \& \ (\forall s.\ \sim r(s) \ \& \ \sim q(s) \dashrightarrow p(t) \mid q(t)) \dashrightarrow p(t) \mid r(t)$
⟨*proof*⟩

39.1.3 Problems requiring quantifier duplication

Theorem B of Peter Andrews, Theorem Proving via General Matings, JACM 28 (1981).

lemma $(\exists x.\ \forall y.\ P(x) = P(y)) \dashrightarrow ((\exists x.\ P(x)) = (\forall y.\ P(y)))$
⟨*proof*⟩

Needs multiple instantiation of the quantifier.

lemma $(\forall x.\ P(x) \dashrightarrow P(f(x))) \ \ \& \ \ P(d) \dashrightarrow P(f(f(f(d))))$
⟨*proof*⟩

Needs double instantiation of the quantifier

lemma $\exists x.\ P(x) \dashrightarrow P(a) \ \& \ P(b)$
⟨*proof*⟩

lemma $\exists z.\ P(z) \dashrightarrow (\forall x.\ P(x))$
⟨*proof*⟩

lemma $\exists x.\ (\exists y.\ P(y)) \dashrightarrow P(x)$
⟨*proof*⟩

39.1.4 Hard examples with quantifiers

Problem 18

lemma $\exists y.\ \forall x.\ P(y) \dashrightarrow P(x)$
⟨*proof*⟩

Problem 19

lemma $\exists x.\ \forall y\ z.\ (P(y) \dashrightarrow Q(z)) \dashrightarrow (P(x) \dashrightarrow Q(x))$
⟨*proof*⟩

Problem 20

lemma $(\forall x\ y.\ \exists z.\ \forall w.\ (P(x) \& Q(y) \dashrightarrow R(z) \& S(w))) \dashrightarrow (\exists x\ y.\ P(x) \ \& \ Q(y)) \dashrightarrow (\exists z.\ R(z))$
⟨*proof*⟩

Problem 21

lemma $(\exists x.\ P{-->}Q(x))\ \&\ (\exists x.\ Q(x){-->}P)\ {-->}\ (\exists x.\ P{=}Q(x))$
⟨*proof*⟩

Problem 22

lemma $(\forall x.\ P = Q(x))\ \ {-->}\ \ (P = (\forall x.\ Q(x)))$
⟨*proof*⟩

Problem 23

lemma $(\forall x.\ P \mid Q(x))\ \ =\ \ (P \mid (\forall x.\ Q(x)))$
⟨*proof*⟩

Problem 24

lemma $\sim(\exists x.\ S(x)\&Q(x))\ \&\ (\forall x.\ P(x)\ {-->}\ Q(x)|R(x))\ \&$
$\quad(\sim(\exists x.\ P(x))\ {-->}\ (\exists x.\ Q(x)))\ \&\ (\forall x.\ Q(x)|R(x)\ {-->}\ S(x))$
$\quad{-->}\ (\exists x.\ P(x)\&R(x))$
⟨*proof*⟩

Problem 25

lemma $(\exists x.\ P(x))\ \&$
$\qquad(\forall x.\ L(x)\ {-->}\ \sim\ (M(x)\ \&\ R(x)))\ \&$
$\qquad(\forall x.\ P(x)\ {-->}\ (M(x)\ \&\ L(x)))\ \&$
$\qquad((\forall x.\ P(x){-->}Q(x))\ \mid\ (\exists x.\ P(x)\&R(x)))$
$\quad{-->}\ (\exists x.\ Q(x)\&P(x))$
⟨*proof*⟩

Problem 26

lemma $((\exists x.\ p(x)) = (\exists x.\ q(x)))\ \&$
$\quad(\forall x.\ \forall y.\ p(x)\ \&\ q(y)\ {-->}\ (r(x) = s(y)))$
$\ {-->}\ ((\forall x.\ p(x){-->}r(x)) = (\forall x.\ q(x){-->}s(x)))$
⟨*proof*⟩

Problem 27

lemma $(\exists x.\ P(x)\ \&\ \sim Q(x))\ \&$
$\qquad(\forall x.\ P(x)\ {-->}\ R(x))\ \&$
$\qquad(\forall x.\ M(x)\ \&\ L(x)\ {-->}\ P(x))\ \&$
$\qquad((\exists x.\ R(x)\ \&\ \sim\ Q(x))\ {-->}\ (\forall x.\ L(x)\ {-->}\ \sim\ R(x)))$
$\quad{-->}\ (\forall x.\ M(x)\ {-->}\ \sim L(x))$
⟨*proof*⟩

Problem 28. AMENDED

lemma $(\forall x.\ P(x)\ {-->}\ (\forall x.\ Q(x)))\ \&$
$\qquad((\forall x.\ Q(x)|R(x))\ {-->}\ (\exists x.\ Q(x)\&S(x)))\ \&$
$\qquad((\exists x.\ S(x))\ {-->}\ (\forall x.\ L(x)\ {-->}\ M(x)))$
$\quad{-->}\ (\forall x.\ P(x)\ \&\ L(x)\ {-->}\ M(x))$
⟨*proof*⟩

Problem 29. Essentially the same as Principia Mathematica *11.71

lemma $(\exists x.\ F(x))\ \&\ (\exists y.\ G(y))$
$\quad --> (\ ((\forall x.\ F(x) --> H(x))\ \&\ (\forall y.\ G(y) --> J(y)))\ =$
$\qquad (\forall x\ y.\ F(x)\ \&\ G(y)\ --> H(x)\ \&\ J(y)))$
⟨*proof*⟩

Problem 30

lemma $(\forall x.\ P(x) \mid Q(x)\ --> {\sim}\ R(x))\ \&$
$\qquad (\forall x.\ (Q(x)\ --> {\sim}\ S(x))\ --> P(x)\ \&\ R(x))$
$\quad --> (\forall x.\ S(x))$
⟨*proof*⟩

Problem 31

lemma ${\sim}(\exists x.\ P(x)\ \&\ (Q(x) \mid R(x)))\ \&$
$\qquad (\exists x.\ L(x)\ \&\ P(x))\ \&$
$\qquad (\forall x.\ {\sim}\ R(x)\ --> M(x))$
$\quad --> (\exists x.\ L(x)\ \&\ M(x))$
⟨*proof*⟩

Problem 32

lemma $(\forall x.\ P(x)\ \&\ (Q(x)|R(x)) --> S(x))\ \&$
$\qquad (\forall x.\ S(x)\ \&\ R(x)\ --> L(x))\ \&$
$\qquad (\forall x.\ M(x)\ --> R(x))$
$\quad --> (\forall x.\ P(x)\ \&\ M(x)\ --> L(x))$
⟨*proof*⟩

Problem 33

lemma $(\forall x.\ P(a)\ \&\ (P(x) --> P(b)) --> P(c))\ =$
$\quad (\forall x.\ ({\sim}P(a) \mid P(x) \mid P(c))\ \&\ ({\sim}P(a) \mid {\sim}P(b) \mid P(c)))$
⟨*proof*⟩

Problem 34 AMENDED (TWICE!!)

Andrews's challenge

lemma $((\exists x.\ \forall y.\ p(x) = p(y))\ =$
$\qquad ((\exists x.\ q(x)) = (\forall y.\ p(y))))\ =$
$\qquad ((\exists x.\ \forall y.\ q(x) = q(y))\ =$
$\qquad ((\exists x.\ p(x)) = (\forall y.\ q(y))))$
⟨*proof*⟩

Problem 35

lemma $\exists x\ y.\ P\ x\ y\ -->\ (\forall u\ v.\ P\ u\ v)$
⟨*proof*⟩

Problem 36

lemma $(\forall x.\ \exists y.\ J\ x\ y)\ \&$
$\qquad (\forall x.\ \exists y.\ G\ x\ y)\ \&$
$\qquad (\forall x\ y.\ J\ x\ y \mid G\ x\ y\ -->$
$\qquad (\forall z.\ J\ y\ z \mid G\ y\ z\ --> H\ x\ z))$

$$--> (\forall x.\ \exists y.\ H\ x\ y)$$
⟨*proof*⟩

Problem 37

lemma $(\forall z.\ \exists w.\ \forall x.\ \exists y.$
$(P\ x\ z\ -->P\ y\ w)\ \&\ P\ y\ z\ \&\ (P\ y\ w\ -->\ (\exists u.\ Q\ u\ w)))\ \&$
$(\forall x\ z.\ \sim(P\ x\ z)\ -->\ (\exists y.\ Q\ y\ z))\ \&$
$((\exists x\ y.\ Q\ x\ y)\ -->\ (\forall x.\ R\ x\ x))$
$--> (\forall x.\ \exists y.\ R\ x\ y)$
⟨*proof*⟩

Problem 38

lemma $(\forall x.\ p(a)\ \&\ (p(x)\ -->\ (\exists y.\ p(y)\ \&\ r\ x\ y))\ -->$
$(\exists z.\ \exists w.\ p(z)\ \&\ r\ x\ w\ \&\ r\ w\ z))\ =$
$(\forall x.\ (\sim p(a)\ |\ p(x)\ |\ (\exists z.\ \exists w.\ p(z)\ \&\ r\ x\ w\ \&\ r\ w\ z))\ \&$
$(\sim p(a)\ |\ \sim(\exists y.\ p(y)\ \&\ r\ x\ y)\ |$
$(\exists z.\ \exists w.\ p(z)\ \&\ r\ x\ w\ \&\ r\ w\ z)))$
⟨*proof*⟩

Problem 39

lemma $\sim\ (\exists x.\ \forall y.\ F\ y\ x = (\sim\ F\ y\ y))$
⟨*proof*⟩

Problem 40. AMENDED

lemma $(\exists y.\ \forall x.\ F\ x\ y = F\ x\ x)$
$-->\ \sim\ (\forall x.\ \exists y.\ \forall z.\ F\ z\ y = (\sim\ F\ z\ x))$
⟨*proof*⟩

Problem 41

lemma $(\forall z.\ \exists y.\ \forall x.\ f\ x\ y = (f\ x\ z\ \&\ \sim\ f\ x\ x))$
$-->\ \sim\ (\exists z.\ \forall x.\ f\ x\ z)$
⟨*proof*⟩

Problem 42

lemma $\sim\ (\exists y.\ \forall x.\ p\ x\ y = (\sim\ (\exists z.\ p\ x\ z\ \&\ p\ z\ x)))$
⟨*proof*⟩

Problem 43!!

lemma $(\forall x::'a.\ \forall y::'a.\ q\ x\ y = (\forall z.\ p\ z\ x = (p\ z\ y::bool)))$
$-->\ (\forall x.\ (\forall y.\ q\ x\ y = (q\ y\ x::bool)))$
⟨*proof*⟩

Problem 44

lemma $(\forall x.\ f(x)\ -->$
$(\exists y.\ g(y)\ \&\ h\ x\ y\ \&\ (\exists y.\ g(y)\ \&\ \sim\ h\ x\ y)))\ \&$
$(\exists x.\ j(x)\ \&\ (\forall y.\ g(y)\ -->\ h\ x\ y))$
$-->\ (\exists x.\ j(x)\ \&\ \sim f(x))$

⟨*proof*⟩

Problem 45

lemma $(\forall x.\ f(x)\ \&\ (\forall y.\ g(y)\ \&\ h\ x\ y\ \text{--->}\ j\ x\ y)$
$\text{--->}\ (\forall y.\ g(y)\ \&\ h\ x\ y\ \text{--->}\ k(y)))\ \&$
$\sim\ (\exists y.\ l(y)\ \&\ k(y))\ \&$
$(\exists x.\ f(x)\ \&\ (\forall y.\ h\ x\ y\ \text{--->}\ l(y))$
$\&\ (\forall y.\ g(y)\ \&\ h\ x\ y\ \text{--->}\ j\ x\ y))$
$\text{--->}\ (\exists x.\ f(x)\ \&\ \sim\ (\exists y.\ g(y)\ \&\ h\ x\ y))$
⟨*proof*⟩

39.1.5 Problems (mainly) involving equality or functions

Problem 48

lemma $(a{=}b\ |\ c{=}d)\ \&\ (a{=}c\ |\ b{=}d)\ \text{--->}\ a{=}d\ |\ b{=}c$
⟨*proof*⟩

Problem 49 NOT PROVED AUTOMATICALLY. Hard because it involves substitution for Vars the type constraint ensures that x,y,z have the same type as a,b,u.

lemma $(\exists x\ y{::}'a.\ \forall z.\ z{=}x\ |\ z{=}y)\ \&\ P(a)\ \&\ P(b)\ \&\ (\sim a{=}b)$
$\text{--->}\ (\forall u{::}'a.\ P(u))$
⟨*proof*⟩

Problem 50. (What has this to do with equality?)

lemma $(\forall x.\ P\ a\ x\ |\ (\forall y.\ P\ x\ y))\ \text{--->}\ (\exists x.\ \forall y.\ P\ x\ y)$
⟨*proof*⟩

Problem 51

lemma $(\exists z\ w.\ \forall x\ y.\ P\ x\ y = (x{=}z\ \&\ y{=}w))\ \text{--->}$
$(\exists z.\ \forall x.\ \exists w.\ (\forall y.\ P\ x\ y = (y{=}w)) = (x{=}z))$
⟨*proof*⟩

Problem 52. Almost the same as 51.

lemma $(\exists z\ w.\ \forall x\ y.\ P\ x\ y = (x{=}z\ \&\ y{=}w))\ \text{--->}$
$(\exists w.\ \forall y.\ \exists z.\ (\forall x.\ P\ x\ y = (x{=}z)) = (y{=}w))$
⟨*proof*⟩

Problem 55

Non-equational version, from Manthey and Bry, CADE-9 (Springer, 1988). fast DISCOVERS who killed Agatha.

schematic-goal $lives(agatha)\ \&\ lives(butler)\ \&\ lives(charles)\ \&$
$(killed\ agatha\ agatha\ |\ killed\ butler\ agatha\ |\ killed\ charles\ agatha)\ \&$
$(\forall x\ y.\ killed\ x\ y\ \text{--->}\ hates\ x\ y\ \&\ \sim richer\ x\ y)\ \&$
$(\forall x.\ hates\ agatha\ x\ \text{--->}\ \sim hates\ charles\ x)\ \&$
$(hates\ agatha\ agatha\ \&\ hates\ agatha\ charles)\ \&$

$(\forall x.\ lives(x)\ \&\ \sim richer\ x\ agatha\ \longrightarrow\ hates\ butler\ x)\ \&$
$(\forall x.\ hates\ agatha\ x\ \longrightarrow\ hates\ butler\ x)\ \&$
$(\forall x.\ \sim hates\ x\ agatha\ |\ \sim hates\ x\ butler\ |\ \sim hates\ x\ charles)\ \longrightarrow$
$killed\ ?who\ agatha$
⟨*proof*⟩

Problem 56

lemma $(\forall x.\ (\exists y.\ P(y)\ \&\ x{=}f(y))\ \longrightarrow\ P(x)) = (\forall x.\ P(x)\ \longrightarrow\ P(f(x)))$
⟨*proof*⟩

Problem 57

lemma $P\ (f\ a\ b)\ (f\ b\ c)\ \&\ P\ (f\ b\ c)\ (f\ a\ c)\ \&$
$(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z\ \longrightarrow\ P\ x\ z)\ \longrightarrow\ P\ (f\ a\ b)\ (f\ a\ c)$
⟨*proof*⟩

Problem 58 NOT PROVED AUTOMATICALLY

lemma $(\forall x\ y.\ f(x){=}g(y))\ \longrightarrow\ (\forall x\ y.\ f(f(x)){=}f(g(y)))$
⟨*proof*⟩

Problem 59

lemma $(\forall x.\ P(x) = (\sim P(f(x))))\ \longrightarrow\ (\exists x.\ P(x)\ \&\ \sim P(f(x)))$
⟨*proof*⟩

Problem 60

lemma $\forall x.\ P\ x\ (f\ x) = (\exists y.\ (\forall z.\ P\ z\ y\ \longrightarrow\ P\ z\ (f\ x))\ \&\ P\ x\ y)$
⟨*proof*⟩

Problem 62 as corrected in JAR 18 (1997), page 135

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ \longrightarrow\ p(f\ x))\ \longrightarrow\ p(f(f\ x)))\ =$
$(\forall x.\ (\sim\ p\ a\ |\ p\ x\ |\ p(f(f\ x)))\ \&$
$(\sim\ p\ a\ |\ \sim\ p(f\ x)\ |\ p(f(f\ x))))$
⟨*proof*⟩

From Davis, Obvious Logical Inferences, IJCAI-81, 530-531 fast indeed copes!

lemma $(\forall x.\ F(x)\ \&\ \sim G(x)\ \longrightarrow\ (\exists y.\ H(x,y)\ \&\ J(y)))\ \&$
$(\exists x.\ K(x)\ \&\ F(x)\ \&\ (\forall y.\ H(x,y)\ \longrightarrow\ K(y)))\ \&$
$(\forall x.\ K(x)\ \longrightarrow\ \sim G(x))\ \longrightarrow\ (\exists x.\ K(x)\ \&\ J(x))$
⟨*proof*⟩

From Rudnicki, Obvious Inferences, JAR 3 (1987), 383-393. It does seem obvious!

lemma $(\forall x.\ F(x)\ \&\ \sim G(x)\ \longrightarrow\ (\exists y.\ H(x,y)\ \&\ J(y)))\ \&$
$(\exists x.\ K(x)\ \&\ F(x)\ \&\ (\forall y.\ H(x,y)\ \longrightarrow\ K(y)))\ \&$
$(\forall x.\ K(x)\ \longrightarrow\ \sim G(x))\ \longrightarrow\ (\exists x.\ K(x)\ \longrightarrow\ \sim G(x))$
⟨*proof*⟩

Attributed to Lewis Carroll by S. G. Pulman. The first or last assumption can be deleted.

lemma $(\forall x.\ honest(x)\ \&\ industrious(x) \longrightarrow healthy(x))\ \&$
$\sim (\exists x.\ grocer(x)\ \&\ healthy(x))\ \&$
$(\forall x.\ industrious(x)\ \&\ grocer(x) \longrightarrow honest(x))\ \&$
$(\forall x.\ cyclist(x) \longrightarrow industrious(x))\ \&$
$(\forall x.\ \sim healthy(x)\ \&\ cyclist(x) \longrightarrow \sim honest(x))$
$\longrightarrow (\forall x.\ grocer(x) \longrightarrow \sim cyclist(x))$
⟨*proof*⟩

lemma $(\forall x\ y.\ R(x,y) \mid R(y,x))\ \&$
$(\forall x\ y.\ S(x,y)\ \&\ S(y,x) \longrightarrow x=y)\ \&$
$(\forall x\ y.\ R(x,y) \longrightarrow S(x,y)) \quad \longrightarrow \quad (\forall x\ y.\ S(x,y) \longrightarrow R(x,y))$
⟨*proof*⟩

39.2 Model Elimination Prover

Trying out meson with arguments

lemma $x < y\ \&\ y < z \longrightarrow \sim (z < (x::nat))$
⟨*proof*⟩

The "small example" from Bezem, Hendriks and de Nivelle, Automatic Proof Construction in Type Theory Using Resolution, JAR 29: 3-4 (2002), pages 253-275

lemma $(\forall x\ y\ z.\ R(x,y)\ \&\ R(y,z) \longrightarrow R(x,z))\ \&$
$(\forall x.\ \exists y.\ R(x,y)) \longrightarrow$
$\sim (\forall x.\ P\ x = (\forall y.\ R(x,y) \longrightarrow \sim\ P\ y))$
⟨*proof*⟩

39.2.1 Pelletier's examples

1

lemma $(P \longrightarrow Q)\ =\ (\sim Q \longrightarrow \sim P)$
⟨*proof*⟩

2

lemma $(\sim\ \sim\ P) =\ P$
⟨*proof*⟩

3

lemma $\sim(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P)$
⟨*proof*⟩

4

lemma $(\sim P \longrightarrow Q)\ =\ (\sim Q \longrightarrow P)$
⟨*proof*⟩

5

lemma $((P|Q) \longrightarrow (P|R)) \longrightarrow (P|(Q \longrightarrow R))$

⟨*proof*⟩

6

lemma $P \mid \sim P$
⟨*proof*⟩

7

lemma $P \mid \sim \sim \sim P$
⟨*proof*⟩

8. Peirce's law

lemma $((P-->Q) \dashrightarrow P) \dashrightarrow P$
⟨*proof*⟩

9

lemma $((P|Q) \,\&\, (\sim P|Q) \,\&\, (P|\sim Q)) \dashrightarrow \sim (\sim P \mid \sim Q)$
⟨*proof*⟩

10

lemma $(Q-->R) \,\&\, (R-->P\&Q) \,\&\, (P-->Q|R) \dashrightarrow (P=Q)$
⟨*proof*⟩

11. Proved in each direction (incorrectly, says Pelletier!!)

lemma $P=(P::bool)$
⟨*proof*⟩

12. "Dijkstra's law"

lemma $((P = Q) = R) = (P = (Q = R))$
⟨*proof*⟩

13. Distributive law

lemma $(P \mid (Q \,\&\, R)) = ((P \mid Q) \,\&\, (P \mid R))$
⟨*proof*⟩

14

lemma $(P = Q) = ((Q \mid \sim P) \,\&\, (\sim Q|P))$
⟨*proof*⟩

15

lemma $(P \dashrightarrow Q) = (\sim P \mid Q)$
⟨*proof*⟩

16

lemma $(P-->Q) \mid (Q-->P)$
⟨*proof*⟩

17

lemma $((P \,\&\, (Q-->R))-->S) \;=\; ((\sim P \mid Q \mid S) \,\&\, (\sim P \mid \sim R \mid S))$
⟨*proof*⟩

39.2.2 Classical Logic: examples with quantifiers

lemma $(\forall x.\ P\ x\ \&\ Q\ x) = ((\forall x.\ P\ x)\ \&\ (\forall x.\ Q\ x))$
⟨*proof*⟩

lemma $(\exists x.\ P\ \longrightarrow\ Q\ x)\ =\ (P\ \longrightarrow\ (\exists x.\ Q\ x))$
⟨*proof*⟩

lemma $(\exists x.\ P\ x\ \longrightarrow\ Q) = ((\forall x.\ P\ x)\ \longrightarrow\ Q)$
⟨*proof*⟩

lemma $((\forall x.\ P\ x)\ |\ Q)\ =\ (\forall x.\ P\ x\ |\ Q)$
⟨*proof*⟩

lemma $(\forall x.\ P\ x\ \longrightarrow\ P(f\ x))\ \ \&\ \ P\ d\ \longrightarrow\ P(f(f(f\ d)))$
⟨*proof*⟩

Needs double instantiation of EXISTS

lemma $\exists x.\ P\ x\ \longrightarrow\ P\ a\ \&\ P\ b$
⟨*proof*⟩

lemma $\exists z.\ P\ z\ \longrightarrow\ (\forall x.\ P\ x)$
⟨*proof*⟩

From a paper by Claire Quigley

lemma $\exists y.\ ((P\ c\ \&\ Q\ y)\ |\ (\exists z.\ \sim\ Q\ z))\ |\ (\exists x.\ \sim\ P\ x\ \&\ Q\ d)$
⟨*proof*⟩

39.2.3 Hard examples with quantifiers

Problem 18

lemma $\exists y.\ \forall x.\ P\ y\ \longrightarrow\ P\ x$
⟨*proof*⟩

Problem 19

lemma $\exists x.\ \forall y\ z.\ (P\ y\ \longrightarrow\ Q\ z)\ \longrightarrow\ (P\ x\ \longrightarrow\ Q\ x)$
⟨*proof*⟩

Problem 20

lemma $(\forall x\ y.\ \exists z.\ \forall w.\ (P\ x\ \&\ Q\ y\ \longrightarrow\ R\ z\ \&\ S\ w))$
$\quad\longrightarrow\ (\exists x\ y.\ P\ x\ \&\ Q\ y)\ \longrightarrow\ (\exists z.\ R\ z)$
⟨*proof*⟩

Problem 21

lemma $(\exists x.\ P\ \longrightarrow\ Q\ x)\ \&\ (\exists x.\ Q\ x\ \longrightarrow\ P)\ \longrightarrow\ (\exists x.\ P{=}Q\ x)$
⟨*proof*⟩

Problem 22

lemma $(\forall x.\ P = Q\ x)\ \ --> \ \ (P = (\forall x.\ Q\ x))$
⟨*proof*⟩

Problem 23

lemma $(\forall x.\ P \mid Q\ x)\ \ =\ \ (P \mid (\forall x.\ Q\ x))$
⟨*proof*⟩

Problem 24

lemma $\sim(\exists x.\ S\ x\ \&\ Q\ x)\ \&\ (\forall x.\ P\ x\ --> Q\ x \mid R\ x)\ \&$
$\quad (\sim(\exists x.\ P\ x)\ --> (\exists x.\ Q\ x))\ \&\ (\forall x.\ Q\ x \mid R\ x\ --> S\ x)$
$\quad --> (\exists x.\ P\ x\ \&\ R\ x)$
⟨*proof*⟩

Problem 25

lemma $(\exists x.\ P\ x)\ \&$
$\quad (\forall x.\ L\ x\ --> \sim (M\ x\ \&\ R\ x))\ \&$
$\quad (\forall x.\ P\ x\ --> (M\ x\ \&\ L\ x))\ \&$
$\quad ((\forall x.\ P\ x\ --> Q\ x) \mid (\exists x.\ P\ x\ \&\ R\ x))$
$\quad --> (\exists x.\ Q\ x\ \&\ P\ x)$
⟨*proof*⟩

Problem 26; has 24 Horn clauses

lemma $((\exists x.\ p\ x) = (\exists x.\ q\ x))\ \&$
$\quad (\forall x.\ \forall y.\ p\ x\ \&\ q\ y\ --> (r\ x = s\ y))$
$\ --> ((\forall x.\ p\ x\ --> r\ x) = (\forall x.\ q\ x\ --> s\ x))$
⟨*proof*⟩

Problem 27; has 13 Horn clauses

lemma $(\exists x.\ P\ x\ \&\ \sim Q\ x)\ \&$
$\quad (\forall x.\ P\ x\ --> R\ x)\ \&$
$\quad (\forall x.\ M\ x\ \&\ L\ x\ --> P\ x)\ \&$
$\quad ((\exists x.\ R\ x\ \&\ \sim\ Q\ x)\ --> (\forall x.\ L\ x\ --> \sim\ R\ x))$
$\quad --> (\forall x.\ M\ x\ --> \sim L\ x)$
⟨*proof*⟩

Problem 28. AMENDED; has 14 Horn clauses

lemma $(\forall x.\ P\ x\ --> (\forall x.\ Q\ x))\ \&$
$\quad ((\forall x.\ Q\ x \mid R\ x)\ --> (\exists x.\ Q\ x\ \&\ S\ x))\ \&$
$\quad ((\exists x.\ S\ x)\ --> (\forall x.\ L\ x\ --> M\ x))$
$\quad --> (\forall x.\ P\ x\ \&\ L\ x\ --> M\ x)$
⟨*proof*⟩

Problem 29. Essentially the same as Principia Mathematica *11.71. 62 Horn clauses

lemma $(\exists x.\ F\ x)\ \&\ (\exists y.\ G\ y)$
$\quad --> (\ ((\forall x.\ F\ x\ --> H\ x)\ \&\ (\forall y.\ G\ y\ --> J\ y))\ \ =$
$\qquad (\forall x\ y.\ F\ x\ \&\ G\ y\ --> H\ x\ \&\ J\ y))$

⟨*proof*⟩

Problem 30

lemma $(\forall x.\ P\ x \mid Q\ x \longrightarrow {\sim} R\ x)\ \&\ (\forall x.\ (Q\ x \longrightarrow {\sim} S\ x) \longrightarrow P\ x\ \&\ R\ x)$
$\longrightarrow (\forall x.\ S\ x)$
⟨*proof*⟩

Problem 31; has 10 Horn clauses; first negative clauses is useless

lemma ${\sim}(\exists x.\ P\ x\ \&\ (Q\ x \mid R\ x))\ \&$
$(\exists x.\ L\ x\ \&\ P\ x)\ \&$
$(\forall x.\ {\sim} R\ x \longrightarrow M\ x)$
$\longrightarrow (\exists x.\ L\ x\ \&\ M\ x)$
⟨*proof*⟩

Problem 32

lemma $(\forall x.\ P\ x\ \&\ (Q\ x \mid R\ x) \longrightarrow S\ x)\ \&$
$(\forall x.\ S\ x\ \&\ R\ x \longrightarrow L\ x)\ \&$
$(\forall x.\ M\ x \longrightarrow R\ x)$
$\longrightarrow (\forall x.\ P\ x\ \&\ M\ x \longrightarrow L\ x)$
⟨*proof*⟩

Problem 33; has 55 Horn clauses

lemma $(\forall x.\ P\ a\ \&\ (P\ x \longrightarrow P\ b) \longrightarrow P\ c)\ =$
$(\forall x.\ ({\sim}P\ a \mid P\ x \mid P\ c)\ \&\ ({\sim}P\ a \mid {\sim}P\ b \mid P\ c))$
⟨*proof*⟩

Problem 34: Andrews's challenge has 924 Horn clauses

lemma $((\exists x.\ \forall y.\ p\ x = p\ y)\ = ((\exists x.\ q\ x) = (\forall y.\ p\ y)))\ \ =$
$((\exists x.\ \forall y.\ q\ x = q\ y)\ = ((\exists x.\ p\ x) = (\forall y.\ q\ y)))$
⟨*proof*⟩

Problem 35

lemma $\exists x\ y.\ P\ x\ y \longrightarrow\ (\forall u\ v.\ P\ u\ v)$
⟨*proof*⟩

Problem 36; has 15 Horn clauses

lemma $(\forall x.\ \exists y.\ J\ x\ y)\ \&\ (\forall x.\ \exists y.\ G\ x\ y)\ \&$
$(\forall x\ y.\ J\ x\ y \mid G\ x\ y \longrightarrow (\forall z.\ J\ y\ z \mid G\ y\ z \longrightarrow H\ x\ z))$
$\longrightarrow (\forall x.\ \exists y.\ H\ x\ y)$
⟨*proof*⟩

Problem 37; has 10 Horn clauses

lemma $(\forall z.\ \exists w.\ \forall x.\ \exists y.$
$(P\ x\ z \longrightarrow P\ y\ w)\ \&\ P\ y\ z\ \&\ (P\ y\ w \longrightarrow (\exists u.\ Q\ u\ w)))\ \&$
$(\forall x\ z.\ {\sim}P\ x\ z \longrightarrow (\exists y.\ Q\ y\ z))\ \&$
$((\exists x\ y.\ Q\ x\ y) \longrightarrow (\forall x.\ R\ x\ x))$
$\longrightarrow (\forall x.\ \exists y.\ R\ x\ y)$

⟨*proof*⟩

Problem 38

Quite hard: 422 Horn clauses!!

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ \text{--->}\ (\exists y.\ p\ y\ \&\ r\ x\ y))\ \text{--->}$
$\quad (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z))\ =$
$\quad (\forall x.\ (\sim p\ a \mid p\ x \mid (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z))\ \&$
$\quad\ (\sim p\ a \mid \sim(\exists y.\ p\ y\ \&\ r\ x\ y) \mid$
$\quad\ (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z)))$
⟨*proof*⟩

Problem 39

lemma $\sim\ (\exists x.\ \forall y.\ F\ y\ x = (\sim F\ y\ y))$
⟨*proof*⟩

Problem 40. AMENDED

lemma $(\exists y.\ \forall x.\ F\ x\ y = F\ x\ x)$
$\quad \text{--->}\ \sim\ (\forall x.\ \exists y.\ \forall z.\ F\ z\ y = (\sim F\ z\ x))$
⟨*proof*⟩

Problem 41

lemma $(\forall z.\ (\exists y.\ (\forall x.\ f\ x\ y = (f\ x\ z\ \&\ \sim\ f\ x\ x))))$
$\quad \text{--->}\ \sim\ (\exists z.\ \forall x.\ f\ x\ z)$
⟨*proof*⟩

Problem 42

lemma $\sim\ (\exists y.\ \forall x.\ p\ x\ y = (\sim\ (\exists z.\ p\ x\ z\ \&\ p\ z\ x)))$
⟨*proof*⟩

Problem 43 NOW PROVED AUTOMATICALLY!!

lemma $(\forall x.\ \forall y.\ q\ x\ y = (\forall z.\ p\ z\ x = (p\ z\ y{::}bool)))$
$\quad \text{--->}\ (\forall x.\ (\forall y.\ q\ x\ y = (q\ y\ x{::}bool)))$
⟨*proof*⟩

Problem 44: 13 Horn clauses; 7-step proof

lemma $(\forall x.\ f\ x\ \text{--->}\ (\exists y.\ g\ y\ \&\ h\ x\ y\ \&\ (\exists y.\ g\ y\ \&\ \sim\ h\ x\ y)))\ \ \&$
$\quad (\exists x.\ j\ x\ \&\ (\forall y.\ g\ y\ \text{--->}\ h\ x\ y))$
$\quad \text{--->}\ (\exists x.\ j\ x\ \&\ \sim f\ x)$
⟨*proof*⟩

Problem 45; has 27 Horn clauses; 54-step proof

lemma $(\forall x.\ f\ x\ \&\ (\forall y.\ g\ y\ \&\ h\ x\ y\ \text{--->}\ j\ x\ y)$
$\quad\ \text{--->}\ (\forall y.\ g\ y\ \&\ h\ x\ y\ \text{--->}\ k\ y))\ \&$
$\quad \sim\ (\exists y.\ l\ y\ \&\ k\ y)\ \&$
$\quad (\exists x.\ f\ x\ \&\ (\forall y.\ h\ x\ y\ \text{--->}\ l\ y)$
$\quad\ \&\ (\forall y.\ g\ y\ \&\ h\ x\ y\ \text{--->}\ j\ x\ y))$

$$--> (\exists\, x.\ f\ x\ \&\ \sim\ (\exists\, y.\ g\ y\ \&\ h\ x\ y))$$
⟨*proof*⟩

Problem 46; has 26 Horn clauses; 21-step proof

lemma $(\forall\, x.\ f\ x\ \&\ (\forall\, y.\ f\ y\ \&\ h\ y\ x\ -->\ g\ y)\ -->\ g\ x)\ \&$
$((\exists\, x.\ f\ x\ \&\ \sim g\ x)\ -->$
$(\exists\, x.\ f\ x\ \&\ \sim g\ x\ \&\ (\forall\, y.\ f\ y\ \&\ \sim g\ y\ -->\ j\ x\ y)))\ \&$
$(\forall\, x\ y.\ f\ x\ \&\ f\ y\ \&\ h\ x\ y\ -->\ \sim j\ y\ x)$
$-->\ (\forall\, x.\ f\ x\ -->\ g\ x)$
⟨*proof*⟩

Problem 47. Schubert's Steamroller. 26 clauses; 63 Horn clauses. 87094 inferences so far. Searching to depth 36

lemma $(\forall\, x.\ wolf\ x \longrightarrow animal\ x)\ \&\ (\exists\, x.\ wolf\ x)\ \&$
$(\forall\, x.\ fox\ x \longrightarrow animal\ x)\ \&\ (\exists\, x.\ fox\ x)\ \&$
$(\forall\, x.\ bird\ x \longrightarrow animal\ x)\ \&\ (\exists\, x.\ bird\ x)\ \&$
$(\forall\, x.\ caterpillar\ x \longrightarrow animal\ x)\ \&\ (\exists\, x.\ caterpillar\ x)\ \&$
$(\forall\, x.\ snail\ x \longrightarrow animal\ x)\ \&\ (\exists\, x.\ snail\ x)\ \&$
$(\forall\, x.\ grain\ x \longrightarrow plant\ x)\ \&\ (\exists\, x.\ grain\ x)\ \&$
$(\forall\, x.\ animal\ x \longrightarrow$
$((\forall\, y.\ plant\ y \longrightarrow eats\ x\ y)\ \vee$
$(\forall\, y.\ animal\ y\ \&\ smaller\text{-}than\ y\ x\ \&$
$(\exists\, z.\ plant\ z\ \&\ eats\ y\ z) \longrightarrow eats\ x\ y)))\ \&$
$(\forall\, x\ y.\ bird\ y\ \&\ (snail\ x \vee caterpillar\ x) \longrightarrow smaller\text{-}than\ x\ y)\ \&$
$(\forall\, x\ y.\ bird\ x\ \&\ fox\ y \longrightarrow smaller\text{-}than\ x\ y)\ \&$
$(\forall\, x\ y.\ fox\ x\ \&\ wolf\ y \longrightarrow smaller\text{-}than\ x\ y)\ \&$
$(\forall\, x\ y.\ wolf\ x\ \&\ (fox\ y \vee grain\ y) \longrightarrow \sim eats\ x\ y)\ \&$
$(\forall\, x\ y.\ bird\ x\ \&\ caterpillar\ y \longrightarrow eats\ x\ y)\ \&$
$(\forall\, x\ y.\ bird\ x\ \&\ snail\ y \longrightarrow \sim eats\ x\ y)\ \&$
$(\forall\, x.\ (caterpillar\ x \vee snail\ x) \longrightarrow (\exists\, y.\ plant\ y\ \&\ eats\ x\ y))$
$\longrightarrow (\exists\, x\ y.\ animal\ x\ \&\ animal\ y\ \&\ (\exists\, z.\ grain\ z\ \&\ eats\ y\ z\ \&\ eats\ x\ y))$
⟨*proof*⟩

The Los problem. Circulated by John Harrison

lemma $(\forall\, x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z\ -->\ P\ x\ z)\ \&$
$(\forall\, x\ y\ z.\ Q\ x\ y\ \&\ Q\ y\ z\ -->\ Q\ x\ z)\ \&$
$(\forall\, x\ y.\ P\ x\ y\ -->\ P\ y\ x)\ \&$
$(\forall\, x\ y.\ P\ x\ y\ |\ Q\ x\ y)$
$-->\ (\forall\, x\ y.\ P\ x\ y)\ |\ (\forall\, x\ y.\ Q\ x\ y)$
⟨*proof*⟩

A similar example, suggested by Johannes Schumann and credited to Pelletier

lemma $(\forall\, x\ y\ z.\ P\ x\ y\ -->\ P\ y\ z\ -->\ P\ x\ z)\ -->$
$(\forall\, x\ y\ z.\ Q\ x\ y\ -->\ Q\ y\ z\ -->\ Q\ x\ z)\ -->$
$(\forall\, x\ y.\ Q\ x\ y\ -->\ Q\ y\ x)\ -->\ (\forall\, x\ y.\ P\ x\ y\ |\ Q\ x\ y)\ -->$
$(\forall\, x\ y.\ P\ x\ y)\ |\ (\forall\, x\ y.\ Q\ x\ y)$
⟨*proof*⟩

Problem 50. What has this to do with equality?

lemma $(\forall x.\ P\ a\ x \mid (\forall y.\ P\ x\ y)) \longrightarrow (\exists x.\ \forall y.\ P\ x\ y)$
⟨*proof*⟩

Problem 54: NOT PROVED

lemma $(\forall y::'a.\ \exists z.\ \forall x.\ F\ x\ z = (x{=}y)) \longrightarrow$
$\sim (\exists w.\ \forall x.\ F\ x\ w = (\forall u.\ F\ x\ u \longrightarrow (\exists y.\ F\ y\ u\ \&\ \sim (\exists z.\ F\ z\ u\ \&\ F\ z\ y))))$
⟨*proof*⟩

Problem 55

Non-equational version, from Manthey and Bry, CADE-9 (Springer, 1988). *meson* cannot report who killed Agatha.

lemma *lives agatha & lives butler & lives charles &*
(killed agatha agatha | killed butler agatha | killed charles agatha) &
(∀ x y. killed x y −−> hates x y & ~richer x y) &
(∀ x. hates agatha x −−> ~hates charles x) &
(hates agatha agatha & hates agatha charles) &
(∀ x. lives x & ~richer x agatha −−> hates butler x) &
(∀ x. hates agatha x −−> hates butler x) &
(∀ x. ~hates x agatha | ~hates x butler | ~hates x charles) −−>
(∃ x. killed x agatha)
⟨*proof*⟩

Problem 57

lemma $P\ (f\ a\ b)\ (f\ b\ c)\ \&\ P\ (f\ b\ c)\ (f\ a\ c)\ \&$
$(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z \longrightarrow P\ x\ z) \longrightarrow P\ (f\ a\ b)\ (f\ a\ c)$
⟨*proof*⟩

Problem 58: Challenge found on info-hol

lemma $\forall P\ Q\ R\ x.\ \exists v\ w.\ \forall y\ z.\ P\ x\ \&\ Q\ y \longrightarrow (P\ v \mid R\ w)\ \&\ (R\ z \longrightarrow Q\ v)$
⟨*proof*⟩

Problem 59

lemma $(\forall x.\ P\ x = (\sim P(f\ x))) \longrightarrow (\exists x.\ P\ x\ \&\ \sim P(f\ x))$
⟨*proof*⟩

Problem 60

lemma $\forall x.\ P\ x\ (f\ x) = (\exists y.\ (\forall z.\ P\ z\ y \longrightarrow P\ z\ (f\ x))\ \&\ P\ x\ y)$
⟨*proof*⟩

Problem 62 as corrected in JAR 18 (1997), page 135

lemma $(\forall x.\ p\ a\ \&\ (p\ x \longrightarrow p(f\ x)) \longrightarrow p(f(f\ x))) =$
$(\forall x.\ (\sim p\ a \mid p\ x \mid p(f(f\ x)))\ \&$
$(\sim p\ a \mid \sim p(f\ x) \mid p(f(f\ x))))$
⟨*proof*⟩

* Charles Morgan's problems *

lemma
 assumes a: $\forall x\ y.\ \ T(i\ x(i\ y\ x))$
  and b: $\forall x\ y\ z.\ T(i\ (i\ x\ (i\ y\ z))\ (i\ (i\ x\ y)\ (i\ x\ z)))$
  and c: $\forall x\ y.\ \ T(i\ (i\ (n\ x)\ (n\ y))\ (i\ y\ x))$
  and c': $\forall x\ y.\ \ T(i\ (i\ y\ x)\ (i\ (n\ x)\ (n\ y)))$
  and d: $\forall x\ y.\ \ T(i\ x\ y)\ \&\ T\ x\ --{>}\ T\ y$
 shows *True*
⟨*proof*⟩

Problem 71, as found in TPTP (SYN007+1.005)

lemma $p1 = (p2 = (p3 = (p4 = (p5 = (p1 = (p2 = (p3 = (p4 = p5))))))))$
⟨*proof*⟩

end

40 Set Theory examples: Cantor's Theorem, Schröder-Bernstein Theorem, etc.

theory *Set-Theory*
imports *Main*
begin

These two are cited in Benzmueller and Kohlhase's system description of LEO, CADE-15, 1998 (pages 139-143) as theorems LEO could not prove.

lemma $(X = Y \cup Z) =$
  $(Y \subseteq X \wedge Z \subseteq X \wedge (\forall V.\ Y \subseteq V \wedge Z \subseteq V \longrightarrow X \subseteq V))$
 ⟨*proof*⟩

lemma $(X = Y \cap Z) =$
  $(X \subseteq Y \wedge X \subseteq Z \wedge (\forall V.\ V \subseteq Y \wedge V \subseteq Z \longrightarrow V \subseteq X))$
 ⟨*proof*⟩

Trivial example of term synthesis: apparently hard for some provers!

schematic-goal $a \neq b \Longrightarrow a \in ?X \wedge b \notin ?X$
 ⟨*proof*⟩

40.1 Examples for the *blast* paper

lemma $(\bigcup x \in C.\ f\ x \cup g\ x) = \bigcup(f\ `\ C)\ \cup\ \bigcup(g\ `\ C)$
 — Union-image, called *Un-Union-image* in Main HOL
 ⟨*proof*⟩

lemma $(\bigcap x \in C.\ f\ x \cap g\ x) = \bigcap(f\ `\ C) \cap \bigcap(g\ `\ C)$
 — Inter-image, called *Int-Inter-image* in Main HOL
 ⟨*proof*⟩

lemma *singleton-example-1*:
  $\bigwedge S::'a\ set\ set.\ \forall x \in S.\ \forall y \in S.\ x \subseteq y \Longrightarrow \exists z.\ S \subseteq \{z\}$

⟨*proof*⟩

lemma *singleton-example-2*:
$\forall x \in S.\ \bigcup S \subseteq x \Longrightarrow \exists z.\ S \subseteq \{z\}$
— Variant of the problem above.
⟨*proof*⟩

lemma $\exists!x.\ f\ (g\ x) = x \Longrightarrow \exists!y.\ g\ (f\ y) = y$
— A unique fixpoint theorem — *fast*/*best*/*meson* all fail.
⟨*proof*⟩

40.2 Cantor's Theorem: There is no surjection from a set to its powerset

lemma *cantor1*: $\neg\ (\exists f::\ 'a \Rightarrow 'a\ set.\ \forall S.\ \exists x.\ f\ x = S)$
— Requires best-first search because it is undirectional.
⟨*proof*⟩

schematic-goal $\forall f::\ 'a \Rightarrow 'a\ set.\ \forall x.\ f\ x \neq ?S\ f$
— This form displays the diagonal term.
⟨*proof*⟩

schematic-goal $?S \notin range\ (f :: 'a \Rightarrow 'a\ set)$
— This form exploits the set constructs.
⟨*proof*⟩

schematic-goal $?S \notin range\ (f :: 'a \Rightarrow 'a\ set)$
— Or just this!
⟨*proof*⟩

40.3 The Schröder-Bernstein Theorem

lemma *disj-lemma*: $-\ (f\ `\ X) = g'\ `\ (-X) \Longrightarrow f\ a = g'\ b \Longrightarrow a \in X \Longrightarrow b \in X$
⟨*proof*⟩

lemma *surj-if-then-else*:
$-(f\ `\ X) = g'\ `\ (-X) \Longrightarrow surj\ (\lambda z.\ if\ z \in X\ then\ f\ z\ else\ g'\ z)$
⟨*proof*⟩

lemma *bij-if-then-else*:
$inj\text{-}on\ f\ X \Longrightarrow inj\text{-}on\ g'\ (-X) \Longrightarrow -(f\ `\ X) = g'\ `\ (-X) \Longrightarrow$
$h = (\lambda z.\ if\ z \in X\ then\ f\ z\ else\ g'\ z) \Longrightarrow inj\ h \wedge surj\ h$
⟨*proof*⟩

lemma *decomposition*: $\exists X.\ X = -\ (g\ `\ (-\ (f\ `\ X)))$
⟨*proof*⟩

theorem *Schroeder-Bernstein*:
$inj\ (f :: 'a \Rightarrow 'b) \Longrightarrow inj\ (g :: 'b \Rightarrow 'a)$

$\Longrightarrow \exists h::'a \Rightarrow 'b.\ inj\ h \wedge surj\ h$
⟨*proof*⟩

40.4 A simple party theorem

At any party there are two people who know the same number of people. Provided the party consists of at least two people and the knows relation is symmetric. Knowing yourself does not count — otherwise knows needs to be reflexive. (From Freek Wiedijk's talk at TPHOLs 2007.)

lemma *equal-number-of-acquaintances*:
assumes *Domain R <= A* **and** *sym R* **and** *card A ≥ 2*
shows ¬ *inj-on* (%*a*. *card*(*R* `` {*a*} − {*a*})) *A*
⟨*proof*⟩

From W. W. Bledsoe and Guohui Feng, SET-VAR. JAR 11 (3), 1993, pages 293-314.

Isabelle can prove the easy examples without any special mechanisms, but it can't prove the hard ones.

lemma $\exists A.\ (\forall x \in A.\ x \leq (0::int))$
— Example 1, page 295.
⟨*proof*⟩

lemma $D \in F \Longrightarrow \exists G.\ \forall A \in G.\ \exists B \in F.\ A \subseteq B$
— Example 2.
⟨*proof*⟩

lemma $P\ a \Longrightarrow \exists A.\ (\forall x \in A.\ P\ x) \wedge (\exists y.\ y \in A)$
— Example 3.
⟨*proof*⟩

lemma $a < b \wedge b < (c::int) \Longrightarrow \exists A.\ a \notin A \wedge b \in A \wedge c \notin A$
— Example 4.
⟨*proof*⟩

lemma $P\ (f\ b) \Longrightarrow \exists s\ A.\ (\forall x \in A.\ P\ x) \wedge f\ s \in A$
— Example 5, page 298.
⟨*proof*⟩

lemma $P\ (f\ b) \Longrightarrow \exists s\ A.\ (\forall x \in A.\ P\ x) \wedge f\ s \in A$
— Example 6.
⟨*proof*⟩

lemma $\exists A.\ a \notin A$
— Example 7.
⟨*proof*⟩

lemma $(\forall u\ v.\ u < (0::int) \longrightarrow u \neq |v|)$
$\longrightarrow (\exists A::int\ set.\ -2 \in A\ \&\ (\forall y.\ |y| \notin A))$

— Example 8 needs a small hint.
⟨*proof*⟩

Example 9 omitted (requires the reals).

The paper has no Example 10!

lemma $(\forall A.\ 0 \in A \wedge (\forall x \in A.\ Suc\ x \in A) \longrightarrow n \in A) \wedge$
$P\ 0 \wedge (\forall x.\ P\ x \longrightarrow P\ (Suc\ x)) \longrightarrow P\ n$
— Example 11: needs a hint.
⟨*proof*⟩

lemma
$(\forall A.\ (0,\ 0) \in A \wedge (\forall x\ y.\ (x,\ y) \in A \longrightarrow (Suc\ x,\ Suc\ y) \in A) \longrightarrow (n,\ m) \in A)$
$\wedge\ P\ n \longrightarrow P\ m$
— Example 12.
⟨*proof*⟩

lemma
$(\forall x.\ (\exists u.\ x = 2 * u) = (\neg\ (\exists v.\ Suc\ x = 2 * v))) \longrightarrow$
$(\exists A.\ \forall x.\ (x \in A) = (Suc\ x \notin A))$
— Example EO1: typo in article, and with the obvious fix it seems to require arithmetic reasoning.
⟨*proof*⟩

end

41 Examples and regression tests for automated termination proofs

theory *Termination*
imports *Main* ~~/*src*/*HOL*/*Library*/*Multiset*
begin

41.1 Manually giving termination relations using *relation* and *measure*

function *sum* :: *nat* ⇒ *nat* ⇒ *nat*
where
sum i N = (*if i* > *N then 0 else i* + *sum* (*Suc i*) *N*)
⟨*proof*⟩

termination ⟨*proof*⟩

function *foo* :: *nat* ⇒ *nat* ⇒ *nat*
where
foo i N = (*if i* > *N*
then (*if N* = *0 then 0 else foo 0* (*N* − *1*))
else i + *foo* (*Suc i*) *N*)

⟨*proof*⟩

termination ⟨*proof*⟩

41.2 *lexicographic-order*: Trivial examples

The *fun* command uses the method *lexicographic-order* by default, so it is not explicitly invoked.

```
fun identity :: nat ⇒ nat
where
  identity n = n

fun yaSuc :: nat ⇒ nat
where
  yaSuc 0 = 0
| yaSuc (Suc n) = Suc (yaSuc n)
```

41.3 Examples on natural numbers

```
fun bin :: (nat * nat) ⇒ nat
where
  bin (0, 0) = 1
| bin (Suc n, 0) = 0
| bin (0, Suc m) = 0
| bin (Suc n, Suc m) = bin (n, m) + bin (Suc n, m)


fun t :: (nat * nat) ⇒ nat
where
  t (0,n) = 0
| t (n,0) = 0
| t (Suc n, Suc m) = (if (n mod 2 = 0) then (t (Suc n, m)) else (t (n, Suc m)))


fun k :: (nat * nat) * (nat * nat) ⇒ nat
where
  k ((0,0),(0,0)) = 0
| k ((Suc z, y), (u,v)) = k((z, y), (u, v))
| k ((0, Suc y), (u,v)) = k((1, y), (u, v))
| k ((0,0), (Suc u, v)) = k((1, 1), (u, v))
| k ((0,0), (0, Suc v)) = k((1,1), (1,v))


fun gcd2 :: nat ⇒ nat ⇒ nat
where
  gcd2 x 0 = x
| gcd2 0 y = y
| gcd2 (Suc x) (Suc y) = (if x < y then gcd2 (Suc x) (y − x)
                                   else gcd2 (x − y) (Suc y))
```

```
fun ack :: (nat * nat) ⇒ nat
where
  ack (0, m) = Suc m
| ack (Suc n, 0) = ack(n, 1)
| ack (Suc n, Suc m) = ack (n, ack (Suc n, m))


fun greedy :: nat * nat * nat * nat * nat => nat
where
  greedy (Suc a, Suc b, Suc c, Suc d, Suc e) =
  (if (a < 10) then greedy (Suc a, Suc b, c, d + 2, Suc e) else
  (if (a < 20) then greedy (Suc a, b, Suc c, d, Suc e) else
  (if (a < 30) then greedy (Suc a, b, Suc c, d, Suc e) else
  (if (a < 40) then greedy (Suc a, b, Suc c, d, Suc e) else
  (if (a < 50) then greedy (Suc a, b, Suc c, d, Suc e) else
  (if (a < 60) then greedy (a, Suc b, Suc c, d, Suc e) else
  (if (a < 70) then greedy (a, Suc b, Suc c, d, Suc e) else
  (if (a < 80) then greedy (a, Suc b, Suc c, d, Suc e) else
  (if (a < 90) then greedy (Suc a, Suc b, Suc c, d, e) else
  greedy (Suc a, Suc b, Suc c, d, e)))))))))
| greedy (a, b, c, d, e) = 0


fun blowup :: nat => nat => nat => nat => nat => nat => nat => nat =>
nat => nat
where
  blowup 0 0 0 0 0 0 0 0 0 = 0
| blowup 0 0 0 0 0 0 0 0 (Suc i) = Suc (blowup i i i i i i i i i)
| blowup 0 0 0 0 0 0 0 (Suc h) i = Suc (blowup h h h h h h h h i)
| blowup 0 0 0 0 0 0 (Suc g) h i = Suc (blowup g g g g g g g h i)
| blowup 0 0 0 0 0 (Suc f) g h i = Suc (blowup f f f f f f g h i)
| blowup 0 0 0 0 (Suc e) f g h i = Suc (blowup e e e e e f g h i)
| blowup 0 0 0 (Suc d) e f g h i = Suc (blowup d d d d e f g h i)
| blowup 0 0 (Suc c) d e f g h i = Suc (blowup c c c d e f g h i)
| blowup 0 (Suc b) c d e f g h i = Suc (blowup b b c d e f g h i)
| blowup (Suc a) b c d e f g h i = Suc (blowup a b c d e f g h i)
```

41.4 Simple examples with other datatypes than nat, e.g. trees and lists

```
datatype tree = Node | Branch tree tree

fun g-tree :: tree * tree ⇒ tree
where
  g-tree (Node, Node) = Node
| g-tree (Node, Branch a b) = Branch Node (g-tree (a,b))
| g-tree (Branch a b, Node) = Branch (g-tree (a,Node)) b
| g-tree (Branch a b, Branch c d) = Branch (g-tree (a,c)) (g-tree (b,d))
```

fun *acklist* :: *'a list* ∗ *'a list* ⇒ *'a list*
where
acklist ([], *m*) = ((*hd m*)#*m*)
| *acklist* (*n*#*ns*, []) = *acklist* (*ns*, [*n*])
| *acklist* ((*n*#*ns*), (*m*#*ms*)) = *acklist* (*ns*, *acklist* ((*n*#*ns*), *ms*))

41.5 Examples with mutual recursion

fun *evn od* :: *nat* ⇒ *bool*
where
evn 0 = *True*
| *od 0* = *False*
| *evn* (*Suc n*) = *od* (*Suc n*)
| *od* (*Suc n*) = *evn n*

fun *sizechange-f* :: *'a list* => *'a list* => *'a list* **and**
sizechange-g :: *'a list* => *'a list* => *'a list* => *'a list*
where
sizechange-f i x = (*if i*=[] *then x else sizechange-g* (*tl i*) *x i*)
| *sizechange-g a b c* = *sizechange-f a* (*b* @ *c*)

fun
pedal :: *nat* => *nat* => *nat* => *nat*
and
coast :: *nat* => *nat* => *nat* => *nat*
where
pedal 0 m c = *c*
| *pedal n 0 c* = *c*
| *pedal n m c* =
(*if n* < *m then coast* (*n* − *1*) (*m* − *1*) (*c* + *m*)
else pedal (*n* − *1*) *m* (*c* + *m*))

| *coast n m c* =
(*if n* < *m then coast n* (*m* − *1*) (*c* + *n*)
else pedal n m (*c* + *n*))

41.6 Refined analysis: The *size-change* method

Unsolvable for *lexicographic-order*

function *fun1* :: *nat* ∗ *nat* ⇒ *nat*
where
fun1 (*0*,*0*) = *1*
| *fun1* (*0*, *Suc b*) = *0*
| *fun1* (*Suc a*, *0*) = *0*
| *fun1* (*Suc a*, *Suc b*) = *fun1* (*b*, *a*)
⟨*proof*⟩

termination ⟨*proof*⟩

lexicographic-order can do the following, but it is much slower.

```
function
  prod :: nat => nat => nat => nat and
  eprod :: nat => nat => nat => nat and
  oprod :: nat => nat => nat => nat
where
  prod x y z = (if y mod 2 = 0 then eprod x y z else oprod x y z)
| oprod x y z = eprod x (y - 1) (z+x)
| eprod x y z = (if y=0 then z else prod (2*x) (y div 2) z)
⟨proof⟩
termination ⟨proof⟩
```

Permutations of arguments:

```
function perm :: nat ⇒ nat ⇒ nat ⇒ nat
where
  perm m n r = (if r > 0 then perm m (r - 1) n
                else if n > 0 then perm r (n - 1) m
                else m)
⟨proof⟩
termination ⟨proof⟩
```

Artificial examples and regression tests:

```
function
  fun2 :: nat ⇒ nat ⇒ nat ⇒ nat
where
  fun2 x y z =
     (if x > 1000 ∧ z > 0 then
         fun2 (min x y) y (z - 1)
      else if y > 0 ∧ x > 100 then
         fun2 x (y - 1) (2 * z)
      else if z > 0 then
         fun2 (min y (z - 1)) x x
      else
         0
     )
⟨proof⟩
termination ⟨proof⟩

definition negate :: int ⇒ int
where negate i = - i

function fun3 :: int => nat
where
  fun3 i =
  (if i < 0 then fun3 (negate i)
   else if i = 0 then 0
   else fun3 (i - 1))
```

⟨*proof*⟩
termination
⟨*proof*⟩

end

42 Coherent Logic Problems

theory *Coherent*
imports *Main*
begin

42.1 Equivalence of two versions of Pappus' Axiom

no-notation
comp (**infixl** *o* 55) **and**
relcomp (**infixr** *O* 75)

lemma *p1p2*:
assumes *col a b c l* ∧ *col d e f m*
and *col b f g n* ∧ *col c e g o*
and *col b d h p* ∧ *col a e h q*
and *col c d i r* ∧ *col a f i s*
and *el n o* ⟹ *goal*
and *el p q* ⟹ *goal*
and *el s r* ⟹ *goal*
and ⋀*A*. *el A A* ⟹ *pl g A* ⟹ *pl h A* ⟹ *pl i A* ⟹ *goal*
and ⋀*A B C D*. *col A B C D* ⟹ *pl A D*
and ⋀*A B C D*. *col A B C D* ⟹ *pl B D*
and ⋀*A B C D*. *col A B C D* ⟹ *pl C D*
and ⋀*A B*. *pl A B* ⟹ *ep A A*
and ⋀*A B*. *ep A B* ⟹ *ep B A*
and ⋀*A B C*. *ep A B* ⟹ *ep B C* ⟹ *ep A C*
and ⋀*A B*. *pl A B* ⟹ *el B B*
and ⋀*A B*. *el A B* ⟹ *el B A*
and ⋀*A B C*. *el A B* ⟹ *el B C* ⟹ *el A C*
and ⋀*A B C*. *ep A B* ⟹ *pl B C* ⟹ *pl A C*
and ⋀*A B C*. *pl A B* ⟹ *el B C* ⟹ *pl A C*
and ⋀*A B C D E F G H I J K L M N O P Q*.
col A B C D ⟹ *col E F G H* ⟹ *col B G I J* ⟹ *col C F I K* ⟹
col B E L M ⟹ *col A F L N* ⟹ *col C E O P* ⟹ *col A G O Q* ⟹
(∃ *R*. *col I L O R*) ∨ *pl A H* ∨ *pl B H* ∨ *pl C H* ∨ *pl E D* ∨ *pl F D* ∨ *pl G D*
and ⋀*A B C D*. *pl A B* ⟹ *pl A C* ⟹ *pl D B* ⟹ *pl D C* ⟹ *ep A D* ∨ *el B C*
and ⋀*A B*. *ep A A* ⟹ *ep B B* ⟹ ∃ *C*. *pl A C* ∧ *pl B C*
shows *goal* ⟨*proof*⟩

lemma *p2p1*:
assumes *col a b c l* ∧ *col d e f m*
and *col b f g n* ∧ *col c e g o*
and *col b d h p* ∧ *col a e h q*
and *col c d i r* ∧ *col a f i s*
and *pl a m* ⟹ *goal*
and *pl b m* ⟹ *goal*
and *pl c m* ⟹ *goal*
and *pl d l* ⟹ *goal*
and *pl e l* ⟹ *goal*
and *pl f l* ⟹ *goal*
and ⋀*A*. *pl g A* ⟹ *pl h A* ⟹ *pl i A* ⟹ *goal*
and ⋀*A B C D*. *col A B C D* ⟹ *pl A D*
and ⋀*A B C D*. *col A B C D* ⟹ *pl B D*
and ⋀*A B C D*. *col A B C D* ⟹ *pl C D*
and ⋀*A B*. *pl A B* ⟹ *ep A A*
and ⋀*A B*. *ep A B* ⟹ *ep B A*
and ⋀*A B C*. *ep A B* ⟹ *ep B C* ⟹ *ep A C*
and ⋀*A B*. *pl A B* ⟹ *el B B*
and ⋀*A B*. *el A B* ⟹ *el B A*
and ⋀*A B C*. *el A B* ⟹ *el B C* ⟹ *el A C*
and ⋀*A B C*. *ep A B* ⟹ *pl B C* ⟹ *pl A C*
and ⋀*A B C*. *pl A B* ⟹ *el B C* ⟹ *pl A C*
and ⋀*A B C D E F G H I J K L M N O P Q*.
col A B C J ⟹ *col D E F K* ⟹ *col B F G L* ⟹ *col C E G M* ⟹
col B D H N ⟹ *col A E H O* ⟹ *col C D I P* ⟹ *col A F I Q* ⟹
(∃ *R*. *col G H I R*) ∨ *el L M* ∨ *el N O* ∨ *el P Q*
and ⋀*A B C D*. *pl C A* ⟹ *pl C B* ⟹ *pl D A* ⟹ *pl D B* ⟹ *ep C D* ∨ *el A B*
and ⋀*A B C*. *ep A A* ⟹ *ep B B* ⟹ ∃ *C*. *pl A C* ∧ *pl B C*
shows *goal* ⟨*proof*⟩

42.2 Preservation of the Diamond Property under reflexive closure

lemma *diamond*:
assumes *reflexive-rewrite a b reflexive-rewrite a c*
and ⋀*A*. *reflexive-rewrite b A* ⟹ *reflexive-rewrite c A* ⟹ *goal*
and ⋀*A*. *equalish A A*
and ⋀*A B*. *equalish A B* ⟹ *equalish B A*
and ⋀*A B C*. *equalish A B* ⟹ *reflexive-rewrite B C* ⟹ *reflexive-rewrite A C*
and ⋀*A B*. *equalish A B* ⟹ *reflexive-rewrite A B*
and ⋀*A B*. *rewrite A B* ⟹ *reflexive-rewrite A B*
and ⋀*A B*. *reflexive-rewrite A B* ⟹ *equalish A B* ∨ *rewrite A B*
and ⋀*A B C*. *rewrite A B* ⟹ *rewrite A C* ⟹ ∃ *D*. *rewrite B D* ∧ *rewrite C D*
shows *goal* ⟨*proof*⟩

end

43 Some examples for Presburger Arithmetic

theory *PresburgerEx*
imports *Presburger*
begin

lemma $\bigwedge m\ n\ ja\ ia.\ [\![\neg\ m \le j;\ \neg\ (n::nat) \le i;\ (e::nat) \neq 0;\ Suc\ j \le ja]\!] \Longrightarrow \exists\, m.\ \forall\, ja\ ia.\ m \le ja \longrightarrow (if\ j = ja \wedge i = ia\ then\ e\ else\ 0) = 0$ ⟨*proof*⟩
lemma $(0::nat) < emBits\ mod\ 8 \Longrightarrow 8 + emBits\ div\ 8 * 8 - emBits = 8 - emBits\ mod\ 8$
⟨*proof*⟩
lemma $(0::nat) < emBits\ mod\ 8 \Longrightarrow 8 + emBits\ div\ 8 * 8 - emBits = 8 - emBits\ mod\ 8$
⟨*proof*⟩

theorem $(\forall\,(y::int).\ 3\ dvd\ y) ==> \forall\,(x::int).\ b < x --> a \le x$
⟨*proof*⟩

theorem !! $(y::int)\ (z::int)\ (n::int).\ 3\ dvd\ z ==> 2\ dvd\ (y::int) ==>$
$(\exists\,(x::int).\ \ 2*x = \ \ y)\ \&\ (\exists\,(k::int).\ 3*k = z)$
⟨*proof*⟩

theorem !! $(y::int)\ (z::int)\ n.\ Suc(n::nat) < 6 ==>\ \ 3\ dvd\ z ==>$
$2\ dvd\ (y::int) ==> (\exists\,(x::int).\ \ 2*x = \ \ y)\ \&\ (\exists\,(k::int).\ 3*k = z)$
⟨*proof*⟩

theorem $\forall\,(x::nat).\ \exists\,(y::nat).\ (0::nat) \le 5 --> y = 5 + x$
⟨*proof*⟩

Slow: about 7 seconds on a 1.6GHz machine.

theorem $\forall\,(x::nat).\ \exists\,(y::nat).\ y = 5 + x \mid x\ div\ 6 + 1 = 2$
⟨*proof*⟩

theorem $\exists\,(x::int).\ 0 < x$
⟨*proof*⟩

theorem $\forall\,(x::int)\ y.\ x < y --> 2 * x + 1 < 2 * y$
⟨*proof*⟩

theorem $\forall\,(x::int)\ y.\ 2 * x + 1 \neq 2 * y$
⟨*proof*⟩

theorem $\exists\,(x::int)\ y.\ 0 < x\ \ \&\ 0 \le y\ \ \&\ 3 * x - 5 * y = 1$
⟨*proof*⟩

theorem ~ $(\exists\,(x::int)\ (y::int)\ (z::int).\ 4*x + (-6::int)*y = 1)$
⟨*proof*⟩

theorem $\forall (x{::}int).\ b < x \dashrightarrow a \leq x$
⟨*proof*⟩

theorem $\sim (\exists (x{::}int).\ False)$
⟨*proof*⟩

theorem $\forall (x{::}int).\ (a{::}int) < 3 * x \dashrightarrow b < 3 * x$
⟨*proof*⟩

theorem $\forall (x{::}int).\ (2\ dvd\ x) \dashrightarrow (\exists (y{::}int).\ x = 2{*}y)$
⟨*proof*⟩

theorem $\forall (x{::}int).\ (2\ dvd\ x) \dashrightarrow (\exists (y{::}int).\ x = 2{*}y)$
⟨*proof*⟩

theorem $\forall (x{::}int).\ (2\ dvd\ x) = (\exists (y{::}int).\ x = 2{*}y)$
⟨*proof*⟩

theorem $\forall (x{::}int).\ ((2\ dvd\ x) = (\forall (y{::}int).\ x \neq 2{*}y + 1))$
⟨*proof*⟩

theorem $\sim (\forall (x{::}int).$
$((2\ dvd\ x) = (\forall (y{::}int).\ x \neq 2{*}y{+}1) \mid$
$(\exists (q{::}int)\ (u{::}int)\ i.\ 3{*}i + 2{*}q - u < 17)$
$\dashrightarrow 0 < x \mid ((\sim 3\ dvd\ x)\ \&(x + 8 = 0))))$
⟨*proof*⟩

theorem $\sim (\forall (i{::}int).\ 4 \leq i \dashrightarrow (\exists\, x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i))$
⟨*proof*⟩

theorem $\forall (i{::}int).\ 8 \leq i \dashrightarrow (\exists\, x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i)$
⟨*proof*⟩

theorem $\exists (j{::}int).\ \forall\, i.\ j \leq i \dashrightarrow (\exists\, x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i)$
⟨*proof*⟩

theorem $\sim (\forall\, j\ (i{::}int).\ j \leq i \dashrightarrow (\exists\, x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i))$
⟨*proof*⟩

Slow: about 5 seconds on a 1.6GHz machine.

theorem $(\exists\, m{::}nat.\ n = 2 * m) \dashrightarrow (n + 1)\ div\ 2 = n\ div\ 2$
⟨*proof*⟩

This following theorem proves that all solutions to the recurrence relation $x_{i+2} = |x_{i+1}| - x_i$ are periodic with period 9. The example was brought to our attention by John Harrison. It does does not require Presburger arithmetic but merely quantifier-free linear arithmetic and holds for the

rationals as well.

Warning: it takes (in 2006) over 4.2 minutes!

lemma $[\![\ x3 = |x2| - x1;\ x4 = |x3| - x2;\ x5 = |x4| - x3;$
$x6 = |x5| - x4;\ x7 = |x6| - x5;\ x8 = |x7| - x6;$
$x9 = |x8| - x7;\ x10 = |x9| - x8;\ x11 = |x10| - x9\ ]\!]$
$\Longrightarrow x1 = x10\ \&\ x2 = (x11::int)$
⟨*proof*⟩

end

44 Generic reflection and reification

theory *Reflection*
imports *Main*
begin

⟨*ML*⟩

end

45 Examples for generic reflection and reification

theory *Reflection-Examples*
imports *Complex-Main* ~~/*src/HOL/Library/Reflection*
begin

This theory presents two methods: reify and reflection

Consider an HOL type σ, the structure of which is not recongnisable on the theory level. This is the case of *bool*, arithmetical terms such as *int*, *real* etc ... In order to implement a simplification on terms of type σ we often need its structure. Traditionnaly such simplifications are written in ML, proofs are synthesized.

An other strategy is to declare an HOL datatype τ and an HOL function (the interpretation) that maps elements of τ to elements of σ.

The functionality of *reify* then is, given a term t of type σ, to compute a term s of type τ. For this it needs equations for the interpretation.

N.B: All the interpretations supported by *reify* must have the type *'a list* $\Rightarrow \tau \Rightarrow \sigma$. The method *reify* can also be told which subterm of the current subgoal should be reified. The general call for *reify* is *reify eqs* (t), where *eqs* are the defining equations of the interpretation and (t) is an optional parameter which specifies the subterm to which reification should be applied to. If (t) is abscent, *reify* tries to reify the whole subgoal.

The method *reflection* uses *reify* and has a very similar signature: *reflection corr-thm eqs* (t). Here again *eqs* and (t) are as described above and *corr-thm*

is a theorem proving $I\ vs\ (f\ t) = I\ vs\ t$. We assume that I is the interpretation and f is some useful and executable simplification of type $\tau \Rightarrow \tau$. The method *reflection* applies reification and hence the theorem $t = I\ xs\ s$ and hence using *corr-thm* derives $t = I\ xs\ (f\ s)$. It then uses normalization by equational rewriting to prove $f\ s = s'$ which almost finishes the proof of $t = t'$ where $I\ xs\ s' = t'$.

Example 1 : Propositional formulae and NNF.

The type *fm* represents simple propositional formulae:

datatype *form* = *TrueF* | *FalseF* | *Less nat nat*
  | *And form form* | *Or form form* | *Neg form* | *ExQ form*

primrec *interp* :: *form* ⇒ ('*a*::*ord*) *list* ⇒ *bool*
where
  interp TrueF vs ⟷ *True*
| *interp FalseF vs* ⟷ *False*
| *interp* (*Less i j*) *vs* ⟷ *vs* ! *i* < *vs* ! *j*
| *interp* (*And f1 f2*) *vs* ⟷ *interp f1 vs* ∧ *interp f2 vs*
| *interp* (*Or f1 f2*) *vs* ⟷ *interp f1 vs* ∨ *interp f2 vs*
| *interp* (*Neg f*) *vs* ⟷ ¬ *interp f vs*
| *interp* (*ExQ f*) *vs* ⟷ (∃ *v*. *interp f* (*v* # *vs*))

lemmas *interp-reify-eqs* = *interp.simps*
declare *interp-reify-eqs* [*reify*]

lemma ∃ *x*. *x* < *y* ∧ *x* < *z*
  ⟨*proof*⟩

datatype *fm* = *And fm fm* | *Or fm fm* | *Imp fm fm* | *Iff fm fm* | *NOT fm* | *At nat*

primrec *Ifm* :: *fm* ⇒ *bool list* ⇒ *bool*
where
  Ifm (*At n*) *vs* ⟷ *vs* ! *n*
| *Ifm* (*And p q*) *vs* ⟷ *Ifm p vs* ∧ *Ifm q vs*
| *Ifm* (*Or p q*) *vs* ⟷ *Ifm p vs* ∨ *Ifm q vs*
| *Ifm* (*Imp p q*) *vs* ⟷ *Ifm p vs* ⟶ *Ifm q vs*
| *Ifm* (*Iff p q*) *vs* ⟷ *Ifm p vs* = *Ifm q vs*
| *Ifm* (*NOT p*) *vs* ⟷ ¬ *Ifm p vs*

lemma *Q* ⟶ (*D* ∧ *F* ∧ ((¬ *D*) ∧ (¬ *F*)))
  ⟨*proof*⟩

Method *reify* maps a *bool* to an *fm*. For this it needs the semantics of *fm*, i.e. the rewrite rules in *Ifm.simps*.

You can also just pick up a subterm to reify.

lemma *Q* ⟶ (*D* ∧ *F* ∧ ((¬ *D*) ∧ (¬ *F*)))

⟨*proof*⟩

Let's perform NNF. This is a version that tends to generate disjunctions

primrec *fmsize* :: *fm* ⇒ *nat*
where
fmsize (*At n*) = *1*
| *fmsize* (*NOT p*) = *1* + *fmsize p*
| *fmsize* (*And p q*) = *1* + *fmsize p* + *fmsize q*
| *fmsize* (*Or p q*) = *1* + *fmsize p* + *fmsize q*
| *fmsize* (*Imp p q*) = *2* + *fmsize p* + *fmsize q*
| *fmsize* (*Iff p q*) = *2* + *2*∗ *fmsize p* + *2*∗ *fmsize q*

lemma [*measure-function*]: *is-measure fmsize* ⟨*proof*⟩

fun *nnf* :: *fm* ⇒ *fm*
where
nnf (*At n*) = *At n*
| *nnf* (*And p q*) = *And* (*nnf p*) (*nnf q*)
| *nnf* (*Or p q*) = *Or* (*nnf p*) (*nnf q*)
| *nnf* (*Imp p q*) = *Or* (*nnf* (*NOT p*)) (*nnf q*)
| *nnf* (*Iff p q*) = *Or* (*And* (*nnf p*) (*nnf q*)) (*And* (*nnf* (*NOT p*)) (*nnf* (*NOT q*)))
| *nnf* (*NOT* (*And p q*)) = *Or* (*nnf* (*NOT p*)) (*nnf* (*NOT q*))
| *nnf* (*NOT* (*Or p q*)) = *And* (*nnf* (*NOT p*)) (*nnf* (*NOT q*))
| *nnf* (*NOT* (*Imp p q*)) = *And* (*nnf p*) (*nnf* (*NOT q*))
| *nnf* (*NOT* (*Iff p q*)) = *Or* (*And* (*nnf p*) (*nnf* (*NOT q*))) (*And* (*nnf* (*NOT p*)) (*nnf q*))
| *nnf* (*NOT* (*NOT p*)) = *nnf p*
| *nnf* (*NOT p*) = *NOT p*

The correctness theorem of *nnf*: it preserves the semantics of *fm*

lemma *nnf* [*reflection*]:
Ifm (*nnf p*) *vs* = *Ifm p vs*
⟨*proof*⟩

Now let's perform NNF using our *nnf* function defined above. First to the whole subgoal.

lemma $A \neq B \wedge (B \longrightarrow A \neq (B \vee C \wedge (B \longrightarrow A \vee D))) \longrightarrow A \vee B \wedge D$
⟨*proof*⟩

Now we specify on which subterm it should be applied

lemma $A \neq B \wedge (B \longrightarrow A \neq (B \vee C \wedge (B \longrightarrow A \vee D))) \longrightarrow A \vee B \wedge D$
⟨*proof*⟩

Example 2: Simple arithmetic formulae

The type *num* reflects linear expressions over natural number

datatype *num* = *C nat* | *Add num num* | *Mul nat num* | *Var nat* | *CN nat nat num*

This is just technical to make recursive definitions easier.

primrec *num-size* :: *num* ⇒ *nat*
where
  num-size (*C c*) = *1*
| *num-size* (*Var n*) = *1*
| *num-size* (*Add a b*) = *1* + *num-size a* + *num-size b*
| *num-size* (*Mul c a*) = *1* + *num-size a*
| *num-size* (*CN n c a*) = *4* + *num-size a*

lemma [*measure-function*]: *is-measure num-size* ⟨*proof*⟩

The semantics of num

primrec *Inum*:: *num* ⇒ *nat list* ⇒ *nat*
where
  Inum-C : *Inum* (*C i*) *vs* = *i*
| *Inum-Var*: *Inum* (*Var n*) *vs* = *vs*!*n*
| *Inum-Add*: *Inum* (*Add s t*) *vs* = *Inum s vs* + *Inum t vs*
| *Inum-Mul*: *Inum* (*Mul c t*) *vs* = *c* ∗ *Inum t vs*
| *Inum-CN* : *Inum* (*CN n c t*) *vs* = *c*∗(*vs*!*n*) + *Inum t vs*

Let's reify some nat expressions ...

lemma *4* ∗ (*2* ∗ *x* + (*y*::*nat*)) + *f a* ≠ *0*
  ⟨*proof*⟩

We're in a bad situation! x, y and f have been recongnized as constants, which is correct but does not correspond to our intuition of the constructor C. It should encapsulate constants, i.e. numbers, i.e. numerals.

So let's leave the *Inum-C* equation at the end and see what happens ...

lemma *4* ∗ (*2* ∗ *x* + (*y*::*nat*)) ≠ *0*
  ⟨*proof*⟩

Hm, let's specialize *Inum-C* with numerals.

lemma *Inum-number*: *Inum* (*C* (*numeral t*)) *vs* = *numeral t* ⟨*proof*⟩
lemmas *Inum-eqs* = *Inum-Var Inum-Add Inum-Mul Inum-CN Inum-number*

Second attempt

lemma *1* ∗ (*2* ∗ *x* + (*y*::*nat*)) ≠ *0*
  ⟨*proof*⟩

That was fine, so let's try another one ...

lemma *1* ∗ (*2* ∗ *x* + (*y*::*nat*) + *0* + *1*) ≠ *0*
  ⟨*proof*⟩

Oh!! 0 is not a variable ... Oh! 0 is not a *numeral* ... thing. The same for 1. So let's add those equations, too.

lemma *Inum-01*: *Inum (C 0) vs = 0 Inum (C 1) vs = 1 Inum (C(Suc n)) vs = Suc n*
⟨*proof*⟩

lemmas *Inum-eqs'= Inum-eqs Inum-01*

Third attempt:

lemma *1 * (2 * x + (y::nat) + 0 + 1) ≠ 0*
⟨*proof*⟩

Okay, let's try reflection. Some simplifications on *Reflection-Examples.num* follow. You can skim until the main theorem *linum*.

fun *lin-add* :: *num ⇒ num ⇒ num*
where
lin-add (CN n1 c1 r1) (CN n2 c2 r2) =
(if n1 = n2 then
(let c = c1 + c2
in (if c = 0 then lin-add r1 r2 else CN n1 c (lin-add r1 r2)))
else if n1 ≤ n2 then (CN n1 c1 (lin-add r1 (CN n2 c2 r2)))
else (CN n2 c2 (lin-add (CN n1 c1 r1) r2)))
| *lin-add (CN n1 c1 r1) t = CN n1 c1 (lin-add r1 t)*
| *lin-add t (CN n2 c2 r2) = CN n2 c2 (lin-add t r2)*
| *lin-add (C b1) (C b2) = C (b1 + b2)*
| *lin-add a b = Add a b*

lemma *lin-add*:
Inum (lin-add t s) bs = Inum (Add t s) bs
⟨*proof*⟩

fun *lin-mul* :: *num ⇒ nat ⇒ num*
where
*lin-mul (C j) i = C (i * j)*
| *lin-mul (CN n c a) i = (if i=0 then (C 0) else CN n (i * c) (lin-mul a i))*
| *lin-mul t i = (Mul i t)*

lemma *lin-mul*:
Inum (lin-mul t i) bs = Inum (Mul i t) bs
⟨*proof*⟩

fun *linum*:: *num ⇒ num*
where
linum (C b) = C b
| *linum (Var n) = CN n 1 (C 0)*
| *linum (Add t s) = lin-add (linum t) (linum s)*
| *linum (Mul c t) = lin-mul (linum t) c*
| *linum (CN n c t) = lin-add (linum (Mul c (Var n))) (linum t)*

lemma *linum* [*reflection*]:
Inum (linum t) bs = Inum t bs

⟨*proof*⟩

Now we can use linum to simplify nat terms using reflection

lemma *Suc (Suc 1) * (x + Suc 1 * y) = 3 * x + 6 * y*
⟨*proof*⟩

Let's lift this to formulae and see what happens

datatype *aform = Lt num num | Eq num num | Ge num num | NEq num num*
|
Conj aform aform | Disj aform aform | NEG aform | T | F

primrec *linaformsize*:: *aform ⇒ nat*
where
linaformsize T = 1
| *linaformsize F = 1*
| *linaformsize (Lt a b) = 1*
| *linaformsize (Ge a b) = 1*
| *linaformsize (Eq a b) = 1*
| *linaformsize (NEq a b) = 1*
| *linaformsize (NEG p) = 2 + linaformsize p*
| *linaformsize (Conj p q) = 1 + linaformsize p + linaformsize q*
| *linaformsize (Disj p q) = 1 + linaformsize p + linaformsize q*

lemma [*measure-function*]: *is-measure linaformsize* ⟨*proof*⟩

primrec *is-aform* :: *aform => nat list => bool*
where
is-aform T vs = True
| *is-aform F vs = False*
| *is-aform (Lt a b) vs = (Inum a vs < Inum b vs)*
| *is-aform (Eq a b) vs = (Inum a vs = Inum b vs)*
| *is-aform (Ge a b) vs = (Inum a vs ≥ Inum b vs)*
| *is-aform (NEq a b) vs = (Inum a vs ≠ Inum b vs)*
| *is-aform (NEG p) vs = (¬ (is-aform p vs))*
| *is-aform (Conj p q) vs = (is-aform p vs ∧ is-aform q vs)*
| *is-aform (Disj p q) vs = (is-aform p vs ∨ is-aform q vs)*

Let's reify and do reflection

lemma (*3*::*nat*) * *x* + *t* < *0* ∧ (*2* * *x* + *y* ≠ *17*)
⟨*proof*⟩

Note that reification handles several interpretations at the same time

lemma (*3*::*nat*) * *x* + *t* < *0* ∧ *x* * *x* + *t* * *x* + *3* + *1* = *z* * *t* * *4* * *z* ∨ *x* + *x* + *1* < *0*
⟨*proof*⟩

For reflection we now define a simple transformation on aform: NNF + linum on atoms

fun *linaform*:: *aform* ⇒ *aform*
where
  linaform (*Lt s t*) = *Lt* (*linum s*) (*linum t*)
| *linaform* (*Eq s t*) = *Eq* (*linum s*) (*linum t*)
| *linaform* (*Ge s t*) = *Ge* (*linum s*) (*linum t*)
| *linaform* (*NEq s t*) = *NEq* (*linum s*) (*linum t*)
| *linaform* (*Conj p q*) = *Conj* (*linaform p*) (*linaform q*)
| *linaform* (*Disj p q*) = *Disj* (*linaform p*) (*linaform q*)
| *linaform* (*NEG T*) = *F*
| *linaform* (*NEG F*) = *T*
| *linaform* (*NEG* (*Lt a b*)) = *Ge a b*
| *linaform* (*NEG* (*Ge a b*)) = *Lt a b*
| *linaform* (*NEG* (*Eq a b*)) = *NEq a b*
| *linaform* (*NEG* (*NEq a b*)) = *Eq a b*
| *linaform* (*NEG* (*NEG p*)) = *linaform p*
| *linaform* (*NEG* (*Conj p q*)) = *Disj* (*linaform* (*NEG p*)) (*linaform* (*NEG q*))
| *linaform* (*NEG* (*Disj p q*)) = *Conj* (*linaform* (*NEG p*)) (*linaform* (*NEG q*))
| *linaform p* = *p*

lemma *linaform*: *is-aform* (*linaform p*) *vs* = *is-aform p vs*
  ⟨*proof*⟩

lemma (*Suc* (*Suc* (*Suc 0*)) ∗ ((*x*::*nat*) + *Suc* (*Suc 0*)) + *Suc* (*Suc* (*Suc 0*)) ∗
  (*Suc* (*Suc* (*Suc 0*))) ∗ ((*x*::*nat*) + *Suc* (*Suc 0*))) < *0* ∧ *Suc 0* + *Suc 0* < *0*
  ⟨*proof*⟩

declare *linaform* [*reflection*]

lemma (*Suc* (*Suc* (*Suc 0*)) ∗ ((*x*::*nat*) + *Suc* (*Suc 0*)) + *Suc* (*Suc* (*Suc 0*)) ∗
  (*Suc* (*Suc* (*Suc 0*))) ∗ ((*x*::*nat*) + *Suc* (*Suc 0*))) < *0* ∧ *Suc 0* + *Suc 0* < *0*
  ⟨*proof*⟩

We now give an example where interpretaions have zero or more than only one envornement of different types and show that automatic reification also deals with bindings

datatype *rb* = *BC bool* | *BAnd rb rb* | *BOr rb rb*

primrec *Irb* :: *rb* ⇒ *bool*
where
  Irb (*BC p*) ⟷ *p*
| *Irb* (*BAnd s t*) ⟷ *Irb s* ∧ *Irb t*
| *Irb* (*BOr s t*) ⟷ *Irb s* ∨ *Irb t*

lemma *A* ∧ (*B* ∨ *D* ∧ *B*) ∧ *A* ∧ (*B* ∨ *D* ∧ *B*) ∨ *A* ∧ (*B* ∨ *D* ∧ *B*) ∨ *A* ∧ (*B* ∨ *D* ∧ *B*)
  ⟨*proof*⟩

datatype *rint* = *IC int* | *IVar nat* | *IAdd rint rint* | *IMult rint rint*
  | *INeg rint* | *ISub rint rint*

primrec *Irint* :: *rint* ⇒ *int list* ⇒ *int*
where
  Irint-Var: *Irint* (*IVar n*) *vs* = *vs* ! *n*
| *Irint-Neg*: *Irint* (*INeg t*) *vs* = − *Irint t vs*
| *Irint-Add*: *Irint* (*IAdd s t*) *vs* = *Irint s vs* + *Irint t vs*
| *Irint-Sub*: *Irint* (*ISub s t*) *vs* = *Irint s vs* − *Irint t vs*
| *Irint-Mult*: *Irint* (*IMult s t*) *vs* = *Irint s vs* ∗ *Irint t vs*
| *Irint-C*: *Irint* (*IC i*) *vs* = *i*

lemma *Irint-C0*: *Irint* (*IC 0*) *vs* = *0*
  ⟨*proof*⟩

lemma *Irint-C1*: *Irint* (*IC 1*) *vs* = *1*
  ⟨*proof*⟩

lemma *Irint-Cnumeral*: *Irint* (*IC* (*numeral x*)) *vs* = *numeral x*
  ⟨*proof*⟩

lemmas *Irint-simps* = *Irint-Var Irint-Neg Irint-Add Irint-Sub Irint-Mult Irint-C0 Irint-C1 Irint-Cnumeral*

lemma (*3*::*int*) ∗ *x* + *y* ∗ *y* − *9* + (− *z*) = *0*
  ⟨*proof*⟩

datatype *rlist* = *LVar nat* | *LEmpty* | *LCons rint rlist* | *LAppend rlist rlist*

primrec *Irlist* :: *rlist* ⇒ *int list* ⇒ *int list list* ⇒ *int list*
where
  Irlist (*LEmpty*) *is vs* = []
| *Irlist* (*LVar n*) *is vs* = *vs* ! *n*
| *Irlist* (*LCons i t*) *is vs* = *Irint i is* # *Irlist t is vs*
| *Irlist* (*LAppend s t*) *is vs* = *Irlist s is vs* @ *Irlist t is vs*

lemma [(*1*::*int*)] = []
  ⟨*proof*⟩

lemma ([(*3*::*int*) ∗ *x* + *y* ∗ *y* − *9* + (− *z*)] @ []) @ *xs* = [*y* ∗ *y* − *z* − *9* + (*3*::*int*) ∗ *x*]
  ⟨*proof*⟩

datatype *rnat* = *NC nat*| *NVar nat*| *NSuc rnat* | *NAdd rnat rnat* | *NMult rnat rnat*
  | *NNeg rnat* | *NSub rnat rnat* | *Nlgth rlist*

primrec *Irnat* :: *rnat* ⇒ *int list* ⇒ *int list list* ⇒ *nat list* ⇒ *nat*
where
  Irnat-Suc: *Irnat* (*NSuc t*) *is ls vs* = *Suc* (*Irnat t is ls vs*)
| *Irnat-Var*: *Irnat* (*NVar n*) *is ls vs* = *vs* ! *n*

| *Irnat-Neg*: *Irnat* (*NNeg t*) *is ls vs* = *0*
| *Irnat-Add*: *Irnat* (*NAdd s t*) *is ls vs* = *Irnat s is ls vs* + *Irnat t is ls vs*
| *Irnat-Sub*: *Irnat* (*NSub s t*) *is ls vs* = *Irnat s is ls vs* − *Irnat t is ls vs*
| *Irnat-Mult*: *Irnat* (*NMult s t*) *is ls vs* = *Irnat s is ls vs* ∗ *Irnat t is ls vs*
| *Irnat-lgth*: *Irnat* (*Nlgth rxs*) *is ls vs* = *length* (*Irlist rxs is ls*)
| *Irnat-C*: *Irnat* (*NC i*) *is ls vs* = *i*

lemma *Irnat-C0*: *Irnat* (*NC 0*) *is ls vs* = *0*
⟨*proof*⟩

lemma *Irnat-C1*: *Irnat* (*NC 1*) *is ls vs* = *1*
⟨*proof*⟩

lemma *Irnat-Cnumeral*: *Irnat* (*NC* (*numeral x*)) *is ls vs* = *numeral x*
⟨*proof*⟩

lemmas *Irnat-simps* = *Irnat-Suc Irnat-Var Irnat-Neg Irnat-Add Irnat-Sub Irnat-Mult Irnat-lgth*
Irnat-C0 Irnat-C1 Irnat-Cnumeral

lemma *Suc n* ∗ *length* (([(*3*::*int*) ∗ *x* + *y* ∗ *y* − *9* + (− *z*)] @ []) @ *xs*) = *length xs*
⟨*proof*⟩

datatype *rifm* = *RT* | *RF* | *RVar nat*
| *RNLT rnat rnat* | *RNILT rnat rint* | *RNEQ rnat rnat*
| *RAnd rifm rifm* | *ROr rifm rifm* | *RImp rifm rifm*| *RIff rifm rifm*
| *RNEX rifm* | *RIEX rifm* | *RLEX rifm* | *RNALL rifm* | *RIALL rifm* | *RLALL rifm*
| *RBEX rifm* | *RBALL rifm*

primrec *Irifm* :: *rifm* ⇒ *bool list* ⇒ *int list* ⇒ (*int list*) *list* ⇒ *nat list* ⇒ *bool*
where
Irifm RT ps is ls ns ⟷ *True*
| *Irifm RF ps is ls ns* ⟷ *False*
| *Irifm* (*RVar n*) *ps is ls ns* ⟷ *ps* ! *n*
| *Irifm* (*RNLT s t*) *ps is ls ns* ⟷ *Irnat s is ls ns* < *Irnat t is ls ns*
| *Irifm* (*RNILT s t*) *ps is ls ns* ⟷ *int* (*Irnat s is ls ns*) < *Irint t is*
| *Irifm* (*RNEQ s t*) *ps is ls ns* ⟷ *Irnat s is ls ns* = *Irnat t is ls ns*
| *Irifm* (*RAnd p q*) *ps is ls ns* ⟷ *Irifm p ps is ls ns* ∧ *Irifm q ps is ls ns*
| *Irifm* (*ROr p q*) *ps is ls ns* ⟷ *Irifm p ps is ls ns* ∨ *Irifm q ps is ls ns*
| *Irifm* (*RImp p q*) *ps is ls ns* ⟷ *Irifm p ps is ls ns* ⟶ *Irifm q ps is ls ns*
| *Irifm* (*RIff p q*) *ps is ls ns* ⟷ *Irifm p ps is ls ns* = *Irifm q ps is ls ns*
| *Irifm* (*RNEX p*) *ps is ls ns* ⟷ (∃ *x*. *Irifm p ps is ls* (*x* # *ns*))
| *Irifm* (*RIEX p*) *ps is ls ns* ⟷ (∃ *x*. *Irifm p ps* (*x* # *is*) *ls ns*)
| *Irifm* (*RLEX p*) *ps is ls ns* ⟷ (∃ *x*. *Irifm p ps is* (*x* # *ls*) *ns*)
| *Irifm* (*RBEX p*) *ps is ls ns* ⟷ (∃ *x*. *Irifm p* (*x* # *ps*) *is ls ns*)
| *Irifm* (*RNALL p*) *ps is ls ns* ⟷ (∀ *x*. *Irifm p ps is ls* (*x*#*ns*))
| *Irifm* (*RIALL p*) *ps is ls ns* ⟷ (∀ *x*. *Irifm p ps* (*x* # *is*) *ls ns*)

| *Irifm (RLALL p) ps is ls ns* $\longleftrightarrow$ $(\forall x.$ *Irifm p ps is (x#ls) ns*)
| *Irifm (RBALL p) ps is ls ns* $\longleftrightarrow$ $(\forall x.$ *Irifm p (x # ps) is ls ns*)

lemma $\forall x.\ \exists n.$ ((*Suc n*) $*$ *length* (([(*3*::*int*) $*$ *x* + *f t* $*$ *y* − *9* + (− *z*)] @ []) @ *xs*) = *length xs*) $\wedge$ *m* < *5*$*$*n* − *length* (*xs* @ [*2*,*3*,*4*,*x*$*$*z* + *8* − *y*]) $\longrightarrow$ $(\exists p.\ \forall q.$ *p* $\wedge$ *q* $\longrightarrow$ *r*)
⟨*proof*⟩

An example for equations containing type variables

datatype *prod* = *Zero* | *One* | *Var nat* | *Mul prod prod*
| *Pw prod nat* | *PNM nat nat prod*

primrec *Iprod* :: *prod* $\Rightarrow$ (*'a*::*linordered-idom*) *list* $\Rightarrow$*'a*
where
Iprod Zero vs = *0*
| *Iprod One vs* = *1*
| *Iprod* (*Var n*) *vs* = *vs* ! *n*
| *Iprod* (*Mul a b*) *vs* = *Iprod a vs* $*$ *Iprod b vs*
| *Iprod* (*Pw a n*) *vs* = *Iprod a vs* ^ *n*
| *Iprod* (*PNM n k t*) *vs* = (*vs* ! *n*) ^ *k* $*$ *Iprod t vs*

datatype *sgn* = *Pos prod* | *Neg prod* | *ZeroEq prod* | *NZeroEq prod* | *Tr* | *F*
| *Or sgn sgn* | *And sgn sgn*

primrec *Isgn* :: *sgn* $\Rightarrow$ (*'a*::*linordered-idom*) *list* $\Rightarrow$ *bool*
where
Isgn Tr vs $\longleftrightarrow$ *True*
| *Isgn F vs* $\longleftrightarrow$ *False*
| *Isgn* (*ZeroEq t*) *vs* $\longleftrightarrow$ *Iprod t vs* = *0*
| *Isgn* (*NZeroEq t*) *vs* $\longleftrightarrow$ *Iprod t vs* $\neq$ *0*
| *Isgn* (*Pos t*) *vs* $\longleftrightarrow$ *Iprod t vs* > *0*
| *Isgn* (*Neg t*) *vs* $\longleftrightarrow$ *Iprod t vs* < *0*
| *Isgn* (*And p q*) *vs* $\longleftrightarrow$ *Isgn p vs* $\wedge$ *Isgn q vs*
| *Isgn* (*Or p q*) *vs* $\longleftrightarrow$ *Isgn p vs* $\vee$ *Isgn q vs*

lemmas *eqs* = *Isgn.simps Iprod.simps*

lemma (*x*::*'a*::{*linordered-idom*}) ^ *4* $*$ *y* $*$ *z* $*$ *y* ^ *2* $*$ *z* ^ *23* > *0*
⟨*proof*⟩

end

46 Factorial (semi)rings

theory *Factorial-Ring*
imports
Main
~~/*src*/*HOL*/*GCD*
~~/*src*/*HOL*/*Library*/*Multiset*

begin

46.1 Irreducible and prime elements

context *comm-semiring-1*
begin

definition *irreducible* :: *'a* ⇒ *bool* **where**
irreducible p ⟷ *p* ≠ *0* ∧ ¬*p dvd 1* ∧ (∀ *a b*. *p* = *a* ∗ *b* ⟶ *a dvd 1* ∨ *b dvd 1*)

lemma *not-irreducible-zero* [*simp*]: ¬*irreducible 0*
⟨*proof*⟩

lemma *irreducible-not-unit*: *irreducible p* ⟹ ¬*p dvd 1*
⟨*proof*⟩

lemma *not-irreducible-one* [*simp*]: ¬*irreducible 1*
⟨*proof*⟩

lemma *irreducibleI*:
p ≠ *0* ⟹ ¬*p dvd 1* ⟹ (⋀*a b*. *p* = *a* ∗ *b* ⟹ *a dvd 1* ∨ *b dvd 1*) ⟹ *irreducible p*
⟨*proof*⟩

lemma *irreducibleD*: *irreducible p* ⟹ *p* = *a* ∗ *b* ⟹ *a dvd 1* ∨ *b dvd 1*
⟨*proof*⟩

definition *prime-elem* :: *'a* ⇒ *bool* **where**
prime-elem p ⟷ *p* ≠ *0* ∧ ¬*p dvd 1* ∧ (∀ *a b*. *p dvd* (*a* ∗ *b*) ⟶ *p dvd a* ∨ *p dvd b*)

lemma *not-prime-elem-zero* [*simp*]: ¬*prime-elem 0*
⟨*proof*⟩

lemma *prime-elem-not-unit*: *prime-elem p* ⟹ ¬*p dvd 1*
⟨*proof*⟩

lemma *prime-elemI*:
p ≠ *0* ⟹ ¬*p dvd 1* ⟹ (⋀*a b*. *p dvd* (*a* ∗ *b*) ⟹ *p dvd a* ∨ *p dvd b*) ⟹ *prime-elem p*
⟨*proof*⟩

lemma *prime-elem-dvd-multD*:
prime-elem p ⟹ *p dvd* (*a* ∗ *b*) ⟹ *p dvd a* ∨ *p dvd b*
⟨*proof*⟩

lemma *prime-elem-dvd-mult-iff*:
prime-elem p ⟹ *p dvd* (*a* ∗ *b*) ⟷ *p dvd a* ∨ *p dvd b*
⟨*proof*⟩

lemma *not-prime-elem-one* [*simp*]:
  ¬ *prime-elem 1*
  ⟨*proof*⟩

lemma *prime-elem-not-zeroI*:
  assumes *prime-elem p*
  shows *p* ≠ *0*
  ⟨*proof*⟩

lemma *prime-elem-dvd-power*:
  prime-elem p ⟹ *p dvd x* ^ *n* ⟹ *p dvd x*
  ⟨*proof*⟩

lemma *prime-elem-dvd-power-iff*:
  prime-elem p ⟹ *n* > *0* ⟹ *p dvd x* ^ *n* ⟷ *p dvd x*
  ⟨*proof*⟩

lemma *prime-elem-imp-nonzero* [*simp*]:
  ASSUMPTION (*prime-elem x*) ⟹ *x* ≠ *0*
  ⟨*proof*⟩

lemma *prime-elem-imp-not-one* [*simp*]:
  ASSUMPTION (*prime-elem x*) ⟹ *x* ≠ *1*
  ⟨*proof*⟩

end

context *algebraic-semidom*
begin

lemma *prime-elem-imp-irreducible*:
  assumes *prime-elem p*
  shows *irreducible p*
⟨*proof*⟩

lemma (**in** *algebraic-semidom*) *unit-imp-no-irreducible-divisors*:
  assumes *is-unit x irreducible p*
  shows ¬*p dvd x*
⟨*proof*⟩

lemma *unit-imp-no-prime-divisors*:
  assumes *is-unit x prime-elem p*
  shows ¬*p dvd x*
  ⟨*proof*⟩

lemma *prime-elem-mono*:
  assumes *prime-elem p* ¬*q dvd 1 q dvd p*
  shows *prime-elem q*

⟨*proof*⟩

lemma *irreducibleD'*:
assumes *irreducible a b dvd a*
shows *a dvd b ∨ is-unit b*
⟨*proof*⟩

lemma *irreducibleI'*:
assumes *a ≠ 0 ¬is-unit a ⋀b. b dvd a ⟹ a dvd b ∨ is-unit b*
shows *irreducible a*
⟨*proof*⟩

lemma *irreducible-altdef*:
irreducible x ⟷ x ≠ 0 ∧ ¬is-unit x ∧ (∀ b. b dvd x ⟶ x dvd b ∨ is-unit b)
⟨*proof*⟩

lemma *prime-elem-multD*:
assumes *prime-elem (a * b)*
shows *is-unit a ∨ is-unit b*
⟨*proof*⟩

lemma *prime-elemD2*:
assumes *prime-elem p* **and** *a dvd p* **and** *¬ is-unit a*
shows *p dvd a*
⟨*proof*⟩

lemma *prime-elem-dvd-prod-msetE*:
assumes *prime-elem p*
assumes *dvd*: *p dvd prod-mset A*
obtains *a* **where** *a ∈# A* **and** *p dvd a*
⟨*proof*⟩

context
begin

private lemma *prime-elem-powerD*:
assumes *prime-elem (p ^ n)*
shows *prime-elem p ∧ n = 1*
⟨*proof*⟩

lemma *prime-elem-power-iff*:
prime-elem (p ^ n) ⟷ prime-elem p ∧ n = 1
⟨*proof*⟩

end

lemma *irreducible-mult-unit-left*:
*is-unit a ⟹ irreducible (a * p) ⟷ irreducible p*
⟨*proof*⟩

lemma *prime-elem-mult-unit-left*:
is-unit a $\Longrightarrow$ *prime-elem* (*a* $*$ *p*) $\longleftrightarrow$ *prime-elem p*
⟨*proof*⟩

lemma *prime-elem-dvd-cases*:
assumes *pk*: *p*$*$*k dvd m*$*$*n* **and** *p*: *prime-elem p*
shows ($\exists$ *x*. *k dvd x*$*$*n* $\wedge$ *m* = *p*$*$*x*) $\vee$ ($\exists$ *y*. *k dvd m*$*$*y* $\wedge$ *n* = *p*$*$*y*)
⟨*proof*⟩

lemma *prime-elem-power-dvd-prod*:
assumes *pc*: *p* ^ *c dvd m*$*$*n* **and** *p*: *prime-elem p*
shows $\exists$ *a b*. *a*+*b* = *c* $\wedge$ *p* ^ *a dvd m* $\wedge$ *p* ^ *b dvd n*
⟨*proof*⟩

lemma *prime-elem-power-dvd-cases*:
assumes *p* ^ *c dvd m* $*$ *n* **and** *a* + *b* = *Suc c* **and** *prime-elem p*
shows *p* ^ *a dvd m* $\vee$ *p* ^ *b dvd n*
⟨*proof*⟩

lemma *prime-elem-not-unit'* [*simp*]:
ASSUMPTION (*prime-elem x*) $\Longrightarrow$ $\neg$*is-unit x*
⟨*proof*⟩

lemma *prime-elem-dvd-power-iff*:
assumes *prime-elem p*
shows *p dvd a* ^ *n* $\longleftrightarrow$ *p dvd a* $\wedge$ *n* > *0*
⟨*proof*⟩

lemma *prime-power-dvd-multD*:
assumes *prime-elem p*
assumes *p* ^ *n dvd a* $*$ *b* **and** *n* > *0* **and** $\neg$ *p dvd a*
shows *p* ^ *n dvd b*
⟨*proof*⟩

end

46.2 Generalized primes: normalized prime elements

context *normalization-semidom*
begin

lemma *irreducible-normalized-divisors*:
assumes *irreducible x y dvd x normalize y* = *y*
shows *y* = *1* $\vee$ *y* = *normalize x*
⟨*proof*⟩

lemma *irreducible-normalize-iff* [*simp*]: *irreducible* (*normalize x*) = *irreducible x*
⟨*proof*⟩

lemma *prime-elem-normalize-iff* [*simp*]: *prime-elem* (*normalize x*) = *prime-elem x*
⟨*proof*⟩

lemma *prime-elem-associated*:
assumes *prime-elem p* **and** *prime-elem q* **and** *q dvd p*
shows *normalize q* = *normalize p*
⟨*proof*⟩

definition *prime* :: *'a* ⇒ *bool* **where**
prime p ⟷ *prime-elem p* ∧ *normalize p* = *p*

lemma *not-prime-0* [*simp*]: ¬*prime 0* ⟨*proof*⟩

lemma *not-prime-unit*: *is-unit x* ⟹ ¬*prime x*
⟨*proof*⟩

lemma *not-prime-1* [*simp*]: ¬*prime 1* ⟨*proof*⟩

lemma *primeI*: *prime-elem x* ⟹ *normalize x* = *x* ⟹ *prime x*
⟨*proof*⟩

lemma *prime-imp-prime-elem* [*dest*]: *prime p* ⟹ *prime-elem p*
⟨*proof*⟩

lemma *normalize-prime*: *prime p* ⟹ *normalize p* = *p*
⟨*proof*⟩

lemma *prime-normalize-iff* [*simp*]: *prime* (*normalize p*) ⟷ *prime-elem p*
⟨*proof*⟩

lemma *prime-power-iff*:
prime (*p* ^ *n*) ⟷ *prime p* ∧ *n* = *1*
⟨*proof*⟩

lemma *prime-imp-nonzero* [*simp*]:
ASSUMPTION (*prime x*) ⟹ *x* ≠ *0*
⟨*proof*⟩

lemma *prime-imp-not-one* [*simp*]:
ASSUMPTION (*prime x*) ⟹ *x* ≠ *1*
⟨*proof*⟩

lemma *prime-not-unit'* [*simp*]:
ASSUMPTION (*prime x*) ⟹ ¬*is-unit x*
⟨*proof*⟩

lemma *prime-normalize'* [*simp*]: *ASSUMPTION* (*prime x*) ⟹ *normalize x* = *x*

⟨*proof*⟩

lemma *unit-factor-prime*: *prime x* ⟹ *unit-factor x* = *1*
⟨*proof*⟩

lemma *unit-factor-prime'* [*simp*]: *ASSUMPTION* (*prime x*) ⟹ *unit-factor x* = *1*
⟨*proof*⟩

lemma *prime-imp-prime-elem'* [*simp*]: *ASSUMPTION* (*prime x*) ⟹ *prime-elem x*
⟨*proof*⟩

lemma *prime-dvd-multD*: *prime p* ⟹ *p dvd a* ∗ *b* ⟹ *p dvd a* ∨ *p dvd b*
⟨*proof*⟩

lemma *prime-dvd-mult-iff* [*simp*]: *prime p* ⟹ *p dvd a* ∗ *b* ⟷ *p dvd a* ∨ *p dvd b*
⟨*proof*⟩

lemma *prime-dvd-power*:
prime p ⟹ *p dvd x* ^ *n* ⟹ *p dvd x*
⟨*proof*⟩

lemma *prime-dvd-power-iff*:
prime p ⟹ *n* > *0* ⟹ *p dvd x* ^ *n* ⟷ *p dvd x*
⟨*proof*⟩

lemma *prime-dvd-prod-mset-iff*: *prime p* ⟹ *p dvd prod-mset A* ⟷ (∃ *x*. *x* ∈# *A* ∧ *p dvd x*)
⟨*proof*⟩

lemma *primes-dvd-imp-eq*:
assumes *prime p prime q p dvd q*
shows *p* = *q*
⟨*proof*⟩

lemma *prime-dvd-prod-mset-primes-iff*:
assumes *prime p* ⋀*q*. *q* ∈# *A* ⟹ *prime q*
shows *p dvd prod-mset A* ⟷ *p* ∈# *A*
⟨*proof*⟩

lemma *prod-mset-primes-dvd-imp-subset*:
assumes *prod-mset A dvd prod-mset B* ⋀*p*. *p* ∈# *A* ⟹ *prime p* ⋀*p*. *p* ∈# *B* ⟹ *prime p*
shows *A* ⊆# *B*
⟨*proof*⟩

lemma *normalize-prod-mset-primes*:

$(\bigwedge p.\ p \in\# A \Longrightarrow prime\ p) \Longrightarrow normalize\ (prod\text{-}mset\ A) = prod\text{-}mset\ A$
⟨*proof*⟩

lemma *prod-mset-dvd-prod-mset-primes-iff*:
assumes $\bigwedge x.\ x \in\# A \Longrightarrow prime\ x$ $\bigwedge x.\ x \in\# B \Longrightarrow prime\ x$
shows $prod\text{-}mset\ A\ dvd\ prod\text{-}mset\ B \longleftrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *is-unit-prod-mset-primes-iff*:
assumes $\bigwedge x.\ x \in\# A \Longrightarrow prime\ x$
shows $is\text{-}unit\ (prod\text{-}mset\ A) \longleftrightarrow A = \{\#\}$
⟨*proof*⟩

lemma *prod-mset-primes-irreducible-imp-prime*:
assumes *irred*: $irreducible\ (prod\text{-}mset\ A)$
assumes *A*: $\bigwedge x.\ x \in\# A \Longrightarrow prime\ x$
assumes *B*: $\bigwedge x.\ x \in\# B \Longrightarrow prime\ x$
assumes *C*: $\bigwedge x.\ x \in\# C \Longrightarrow prime\ x$
assumes *dvd*: $prod\text{-}mset\ A\ dvd\ prod\text{-}mset\ B * prod\text{-}mset\ C$
shows $prod\text{-}mset\ A\ dvd\ prod\text{-}mset\ B \vee prod\text{-}mset\ A\ dvd\ prod\text{-}mset\ C$
⟨*proof*⟩

lemma *prod-mset-primes-finite-divisor-powers*:
assumes *A*: $\bigwedge x.\ x \in\# A \Longrightarrow prime\ x$
assumes *B*: $\bigwedge x.\ x \in\# B \Longrightarrow prime\ x$
assumes $A \neq \{\#\}$
shows $finite\ \{n.\ prod\text{-}mset\ A\ \hat{}\ n\ dvd\ prod\text{-}mset\ B\}$
⟨*proof*⟩

end

46.3 In a semiring with GCD, each irreducible element is a prime elements

context *semiring-gcd*
begin

lemma *irreducible-imp-prime-elem-gcd*:
assumes $irreducible\ x$
shows $prime\text{-}elem\ x$
⟨*proof*⟩

lemma *prime-elem-imp-coprime*:
assumes $prime\text{-}elem\ p\ \neg p\ dvd\ n$
shows $coprime\ p\ n$
⟨*proof*⟩

lemma *prime-imp-coprime*:
assumes $prime\ p\ \neg p\ dvd\ n$

shows *coprime p n*
⟨*proof*⟩

lemma *prime-elem-imp-power-coprime*:
prime-elem p ⟹ ¬*p dvd a* ⟹ *coprime a* (*p* ^ *m*)
⟨*proof*⟩

lemma *prime-imp-power-coprime*:
prime p ⟹ ¬*p dvd a* ⟹ *coprime a* (*p* ^ *m*)
⟨*proof*⟩

lemma *prime-elem-divprod-pow*:
assumes *p*: *prime-elem p* **and** *ab*: *coprime a b* **and** *pab*: *p*^*n dvd a* ∗ *b*
shows *p*^*n dvd a* ∨ *p*^*n dvd b*
⟨*proof*⟩

lemma *primes-coprime*:
prime p ⟹ *prime q* ⟹ *p* ≠ *q* ⟹ *coprime p q*
⟨*proof*⟩

end

46.4 Factorial semirings: algebraic structures with unique prime factorizations

class *factorial-semiring* = *normalization-semidom* +
assumes *prime-factorization-exists*:
x ≠ *0* ⟹ ∃ *A*. (∀ *x*. *x* ∈# *A* ⟶ *prime-elem x*) ∧ *prod-mset A* = *normalize x*

Alternative characterization

lemma (**in** *normalization-semidom*) *factorial-semiring-altI-aux*:
assumes *finite-divisors*: ⋀*x*. *x* ≠ *0* ⟹ *finite* {*y*. *y dvd x* ∧ *normalize y* = *y*}
assumes *irreducible-imp-prime-elem*: ⋀*x*. *irreducible x* ⟹ *prime-elem x*
assumes *x* ≠ *0*
shows ∃ *A*. (∀ *x*. *x* ∈# *A* ⟶ *prime-elem x*) ∧ *prod-mset A* = *normalize x*
⟨*proof*⟩

lemma *factorial-semiring-altI*:
assumes *finite-divisors*: ⋀*x*::*'a*. *x* ≠ *0* ⟹ *finite* {*y*. *y dvd x* ∧ *normalize y* = *y*}
assumes *irreducible-imp-prime*: ⋀*x*::*'a*. *irreducible x* ⟹ *prime-elem x*
shows *OFCLASS*(*'a* :: *normalization-semidom*, *factorial-semiring-class*)
⟨*proof*⟩

Properties

context *factorial-semiring*
begin

lemma *prime-factorization-exists'*:

assumes $x \neq 0$
obtains A **where** $\bigwedge x.\ x \in\# A \Longrightarrow \mathit{prime}\ x\ \mathit{prod\text{-}mset}\ A = \mathit{normalize}\ x$
⟨*proof*⟩

lemma *irreducible-imp-prime-elem*:
assumes *irreducible x*
shows *prime-elem x*
⟨*proof*⟩

lemma *finite-divisor-powers*:
assumes $y \neq 0$ $\neg$*is-unit x*
shows *finite* $\{n.\ x\ \hat{}\ n\ \mathit{dvd}\ y\}$
⟨*proof*⟩

lemma *finite-prime-divisors*:
assumes $x \neq 0$
shows *finite* $\{p.\ \mathit{prime}\ p \wedge p\ \mathit{dvd}\ x\}$
⟨*proof*⟩

lemma *prime-elem-iff-irreducible*: *prime-elem x* $\longleftrightarrow$ *irreducible x*
⟨*proof*⟩

lemma *prime-divisor-exists*:
assumes $a \neq 0$ $\neg$*is-unit a*
shows $\exists b.\ b\ \mathit{dvd}\ a \wedge \mathit{prime}\ b$
⟨*proof*⟩

lemma *prime-divisors-induct* [*case-names zero unit factor*]:
assumes $P\ 0$ $\bigwedge x.\ \mathit{is\text{-}unit}\ x \Longrightarrow P\ x$ $\bigwedge p\ x.\ \mathit{prime}\ p \Longrightarrow P\ x \Longrightarrow P\ (p * x)$
shows $P\ x$
⟨*proof*⟩

lemma *no-prime-divisors-imp-unit*:
assumes $a \neq 0$ $\bigwedge b.\ b\ \mathit{dvd}\ a \Longrightarrow \mathit{normalize}\ b = b \Longrightarrow \neg\ \mathit{prime\text{-}elem}\ b$
shows *is-unit a*
⟨*proof*⟩

lemma *prime-divisorE*:
assumes $a \neq 0$ **and** $\neg$ *is-unit a*
obtains *p* **where** *prime p* **and** *p dvd a*
⟨*proof*⟩

definition *multiplicity* :: $'a \Rightarrow 'a \Rightarrow$ *nat* **where**
multiplicity p x = (*if finite* $\{n.\ p\ \hat{}\ n\ \mathit{dvd}\ x\}$ *then Max* $\{n.\ p\ \hat{}\ n\ \mathit{dvd}\ x\}$ *else 0*)

lemma *multiplicity-dvd*: $p\ \hat{}$ *multiplicity p x dvd x*
⟨*proof*⟩

lemma *multiplicity-dvd'*: $n \leq \mathit{multiplicity}\ p\ x \Longrightarrow p\ \hat{}\ n\ \mathit{dvd}\ x$

⟨*proof*⟩

context
fixes *x p* :: *'a*
assumes *xp*: *x* ≠ *0* ¬*is-unit p*
begin

lemma *multiplicity-eq-Max*: *multiplicity p x* = *Max* {*n*. *p* ^ *n dvd x*}
⟨*proof*⟩

lemma *multiplicity-geI*:
assumes *p* ^ *n dvd x*
shows *multiplicity p x* ≥ *n*
⟨*proof*⟩

lemma *multiplicity-lessI*:
assumes ¬*p* ^ *n dvd x*
shows *multiplicity p x* < *n*
⟨*proof*⟩

lemma *power-dvd-iff-le-multiplicity*:
p ^ *n dvd x* ⟷ *n* ≤ *multiplicity p x*
⟨*proof*⟩

lemma *multiplicity-eq-zero-iff*:
shows *multiplicity p x* = *0* ⟷ ¬*p dvd x*
⟨*proof*⟩

lemma *multiplicity-gt-zero-iff*:
shows *multiplicity p x* > *0* ⟷ *p dvd x*
⟨*proof*⟩

lemma *multiplicity-decompose*:
¬*p dvd* (*x div p* ^ *multiplicity p x*)
⟨*proof*⟩

lemma *multiplicity-decompose'*:
obtains *y* **where** *x* = *p* ^ *multiplicity p x* ∗ *y* ¬*p dvd y*
⟨*proof*⟩

end

lemma *multiplicity-zero* [*simp*]: *multiplicity p 0* = *0*
⟨*proof*⟩

lemma *prime-elem-multiplicity-eq-zero-iff*:
prime-elem p ⟹ *x* ≠ *0* ⟹ *multiplicity p x* = *0* ⟷ ¬*p dvd x*
⟨*proof*⟩

lemma *prime-multiplicity-other*:
assumes *prime p prime q p* $\neq$ *q*
shows *multiplicity p q = 0*
⟨*proof*⟩

lemma *prime-multiplicity-gt-zero-iff*:
prime-elem p $\Longrightarrow$ *x* $\neq$ *0* $\Longrightarrow$ *multiplicity p x > 0* $\longleftrightarrow$ *p dvd x*
⟨*proof*⟩

lemma *multiplicity-unit-left*: *is-unit p* $\Longrightarrow$ *multiplicity p x = 0*
⟨*proof*⟩

lemma *multiplicity-unit-right*:
assumes *is-unit x*
shows *multiplicity p x = 0*
⟨*proof*⟩

lemma *multiplicity-one* [*simp*]: *multiplicity p 1 = 0*
⟨*proof*⟩

lemma *multiplicity-eqI*:
assumes *p* ^ *n dvd x* ¬*p* ^ *Suc n dvd x*
shows *multiplicity p x = n*
⟨*proof*⟩

context
fixes *x p* :: *'a*
assumes *xp*: *x* $\neq$ *0* ¬*is-unit p*
begin

lemma *multiplicity-times-same*:
assumes *p* $\neq$ *0*
shows *multiplicity p* (*p* * *x*) = *Suc* (*multiplicity p x*)
⟨*proof*⟩

end

lemma *multiplicity-same-power'*: *multiplicity p* (*p* ^ *n*) = (*if p = 0* ∨ *is-unit p then 0 else n*)
⟨*proof*⟩

lemma *multiplicity-same-power*:
p $\neq$ *0* $\Longrightarrow$ ¬*is-unit p* $\Longrightarrow$ *multiplicity p* (*p* ^ *n*) = *n*
⟨*proof*⟩

lemma *multiplicity-prime-elem-times-other*:
assumes *prime-elem p* ¬*p dvd q*
shows *multiplicity p* (*q* * *x*) = *multiplicity p x*

⟨*proof*⟩

lemma *multiplicity-self*:
  assumes $p \neq 0$ $\neg$*is-unit p*
  shows *multiplicity p p = 1*
⟨*proof*⟩

lemma *multiplicity-times-unit-left*:
  assumes *is-unit c*
  shows *multiplicity* $(c * p)$ *x = multiplicity p x*
⟨*proof*⟩

lemma *multiplicity-times-unit-right*:
  assumes *is-unit c*
  shows *multiplicity p* $(c * x)$ *= multiplicity p x*
⟨*proof*⟩

lemma *multiplicity-normalize-left* [*simp*]:
  multiplicity (*normalize p*) *x = multiplicity p x*
⟨*proof*⟩

lemma *multiplicity-normalize-right* [*simp*]:
  multiplicity p (*normalize x*) *= multiplicity p x*
⟨*proof*⟩

lemma *multiplicity-prime* [*simp*]: *prime-elem p* $\Longrightarrow$ *multiplicity p p = 1*
  ⟨*proof*⟩

lemma *multiplicity-prime-power* [*simp*]: *prime-elem p* $\Longrightarrow$ *multiplicity p* $(p \hat{} n)$
= n
  ⟨*proof*⟩

lift-definition *prime-factorization* :: *'a* $\Rightarrow$ *'a multiset* **is**
  λ*x p. if prime p then multiplicity p x else 0*
  ⟨*proof*⟩

abbreviation *prime-factors* :: *'a* $\Rightarrow$ *'a set* **where**
  prime-factors a $\equiv$ *set-mset* (*prime-factorization a*)

lemma *count-prime-factorization-nonprime*:
  $\neg$*prime p* $\Longrightarrow$ *count* (*prime-factorization x*) *p = 0*
  ⟨*proof*⟩

lemma *count-prime-factorization-prime*:
  prime p $\Longrightarrow$ *count* (*prime-factorization x*) *p = multiplicity p x*
  ⟨*proof*⟩

lemma *count-prime-factorization*:
  count (*prime-factorization x*) *p* = (*if prime p then multiplicity p x else 0*)

⟨*proof*⟩

lemma *dvd-imp-multiplicity-le*:
assumes *a dvd b b* ≠ *0*
shows *multiplicity p a* ≤ *multiplicity p b*
⟨*proof*⟩

lemma *prime-factorization-0* [*simp*]: *prime-factorization 0* = {#}
⟨*proof*⟩

lemma *prime-factorization-empty-iff*:
prime-factorization x = {#} ⟷ *x* = *0* ∨ *is-unit x*
⟨*proof*⟩

lemma *prime-factorization-unit*:
assumes *is-unit x*
shows *prime-factorization x* = {#}
⟨*proof*⟩

lemma *prime-factorization-1* [*simp*]: *prime-factorization 1* = {#}
⟨*proof*⟩

lemma *prime-factorization-times-prime*:
assumes *x* ≠ *0 prime p*
shows *prime-factorization* (*p* ∗ *x*) = {#*p*#} + *prime-factorization x*
⟨*proof*⟩

lemma *prod-mset-prime-factorization*:
assumes *x* ≠ *0*
shows *prod-mset* (*prime-factorization x*) = *normalize x*
⟨*proof*⟩

lemma *in-prime-factors-iff*:
p ∈ *prime-factors x* ⟷ *x* ≠ *0* ∧ *p dvd x* ∧ *prime p*
⟨*proof*⟩

lemma *in-prime-factors-imp-prime* [*intro*]:
p ∈ *prime-factors x* ⟹ *prime p*
⟨*proof*⟩

lemma *in-prime-factors-imp-dvd* [*dest*]:
p ∈ *prime-factors x* ⟹ *p dvd x*
⟨*proof*⟩

lemma *prime-factorsI*:
x ≠ *0* ⟹ *prime p* ⟹ *p dvd x* ⟹ *p* ∈ *prime-factors x*
⟨*proof*⟩

lemma *prime-factors-dvd*:

$x \neq 0 \Longrightarrow$ *prime-factors* $x = \{p.$ *prime* $p \wedge p$ *dvd* $x\}$
⟨*proof*⟩

lemma *prime-factors-multiplicity*:
prime-factors $n = \{p.$ *prime* $p \wedge$ *multiplicity* $p\ n > 0\}$
⟨*proof*⟩

lemma *prime-factorization-prime*:
assumes *prime* p
shows *prime-factorization* $p = \{\#p\#\}$
⟨*proof*⟩

lemma *prime-factorization-prod-mset-primes*:
assumes $\bigwedge p.\ p \in\# A \Longrightarrow$ *prime* p
shows *prime-factorization* (*prod-mset* A) $= A$
⟨*proof*⟩

lemma *prime-factorization-cong*:
normalize $x =$ *normalize* $y \Longrightarrow$ *prime-factorization* $x =$ *prime-factorization* y
⟨*proof*⟩

lemma *prime-factorization-unique*:
assumes $x \neq 0\ y \neq 0$
shows *prime-factorization* $x =$ *prime-factorization* $y \longleftrightarrow$ *normalize* $x =$ *normalize* y
⟨*proof*⟩

lemma *prime-factorization-mult*:
assumes $x \neq 0\ y \neq 0$
shows *prime-factorization* $(x * y) =$ *prime-factorization* $x +$ *prime-factorization* y
⟨*proof*⟩

lemma *prime-elem-multiplicity-mult-distrib*:
assumes *prime-elem* $p\ x \neq 0\ y \neq 0$
shows *multiplicity* $p\ (x * y) =$ *multiplicity* $p\ x +$ *multiplicity* $p\ y$
⟨*proof*⟩

lemma *prime-elem-multiplicity-prod-mset-distrib*:
assumes *prime-elem* $p\ 0 \notin\# A$
shows *multiplicity* p (*prod-mset* A) $=$ *sum-mset* (*image-mset* (*multiplicity* p) A)
⟨*proof*⟩

lemma *prime-elem-multiplicity-power-distrib*:
assumes *prime-elem* $p\ x \neq 0$
shows *multiplicity* $p\ (x\ \hat{}\ n) = n *$ *multiplicity* $p\ x$
⟨*proof*⟩

lemma *prime-elem-multiplicity-prod-distrib*:
assumes *prime-elem p 0* $\notin$ *f ' A finite A*
shows *multiplicity p* (*prod f A*) = ($\sum$ *x*$\in$*A*. *multiplicity p* (*f x*))
⟨*proof*⟩

lemma *multiplicity-distinct-prime-power*:
prime p $\Longrightarrow$ *prime q* $\Longrightarrow$ *p* $\neq$ *q* $\Longrightarrow$ *multiplicity p* (*q* ^ *n*) = *0*
⟨*proof*⟩

lemma *prime-factorization-prime-power*:
prime p $\Longrightarrow$ *prime-factorization* (*p* ^ *n*) = *replicate-mset n p*
⟨*proof*⟩

lemma *prime-decomposition*: *unit-factor x* $*$ *prod-mset* (*prime-factorization x*) = *x*
⟨*proof*⟩

lemma *prime-factorization-subset-iff-dvd*:
assumes [*simp*]: *x* $\neq$ *0 y* $\neq$ *0*
shows *prime-factorization x* $\subseteq\#$ *prime-factorization y* $\longleftrightarrow$ *x dvd y*
⟨*proof*⟩

lemma *prime-factorization-subset-imp-dvd*:
x $\neq$ *0* $\Longrightarrow$ (*prime-factorization x* $\subseteq\#$ *prime-factorization y*) $\Longrightarrow$ *x dvd y*
⟨*proof*⟩

lemma *prime-factorization-divide*:
assumes *b dvd a*
shows *prime-factorization* (*a div b*) = *prime-factorization a* − *prime-factorization b*
⟨*proof*⟩

lemma *zero-not-in-prime-factors* [*simp*]: *0* $\notin$ *prime-factors x*
⟨*proof*⟩

lemma *prime-prime-factors*:
prime p $\Longrightarrow$ *prime-factors p* = {*p*}
⟨*proof*⟩

lemma *prod-prime-factors*:
assumes *x* $\neq$ *0*
shows ($\prod$ *p* $\in$ *prime-factors x*. *p* ^ *multiplicity p x*) = *normalize x*
⟨*proof*⟩

lemma *prime-factorization-unique''*:
assumes *S-eq*: *S* = {*p*. *0* < *f p*}
and *finite S*
and *S*: $\forall$ *p*$\in$*S*. *prime p normalize n* = ($\prod$ *p*$\in$*S*. *p* ^ *f p*)
shows *S* = *prime-factors n* $\wedge$ ($\forall$ *p*. *prime p* $\longrightarrow$ *f p* = *multiplicity p n*)

⟨*proof*⟩

lemma *prime-factors-product*:
  $x \neq 0 \Longrightarrow y \neq 0 \Longrightarrow$ *prime-factors* $(x * y) =$ *prime-factors* $x \cup$ *prime-factors* y
  ⟨*proof*⟩

lemma *dvd-prime-factors* [*intro*]:
  $y \neq 0 \Longrightarrow x$ *dvd* $y \Longrightarrow$ *prime-factors* $x \subseteq$ *prime-factors* y
  ⟨*proof*⟩

lemma *multiplicity-le-imp-dvd*:
  assumes $x \neq 0$ $\bigwedge p.$ *prime* $p \Longrightarrow$ *multiplicity* p $x \leq$ *multiplicity* p y
  shows x *dvd* y
⟨*proof*⟩

lemma *dvd-multiplicity-eq*:
  $x \neq 0 \Longrightarrow y \neq 0 \Longrightarrow x$ *dvd* $y \longleftrightarrow (\forall p.$ *multiplicity* p $x \leq$ *multiplicity* p $y)$
  ⟨*proof*⟩

lemma *multiplicity-eq-imp-eq*:
  assumes $x \neq 0$ $y \neq 0$
  assumes $\bigwedge p.$ *prime* $p \Longrightarrow$ *multiplicity* p $x =$ *multiplicity* p y
  shows *normalize* $x =$ *normalize* y
  ⟨*proof*⟩

lemma *prime-factorization-unique'*:
  assumes $\forall p \in\# M.$ *prime* p $\forall p \in\# N.$ *prime* p $(\prod i \in\# M.\ i) = (\prod i \in\# N.\ i)$
  shows $M = N$
⟨*proof*⟩

lemma *multiplicity-cong*:
  $(\bigwedge r.\ p \hat{}\ r$ *dvd* $a \longleftrightarrow p \hat{}\ r$ *dvd* $b) \Longrightarrow$ *multiplicity* p $a =$ *multiplicity* p b
  ⟨*proof*⟩

lemma *not-dvd-imp-multiplicity-0*:
  assumes $\neg p$ *dvd* x
  shows *multiplicity* p $x = 0$
⟨*proof*⟩

46.5 GCD and LCM computation with unique factorizations

definition *gcd-factorial* a $b =$ (*if* $a = 0$ *then normalize* b
    else if $b = 0$ *then normalize* a
    else prod-mset (*prime-factorization* $a \cap\#$ *prime-factorization* b))

definition *lcm-factorial* a $b =$ (*if* $a = 0 \vee b = 0$ *then* 0
    else prod-mset (*prime-factorization* $a \cup\#$ *prime-factorization* b))

definition *Gcd-factorial A* =
(*if A* ⊆ {*0*} *then 0 else prod-mset* (*Inf* (*prime-factorization* ‘ (*A* − {*0*}))))

definition *Lcm-factorial A* =
(*if A* = {} *then 1*
else if 0 ∉ *A* ∧ *subset-mset.bdd-above* (*prime-factorization* ‘ (*A* − {*0*})) *then*
prod-mset (*Sup* (*prime-factorization* ‘ *A*))
else
0)

lemma *prime-factorization-gcd-factorial*:
assumes [*simp*]: *a* ≠ *0 b* ≠ *0*
shows *prime-factorization* (*gcd-factorial a b*) = *prime-factorization a* ∩# *prime-factorization b*
⟨*proof*⟩

lemma *prime-factorization-lcm-factorial*:
assumes [*simp*]: *a* ≠ *0 b* ≠ *0*
shows *prime-factorization* (*lcm-factorial a b*) = *prime-factorization a* ∪# *prime-factorization b*
⟨*proof*⟩

lemma *prime-factorization-Gcd-factorial*:
assumes ¬*A* ⊆ {*0*}
shows *prime-factorization* (*Gcd-factorial A*) = *Inf* (*prime-factorization* ‘ (*A* − {*0*}))
⟨*proof*⟩

lemma *prime-factorization-Lcm-factorial*:
assumes *0* ∉ *A subset-mset.bdd-above* (*prime-factorization* ‘ *A*)
shows *prime-factorization* (*Lcm-factorial A*) = *Sup* (*prime-factorization* ‘ *A*)
⟨*proof*⟩

lemma *gcd-factorial-commute*: *gcd-factorial a b* = *gcd-factorial b a*
⟨*proof*⟩

lemma *gcd-factorial-dvd1*: *gcd-factorial a b dvd a*
⟨*proof*⟩

lemma *gcd-factorial-dvd2*: *gcd-factorial a b dvd b*
⟨*proof*⟩

lemma *normalize-gcd-factorial*: *normalize* (*gcd-factorial a b*) = *gcd-factorial a b*
⟨*proof*⟩

lemma *gcd-factorial-greatest*: *c dvd gcd-factorial a b* **if** *c dvd a c dvd b* **for** *a b c*
⟨*proof*⟩

lemma *lcm-factorial-gcd-factorial*:
lcm-factorial a b = *normalize* (*a* ∗ *b*) *div gcd-factorial a b* **for** *a b*
⟨*proof*⟩

lemma *normalize-Gcd-factorial*:
normalize (*Gcd-factorial A*) = *Gcd-factorial A*
⟨*proof*⟩

lemma *Gcd-factorial-eq-0-iff*:
Gcd-factorial A = *0* ⟷ *A* ⊆ {*0*}
⟨*proof*⟩

lemma *Gcd-factorial-dvd*:
assumes *x* ∈ *A*
shows *Gcd-factorial A dvd x*
⟨*proof*⟩

lemma *Gcd-factorial-greatest*:
assumes ⋀*y*. *y* ∈ *A* ⟹ *x dvd y*
shows *x dvd Gcd-factorial A*
⟨*proof*⟩

lemma *normalize-Lcm-factorial*:
normalize (*Lcm-factorial A*) = *Lcm-factorial A*
⟨*proof*⟩

lemma *Lcm-factorial-eq-0-iff*:
Lcm-factorial A = *0* ⟷ *0* ∈ *A* ∨ ¬*subset-mset.bdd-above* (*prime-factorization*
‘ *A*)
⟨*proof*⟩

lemma *dvd-Lcm-factorial*:
assumes *x* ∈ *A*
shows *x dvd Lcm-factorial A*
⟨*proof*⟩

lemma *Lcm-factorial-least*:
assumes ⋀*y*. *y* ∈ *A* ⟹ *y dvd x*
shows *Lcm-factorial A dvd x*
⟨*proof*⟩

lemmas *gcd-lcm-factorial* =
gcd-factorial-dvd1 gcd-factorial-dvd2 gcd-factorial-greatest
normalize-gcd-factorial lcm-factorial-gcd-factorial
normalize-Gcd-factorial Gcd-factorial-dvd Gcd-factorial-greatest
normalize-Lcm-factorial dvd-Lcm-factorial Lcm-factorial-least

end

class *factorial-semiring-gcd* = *factorial-semiring* + *gcd* + *Gcd* +
  assumes *gcd-eq-gcd-factorial*: *gcd a b* = *gcd-factorial a b*
  and *lcm-eq-lcm-factorial*: *lcm a b* = *lcm-factorial a b*
  and *Gcd-eq-Gcd-factorial*: *Gcd A* = *Gcd-factorial A*
  and *Lcm-eq-Lcm-factorial*: *Lcm A* = *Lcm-factorial A*
begin

lemma *prime-factorization-gcd*:
  assumes [*simp*]: *a* ≠ *0 b* ≠ *0*
  shows *prime-factorization* (*gcd a b*) = *prime-factorization a* ∩# *prime-factorization b*
  ⟨*proof*⟩

lemma *prime-factorization-lcm*:
  assumes [*simp*]: *a* ≠ *0 b* ≠ *0*
  shows *prime-factorization* (*lcm a b*) = *prime-factorization a* ∪# *prime-factorization b*
  ⟨*proof*⟩

lemma *prime-factorization-Gcd*:
  assumes *Gcd A* ≠ *0*
  shows *prime-factorization* (*Gcd A*) = *Inf* (*prime-factorization* ‘ (*A* − {*0*}))
  ⟨*proof*⟩

lemma *prime-factorization-Lcm*:
  assumes *Lcm A* ≠ *0*
  shows *prime-factorization* (*Lcm A*) = *Sup* (*prime-factorization* ‘ *A*)
  ⟨*proof*⟩

subclass *semiring-gcd*
  ⟨*proof*⟩

subclass *semiring-Gcd*
  ⟨*proof*⟩

lemma
  assumes *x* ≠ *0 y* ≠ *0*
  shows *gcd-eq-factorial′*:
    gcd x y = (∏ *p* ∈ *prime-factors x* ∩ *prime-factors y*.
      p ^ *min* (*multiplicity p x*) (*multiplicity p y*)) (**is** - = *?rhs1*)
   and *lcm-eq-factorial′*:
    lcm x y = (∏ *p* ∈ *prime-factors x* ∪ *prime-factors y*.
      p ^ *max* (*multiplicity p x*) (*multiplicity p y*)) (**is** - = *?rhs2*)
⟨*proof*⟩

lemma
  assumes *x* ≠ *0 y* ≠ *0 prime p*
  shows *multiplicity-gcd*: *multiplicity p* (*gcd x y*) = *min* (*multiplicity p x*)

(*multiplicity p y*)
and *multiplicity-lcm*: *multiplicity p* (*lcm x y*) = *max* (*multiplicity p x*) (*multiplicity p y*)
⟨*proof*⟩

lemma *gcd-lcm-distrib*:
gcd x (*lcm y z*) = *lcm* (*gcd x y*) (*gcd x z*)
⟨*proof*⟩

lemma *lcm-gcd-distrib*:
lcm x (*gcd y z*) = *gcd* (*lcm x y*) (*lcm x z*)
⟨*proof*⟩

end

class *factorial-ring-gcd* = *factorial-semiring-gcd* + *idom*
begin

subclass *ring-gcd* ⟨*proof*⟩

subclass *idom-divide* ⟨*proof*⟩

end

end

47 Abstract euclidean algorithm

theory *Euclidean-Algorithm*
imports ~~/*src*/*HOL*/*GCD Factorial-Ring*
begin

A Euclidean semiring is a semiring upon which the Euclidean algorithm can be implemented. It must provide:

- division with remainder
- a size function such that *size* (*a mod b*) < *size b* for any $b \neq (0::'a)$

The existence of these functions makes it possible to derive gcd and lcm functions for any Euclidean semiring.

class *euclidean-semiring* = *semiring-modulo* + *normalization-semidom* +
fixes *euclidean-size* :: $'a \Rightarrow nat$
assumes *size-0* [*simp*]: *euclidean-size 0* = *0*
assumes *mod-size-less*:
$b \neq 0 \Longrightarrow$ *euclidean-size* (*a mod b*) < *euclidean-size b*
assumes *size-mult-mono*:
$b \neq 0 \Longrightarrow$ *euclidean-size a* ≤ *euclidean-size* (*a* ∗ *b*)

begin

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

lemma *dvd-mod-iff*:
assumes *k dvd n*
shows *(k dvd m mod n) = (k dvd m)*
⟨*proof*⟩

lemma *mod-0-imp-dvd*:
assumes *a mod b = 0*
shows *b dvd a*
⟨*proof*⟩

lemma *euclidean-size-normalize* [*simp*]:
euclidean-size (normalize a) = euclidean-size a
⟨*proof*⟩

lemma *euclidean-division*:
fixes *a* :: *'a* **and** *b* :: *'a*
assumes *b ≠ 0*
obtains *s* **and** *t* **where** *a = s * b + t*
and *euclidean-size t < euclidean-size b*
⟨*proof*⟩

lemma *dvd-euclidean-size-eq-imp-dvd*:
assumes *a ≠ 0* **and** *b-dvd-a*: *b dvd a* **and** *size-eq*: *euclidean-size a = euclidean-size b*
shows *a dvd b*
⟨*proof*⟩

lemma *size-mult-mono'*: *b ≠ 0 ⟹ euclidean-size a ≤ euclidean-size (b * a)*
⟨*proof*⟩

lemma *euclidean-size-times-unit*:
assumes *is-unit a*
shows *euclidean-size (a * b) = euclidean-size b*
⟨*proof*⟩

lemma *euclidean-size-unit*: *is-unit a ⟹ euclidean-size a = euclidean-size 1*
⟨*proof*⟩

lemma *unit-iff-euclidean-size*:
is-unit a ⟷ euclidean-size a = euclidean-size 1 ∧ a ≠ 0
⟨*proof*⟩

lemma *euclidean-size-times-nonunit*:
assumes *a ≠ 0 b ≠ 0 ¬is-unit a*

shows *euclidean-size b < euclidean-size (a * b)*
⟨*proof*⟩

lemma *dvd-imp-size-le*:
assumes *a dvd b b ≠ 0*
shows *euclidean-size a ≤ euclidean-size b*
⟨*proof*⟩

lemma *dvd-proper-imp-size-less*:
assumes *a dvd b ¬b dvd a b ≠ 0*
shows *euclidean-size a < euclidean-size b*
⟨*proof*⟩

function *gcd-eucl* :: *'a ⇒ 'a ⇒ 'a*
where
gcd-eucl a b = (if b = 0 then normalize a else gcd-eucl b (a mod b))
⟨*proof*⟩
termination
⟨*proof*⟩

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

lemma *gcd-eucl-induct* [*case-names zero mod*]:
assumes *H1*: *⋀b. P b 0*
and *H2*: *⋀a b. b ≠ 0 ⟹ P b (a mod b) ⟹ P a b*
shows *P a b*
⟨*proof*⟩

definition *lcm-eucl* :: *'a ⇒ 'a ⇒ 'a*
where
*lcm-eucl a b = normalize (a * b) div gcd-eucl a b*

definition *Lcm-eucl* :: *'a set ⇒ 'a* — Somewhat complicated definition of Lcm that has the advantage of working for infinite sets as well
where
Lcm-eucl A = (if ∃ l. l ≠ 0 ∧ (∀ a∈A. a dvd l) then
let l = SOME l. l ≠ 0 ∧ (∀ a∈A. a dvd l) ∧ euclidean-size l =
(LEAST n. ∃ l. l ≠ 0 ∧ (∀ a∈A. a dvd l) ∧ euclidean-size l = n)
in normalize l
else 0)

definition *Gcd-eucl* :: *'a set ⇒ 'a*
where
Gcd-eucl A = Lcm-eucl {d. ∀ a∈A. d dvd a}

declare *Lcm-eucl-def Gcd-eucl-def* [*code del*]

lemma *gcd-eucl-0*:
gcd-eucl a 0 = normalize a

⟨*proof*⟩

lemma *gcd-eucl-0-left*:
gcd-eucl 0 a = normalize a
⟨*proof*⟩

lemma *gcd-eucl-non-0*:
b ≠ 0 ⟹ gcd-eucl a b = gcd-eucl b (a mod b)
⟨*proof*⟩

lemma *gcd-eucl-dvd1* [*iff*]: *gcd-eucl a b dvd a*
and *gcd-eucl-dvd2* [*iff*]: *gcd-eucl a b dvd b*
⟨*proof*⟩

lemma *normalize-gcd-eucl* [*simp*]:
normalize (gcd-eucl a b) = gcd-eucl a b
⟨*proof*⟩

lemma *gcd-eucl-greatest*:
fixes *k a b* :: *'a*
shows *k dvd a ⟹ k dvd b ⟹ k dvd gcd-eucl a b*
⟨*proof*⟩

lemma *gcd-euclI*:
fixes *gcd* :: *'a ⇒ 'a ⇒ 'a*
assumes *d dvd a d dvd b normalize d = d*
⋀k. k dvd a ⟹ k dvd b ⟹ k dvd d
shows *gcd-eucl a b = d*
⟨*proof*⟩

lemma *eq-gcd-euclI*:
fixes *gcd* :: *'a ⇒ 'a ⇒ 'a*
assumes *⋀a b. gcd a b dvd a ⋀a b. gcd a b dvd b ⋀a b. normalize (gcd a b) = gcd a b*
⋀a b k. k dvd a ⟹ k dvd b ⟹ k dvd gcd a b
shows *gcd = gcd-eucl*
⟨*proof*⟩

lemma *gcd-eucl-zero* [*simp*]:
gcd-eucl a b = 0 ⟷ a = 0 ∧ b = 0
⟨*proof*⟩

lemma *dvd-Lcm-eucl* [*simp*]: *a ∈ A ⟹ a dvd Lcm-eucl A*
and *Lcm-eucl-least*: *(⋀a. a ∈ A ⟹ a dvd b) ⟹ Lcm-eucl A dvd b*
and *unit-factor-Lcm-eucl* [*simp*]:
unit-factor (Lcm-eucl A) = (if Lcm-eucl A = 0 then 0 else 1)
⟨*proof*⟩

lemma *normalize-Lcm-eucl* [*simp*]:
normalize (*Lcm-eucl A*) = *Lcm-eucl A*
⟨*proof*⟩

lemma *eq-Lcm-euclI*:
fixes *lcm* :: *'a set* ⇒ *'a*
assumes ⋀*A a*. *a* ∈ *A* ⟹ *a dvd lcm A* **and** ⋀*A c*. (⋀*a*. *a* ∈ *A* ⟹ *a dvd c*) ⟹ *lcm A dvd c*
⋀*A*. *normalize* (*lcm A*) = *lcm A* **shows** *lcm* = *Lcm-eucl*
⟨*proof*⟩

lemma *Gcd-eucl-dvd*: *a* ∈ *A* ⟹ *Gcd-eucl A dvd a*
⟨*proof*⟩

lemma *Gcd-eucl-greatest*: (⋀*x*. *x* ∈ *A* ⟹ *d dvd x*) ⟹ *d dvd Gcd-eucl A*
⟨*proof*⟩

lemma *normalize-Gcd-eucl* [*simp*]: *normalize* (*Gcd-eucl A*) = *Gcd-eucl A*
⟨*proof*⟩

lemma *Lcm-euclI*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *x dvd d* ⋀*d'*. (⋀*x*. *x* ∈ *A* ⟹ *x dvd d'*) ⟹ *d dvd d'* *normalize d* = *d*
shows *Lcm-eucl A* = *d*
⟨*proof*⟩

lemma *Gcd-euclI*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *d dvd x* ⋀*d'*. (⋀*x*. *x* ∈ *A* ⟹ *d' dvd x*) ⟹ *d' dvd d* *normalize d* = *d*
shows *Gcd-eucl A* = *d*
⟨*proof*⟩

lemmas *lcm-gcd-eucl-facts* =
gcd-eucl-dvd1 gcd-eucl-dvd2 gcd-eucl-greatest normalize-gcd-eucl lcm-eucl-def
Gcd-eucl-def Gcd-eucl-dvd Gcd-eucl-greatest normalize-Gcd-eucl
dvd-Lcm-eucl Lcm-eucl-least normalize-Lcm-eucl

lemma *normalized-factors-product*:
{*p*. *p dvd a* ∗ *b* ∧ *normalize p* = *p*} =
(λ(*x*,*y*). *x* ∗ *y*) ‘ ({*p*. *p dvd a* ∧ *normalize p* = *p*} × {*p*. *p dvd b* ∧ *normalize p* = *p*})
⟨*proof*⟩

subclass *factorial-semiring*
⟨*proof*⟩

lemma *gcd-eucl-eq-gcd-factorial*: *gcd-eucl* = *gcd-factorial*
⟨*proof*⟩

lemma *lcm-eucl-eq-lcm-factorial*: *lcm-eucl* = *lcm-factorial*
⟨*proof*⟩

lemma *Gcd-eucl-eq-Gcd-factorial*: *Gcd-eucl* = *Gcd-factorial*
⟨*proof*⟩

lemma *Lcm-eucl-eq-Lcm-factorial*: *Lcm-eucl* = *Lcm-factorial*
⟨*proof*⟩

lemmas *eucl-eq-factorial* =
gcd-eucl-eq-gcd-factorial lcm-eucl-eq-lcm-factorial
Gcd-eucl-eq-Gcd-factorial Lcm-eucl-eq-Lcm-factorial

end

class *euclidean-ring* = *euclidean-semiring* + *idom*
begin

function *euclid-ext-aux* :: *'a* ⇒ - **where**
euclid-ext-aux r' r s' s t' t = (
if r = *0 then let c* = *1 div unit-factor r' in* (*s'* ∗ *c*, *t'* ∗ *c*, *normalize r'*)
else let q = *r' div r*
in euclid-ext-aux r (*r' mod r*) *s* (*s'* − *q* ∗ *s*) *t* (*t'* − *q* ∗ *t*))
⟨*proof*⟩
termination ⟨*proof*⟩

declare *euclid-ext-aux.simps* [*simp del*]

lemma *euclid-ext-aux-correct*:
assumes *gcd-eucl r' r* = *gcd-eucl a b*
assumes *s'* ∗ *a* + *t'* ∗ *b* = *r'*
assumes *s* ∗ *a* + *t* ∗ *b* = *r*
shows *case euclid-ext-aux r' r s' s t' t of* (*x*,*y*,*c*) ⇒
x ∗ *a* + *y* ∗ *b* = *c* ∧ *c* = *gcd-eucl a b* (**is** *?P* (*euclid-ext-aux r' r s' s t' t*))
⟨*proof*⟩

definition *euclid-ext* **where**
euclid-ext a b = *euclid-ext-aux a b 1 0 0 1*

lemma *euclid-ext-0*:
euclid-ext a 0 = (*1 div unit-factor a*, *0*, *normalize a*)
⟨*proof*⟩

lemma *euclid-ext-left-0*:
euclid-ext 0 a = (*0*, *1 div unit-factor a*, *normalize a*)
⟨*proof*⟩

lemma *euclid-ext-correct'*:
case euclid-ext a b of $(x,y,c) \Rightarrow x * a + y * b = c \wedge c = gcd\text{-}eucl\ a\ b$
⟨*proof*⟩

lemma *euclid-ext-gcd-eucl*:
$(case\ euclid\text{-}ext\ a\ b\ of\ (x,y,c) \Rightarrow c) = gcd\text{-}eucl\ a\ b$
⟨*proof*⟩

definition *euclid-ext'* **where**
$euclid\text{-}ext'\ a\ b = (case\ euclid\text{-}ext\ a\ b\ of\ (x,\ y,\ \text{-}) \Rightarrow (x,\ y))$

lemma *euclid-ext'-correct'*:
case euclid-ext' a b of $(x,y) \Rightarrow x * a + y * b = gcd\text{-}eucl\ a\ b$
⟨*proof*⟩

lemma *euclid-ext'-0*: $euclid\text{-}ext'\ a\ 0 = (1\ div\ unit\text{-}factor\ a,\ 0)$
⟨*proof*⟩

lemma *euclid-ext'-left-0*: $euclid\text{-}ext'\ 0\ a = (0,\ 1\ div\ unit\text{-}factor\ a)$
⟨*proof*⟩

end

class *euclidean-semiring-gcd* = *euclidean-semiring* + *gcd* + *Gcd* +
assumes *gcd-gcd-eucl*: *gcd* = *gcd-eucl* **and** *lcm-lcm-eucl*: *lcm* = *lcm-eucl*
assumes *Gcd-Gcd-eucl*: *Gcd* = *Gcd-eucl* **and** *Lcm-Lcm-eucl*: *Lcm* = *Lcm-eucl*
begin

subclass *semiring-gcd*
⟨*proof*⟩

subclass *semiring-Gcd*
⟨*proof*⟩

subclass *factorial-semiring-gcd*
⟨*proof*⟩

lemma *gcd-non-0*:
$b \neq 0 \Longrightarrow gcd\ a\ b = gcd\ b\ (a\ mod\ b)$
⟨*proof*⟩

lemmas *gcd-0* = *gcd-0-right*
lemmas *dvd-gcd-iff* = *gcd-greatest-iff*
lemmas *gcd-greatest-iff* = *dvd-gcd-iff*

lemma *gcd-mod1* [*simp*]:
$gcd\ (a\ mod\ b)\ b = gcd\ a\ b$
⟨*proof*⟩

lemma *gcd-mod2* [*simp*]:
gcd a (*b mod a*) = *gcd a b*
⟨*proof*⟩

lemma *euclidean-size-gcd-le1* [*simp*]:
assumes *a* ≠ *0*
shows *euclidean-size* (*gcd a b*) ≤ *euclidean-size a*
⟨*proof*⟩

lemma *euclidean-size-gcd-le2* [*simp*]:
b ≠ *0* ⟹ *euclidean-size* (*gcd a b*) ≤ *euclidean-size b*
⟨*proof*⟩

lemma *euclidean-size-gcd-less1*:
assumes *a* ≠ *0* **and** ¬*a dvd b*
shows *euclidean-size* (*gcd a b*) < *euclidean-size a*
⟨*proof*⟩

lemma *euclidean-size-gcd-less2*:
assumes *b* ≠ *0* **and** ¬*b dvd a*
shows *euclidean-size* (*gcd a b*) < *euclidean-size b*
⟨*proof*⟩

lemma *euclidean-size-lcm-le1*:
assumes *a* ≠ *0* **and** *b* ≠ *0*
shows *euclidean-size a* ≤ *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *euclidean-size-lcm-le2*:
a ≠ *0* ⟹ *b* ≠ *0* ⟹ *euclidean-size b* ≤ *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *euclidean-size-lcm-less1*:
assumes *b* ≠ *0* **and** ¬*b dvd a*
shows *euclidean-size a* < *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *euclidean-size-lcm-less2*:
assumes *a* ≠ *0* **and** ¬*a dvd b*
shows *euclidean-size b* < *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *Lcm-eucl-set* [*code*]:
Lcm-eucl (*set xs*) = *foldl lcm-eucl 1 xs*
⟨*proof*⟩

lemma *Gcd-eucl-set* [*code*]:
Gcd-eucl (*set xs*) = *foldl gcd-eucl 0 xs*
⟨*proof*⟩

end

A Euclidean ring is a Euclidean semiring with additive inverses. It provides a few more lemmas; in particular, Bezout's lemma holds for any Euclidean ring.

class *euclidean-ring-gcd* = *euclidean-semiring-gcd* + *idom*
begin

subclass *euclidean-ring* ⟨*proof*⟩
subclass *ring-gcd* ⟨*proof*⟩
subclass *factorial-ring-gcd* ⟨*proof*⟩

lemma *euclid-ext-gcd* [*simp*]:
(*case euclid-ext a b of* (-, - , *t*) ⇒ *t*) = *gcd a b*
⟨*proof*⟩

lemma *euclid-ext-gcd'* [*simp*]:
euclid-ext a b = (*r*, *s*, *t*) ⟹ *t* = *gcd a b*
⟨*proof*⟩

lemma *euclid-ext-correct*:
case euclid-ext a b of (*x*,*y*,*c*) ⇒ *x* ∗ *a* + *y* ∗ *b* = *c* ∧ *c* = *gcd a b*
⟨*proof*⟩

lemma *euclid-ext'-correct*:
fst (*euclid-ext' a b*) ∗ *a* + *snd* (*euclid-ext' a b*) ∗ *b* = *gcd a b*
⟨*proof*⟩

lemma *bezout*: ∃ *s t*. *s* ∗ *a* + *t* ∗ *b* = *gcd a b*
⟨*proof*⟩

end

47.1 Typical instances

instantiation *nat* :: *euclidean-semiring*
begin

definition [*simp*]:
euclidean-size-nat = (*id* :: *nat* ⇒ *nat*)

instance ⟨*proof*⟩

end

instantiation *int* :: *euclidean-ring*
begin

definition [*simp*]:
euclidean-size-int = (*nat* ∘ *abs* :: *int* ⇒ *nat*)

instance ⟨*proof*⟩

end

instance *nat* :: *euclidean-semiring-gcd*
⟨*proof*⟩

instance *int* :: *euclidean-ring-gcd*
⟨*proof*⟩

end

48 Primes

theory *Primes*
imports ~~/*src*/*HOL*/*Binomial Euclidean-Algorithm*
begin

declare [[*coercion int*]]
declare [[*coercion-enabled*]]

lemma *prime-elem-nat-iff*:
prime-elem (*n* :: *nat*) ⟷ (*1* < *n* ∧ (∀ *m*. *m dvd n* ⟶ *m* = *1* ∨ *m* = *n*))
⟨*proof*⟩

lemma *prime-nat-iff-prime-elem*: *prime* (*n* :: *nat*) ⟷ *prime-elem n*
⟨*proof*⟩

lemma *prime-nat-iff*:
prime (*n* :: *nat*) ⟷ (*1* < *n* ∧ (∀ *m*. *m dvd n* ⟶ *m* = *1* ∨ *m* = *n*))
⟨*proof*⟩

lemma *prime-elem-int-nat-transfer*: *prime-elem* (*n*::*int*) ⟷ *prime-elem* (*nat* (*abs n*))
⟨*proof*⟩

lemma *prime-elem-nat-int-transfer* [*simp*]: *prime-elem* (*int n*) ⟷ *prime-elem n*
⟨*proof*⟩

lemma *prime-nat-int-transfer* [*simp*]: *prime* (*int n*) ⟷ *prime n*
⟨*proof*⟩

lemma *prime-int-nat-transfer*: *prime* (*n*::*int*) ⟷ *n* ≥ *0* ∧ *prime* (*nat n*)

⟨*proof*⟩

lemma *prime-int-iff*:
prime (*n*::*int*) ⟷ (*1* < *n* ∧ (∀ *m*. *m* ≥ *0* ∧ *m dvd n* ⟶ *m* = *1* ∨ *m* = *n*))
⟨*proof*⟩

lemma *prime-nat-not-dvd*:
assumes *prime p p* > *n n* ≠ (*1*::*nat*)
shows ¬*n dvd p*
⟨*proof*⟩

lemma *prime-int-not-dvd*:
assumes *prime p p* > *n n* > (*1*::*int*)
shows ¬*n dvd p*
⟨*proof*⟩

lemma *prime-odd-nat*: *prime p* ⟹ *p* > (*2*::*nat*) ⟹ *odd p*
⟨*proof*⟩

lemma *prime-odd-int*: *prime p* ⟹ *p* > (*2*::*int*) ⟹ *odd p*
⟨*proof*⟩

lemma *prime-ge-0-int*: *prime p* ⟹ *p* ≥ (*0*::*int*)
⟨*proof*⟩

lemma *prime-gt-0-nat*: *prime p* ⟹ *p* > (*0*::*nat*)
⟨*proof*⟩

lemma *prime-gt-0-int*: *prime p* ⟹ *p* > (*0*::*int*)
⟨*proof*⟩

lemma *prime-ge-1-nat*: *prime p* ⟹ *p* ≥ (*1*::*nat*)
⟨*proof*⟩

lemma *prime-ge-Suc-0-nat*: *prime p* ⟹ *p* ≥ *Suc 0*
⟨*proof*⟩

lemma *prime-ge-1-int*: *prime p* ⟹ *p* ≥ (*1*::*int*)
⟨*proof*⟩

lemma *prime-gt-1-nat*: *prime p* ⟹ *p* > (*1*::*nat*)
⟨*proof*⟩

lemma *prime-gt-Suc-0-nat*: *prime p* ⟹ *p* > *Suc 0*
⟨*proof*⟩

lemma *prime-gt-1-int*: *prime p* ⟹ *p* > (*1*::*int*)
⟨*proof*⟩

lemma *prime-ge-2-nat*: $prime\ p \Longrightarrow p \geq (2::nat)$
⟨*proof*⟩

lemma *prime-ge-2-int*: $prime\ p \Longrightarrow p \geq (2::int)$
⟨*proof*⟩

lemma *prime-int-altdef*:
$prime\ p = (1 < p \wedge (\forall m::int.\ m \geq 0 \longrightarrow m\ dvd\ p \longrightarrow m = 1 \vee m = p))$
⟨*proof*⟩

lemma *not-prime-eq-prod-nat*:
assumes $m > 1\ \neg prime\ (m::nat)$
shows $\exists n\ k.\ n = m * k \wedge 1 < m \wedge m < n \wedge 1 < k \wedge k < n$
⟨*proof*⟩

48.0.1 Make prime naively executable

lemma *Suc-0-not-prime-nat* [*simp*]: $\sim prime\ (Suc\ 0)$
⟨*proof*⟩

lemma *prime-nat-iff'*:
$prime\ (p :: nat) \longleftrightarrow p > 1 \wedge (\forall n \in \{1<..<p\}.\ \sim\ n\ dvd\ p)$
⟨*proof*⟩

lemma *prime-nat-code* [*code-unfold*]:
$(prime :: nat \Rightarrow bool) = (\lambda p.\ p > 1 \wedge (\forall n \in \{1<..<p\}.\ \sim\ n\ dvd\ p))$
⟨*proof*⟩

lemma *prime-int-iff'*:
$prime\ (p :: int) \longleftrightarrow p > 1 \wedge (\forall n \in \{1<..<p\}.\ \sim\ n\ dvd\ p)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *prime-int-code* [*code-unfold*]:
$(prime :: int \Rightarrow bool) = (\lambda p.\ p > 1 \wedge (\forall n \in \{1<..<p\}.\ \sim\ n\ dvd\ p))$
⟨*proof*⟩

lemma *prime-nat-simp*:
$prime\ p \longleftrightarrow p > 1 \wedge (\forall n \in set\ [2..<p].\ \neg\ n\ dvd\ p)$
⟨*proof*⟩

lemma *prime-int-simp*:
$prime\ (p::int) \longleftrightarrow p > 1 \wedge (\forall n \in \{2..<p\}.\ \neg\ n\ dvd\ p)$
⟨*proof*⟩

lemmas *prime-nat-simp-numeral* [*simp*] = *prime-nat-simp* [*of numeral m*] **for** *m*

lemma *two-is-prime-nat* [*simp*]: $prime\ (2::nat)$

⟨*proof*⟩

declare *prime-int-nat-transfer*[*of numeral m* **for** *m*, *simp*]

A bit of regression testing:

lemma *prime*(*97*::*nat*) ⟨*proof*⟩
lemma *prime*(*997*::*nat*) ⟨*proof*⟩
lemma *prime*(*97*::*int*) ⟨*proof*⟩
lemma *prime*(*997*::*int*) ⟨*proof*⟩

lemma *prime-factor-nat*:
  $n \neq (1::nat) \Longrightarrow \exists p.\ prime\ p \wedge p\ dvd\ n$
  ⟨*proof*⟩

48.1 Infinitely many primes

lemma *next-prime-bound*: $\exists p::nat.\ prime\ p \wedge n < p \wedge p \leq fact\ n + 1$
⟨*proof*⟩

lemma *bigger-prime*: $\exists p.\ prime\ p \wedge p > (n::nat)$
  ⟨*proof*⟩

lemma *primes-infinite*: $\neg$ (*finite* $\{(p::nat).\ prime\ p\}$)
⟨*proof*⟩

48.2 Powers of Primes

Versions for type nat only

lemma *prime-product*:
  fixes *p*::*nat*
  assumes *prime* $(p * q)$
  shows $p = 1 \vee q = 1$
⟨*proof*⟩

lemma *prime-power-mult-nat*:
  fixes *p*::*nat*
  assumes *p*: *prime p* **and** *xy*: $x * y = p\ \hat{}\ k$
  shows $\exists i\ j.\ x = p\ \hat{}\ i \wedge y = p \hat{}\ j$
⟨*proof*⟩

lemma *prime-power-exp-nat*:
  fixes *p*::*nat*
  assumes *p*: *prime p* **and** *n*: $n \neq 0$
    and *xn*: $x \hat{} n = p \hat{} k$ **shows** $\exists i.\ x = p \hat{} i$
  ⟨*proof*⟩

lemma *divides-primepow-nat*:
  fixes *p*::*nat*

assumes p: *prime p*
shows *d dvd p^k* $\longleftrightarrow$ $(\exists\ i.\ i \leq k \wedge d = p\ \hat{}\ i)$
⟨*proof*⟩

48.3 Chinese Remainder Theorem Variants

lemma *bezout-gcd-nat*:
fixes *a*::*nat* **shows** $\exists\, x\ y.\ a * x - b * y = gcd\ a\ b \vee b * x - a * y = gcd\ a\ b$
⟨*proof*⟩

lemma *gcd-bezout-sum-nat*:
fixes *a*::*nat*
assumes $a * x + b * y = d$
shows *gcd a b dvd d*
⟨*proof*⟩

A binary form of the Chinese Remainder Theorem.

lemma *chinese-remainder*:
fixes *a*::*nat* **assumes** *ab*: *coprime a b* **and** *a*: $a \neq 0$ **and** *b*: $b \neq 0$
shows $\exists\, x\ q1\ q2.\ x = u + q1 * a \wedge x = v + q2 * b$
⟨*proof*⟩

Primality

lemma *coprime-bezout-strong*:
fixes *a*::*nat* **assumes** *coprime a b* $b \neq 1$
shows $\exists\, x\ y.\ a * x = b * y + 1$
⟨*proof*⟩

lemma *bezout-prime*:
assumes *p*: *prime p* **and** *pa*: $\neg$ *p dvd a*
shows $\exists\, x\ y.\ a*x = Suc\ (p*y)$
⟨*proof*⟩

48.4 Multiplicity and primality for natural numbers and integers

lemma *prime-factors-gt-0-nat*:
$p \in$ *prime-factors x* $\Longrightarrow p > (0::nat)$
⟨*proof*⟩

lemma *prime-factors-gt-0-int*:
$p \in$ *prime-factors x* $\Longrightarrow p > (0::int)$
⟨*proof*⟩

lemma *prime-factors-ge-0-int* [*elim*]:
fixes $n :: int$
shows $p \in$ *prime-factors n* $\Longrightarrow p \geq 0$
⟨*proof*⟩

lemma *prod-mset-prime-factorization-int*:
fixes n :: *int*
assumes $n > 0$
shows *prod-mset* (*prime-factorization* n) = n
⟨*proof*⟩

lemma *prime-factorization-exists-nat*:
$n > 0 \Longrightarrow (\exists M.\ (\forall p{::}nat \in \#\text{set-mset}\ M.\ \text{prime}\ p) \wedge n = (\prod i \in\# M.\ i))$
⟨*proof*⟩

lemma *prod-mset-prime-factorization-nat* [*simp*]:
$(n{::}nat) > 0 \Longrightarrow$ *prod-mset* (*prime-factorization* n) = n
⟨*proof*⟩

lemma *prime-factorization-nat*:
$n > (0{::}nat) \Longrightarrow n = (\prod p \in \text{prime-factors}\ n.\ p\ \hat{}\ \text{multiplicity}\ p\ n)$
⟨*proof*⟩

lemma *prime-factorization-int*:
$n > (0{::}int) \Longrightarrow n = (\prod p \in \text{prime-factors}\ n.\ p\ \hat{}\ \text{multiplicity}\ p\ n)$
⟨*proof*⟩

lemma *prime-factorization-unique-nat*:
fixes f :: $nat \Rightarrow$ -
assumes *S-eq*: $S = \{p.\ 0 < f\ p\}$
and *finite* S
and S: $\forall p{\in}S.\ \text{prime}\ p\ n = (\prod p{\in}S.\ p\ \hat{}\ f\ p)$
shows $S = \text{prime-factors}\ n \wedge (\forall p.\ \text{prime}\ p \longrightarrow f\ p = \text{multiplicity}\ p\ n)$
⟨*proof*⟩

lemma *prime-factorization-unique-int*:
fixes f :: $int \Rightarrow$ -
assumes *S-eq*: $S = \{p.\ 0 < f\ p\}$
and *finite* S
and S: $\forall p{\in}S.\ \text{prime}\ p\ \text{abs}\ n = (\prod p{\in}S.\ p\ \hat{}\ f\ p)$
shows $S = \text{prime-factors}\ n \wedge (\forall p.\ \text{prime}\ p \longrightarrow f\ p = \text{multiplicity}\ p\ n)$
⟨*proof*⟩

lemma *prime-factors-characterization-nat*:
$S = \{p.\ 0 < f\ (p{::}nat)\} \Longrightarrow$
finite $S \Longrightarrow \forall p{\in}S.\ \text{prime}\ p \Longrightarrow n = (\prod p{\in}S.\ p\ \hat{}\ f\ p) \Longrightarrow \text{prime-factors}\ n = S$
⟨*proof*⟩

lemma *prime-factors-characterization′-nat*:
finite $\{p.\ 0 < f\ (p{::}nat)\} \Longrightarrow$
$(\forall p.\ 0 < f\ p \longrightarrow \text{prime}\ p) \Longrightarrow$
$\text{prime-factors}\ (\prod p \mid 0 < f\ p.\ p\ \hat{}\ f\ p) = \{p.\ 0 < f\ p\}$
⟨*proof*⟩

lemma *prime-factors-characterization-int*:
$S = \{p.\ 0 < f\ (p::int)\} \Longrightarrow$ *finite* $S \Longrightarrow$
$\forall p{\in}S.\ prime\ p \Longrightarrow abs\ n = (\prod p{\in}S.\ p\ \hat{}\ f\ p) \Longrightarrow prime\text{-}factors\ n = S$
⟨*proof*⟩

lemma *abs-prod*: $abs\ (prod\ f\ A :: 'a :: linordered\text{-}idom) = prod\ (\lambda x.\ abs\ (f\ x))\ A$
⟨*proof*⟩

lemma *primes-characterization'-int* [*rule-format*]:
finite $\{p.\ p \geq 0 \wedge 0 < f\ (p::int)\} \Longrightarrow \forall p.\ 0 < f\ p \longrightarrow prime\ p \Longrightarrow$
$prime\text{-}factors\ (\prod p \mid p \geq 0 \wedge 0 < f\ p.\ p\ \hat{}\ f\ p) = \{p.\ p \geq 0 \wedge 0 < f\ p\}$
⟨*proof*⟩

lemma *multiplicity-characterization-nat*:
$S = \{p.\ 0 < f\ (p::nat)\} \Longrightarrow$ *finite* $S \Longrightarrow \forall p{\in}S.\ prime\ p \Longrightarrow prime\ p \Longrightarrow$
$n = (\prod p{\in}S.\ p\ \hat{}\ f\ p) \Longrightarrow multiplicity\ p\ n = f\ p$
⟨*proof*⟩

lemma *multiplicity-characterization'-nat*: *finite* $\{p.\ 0 < f\ (p::nat)\} \longrightarrow$
$(\forall p.\ 0 < f\ p \longrightarrow prime\ p) \longrightarrow prime\ p \longrightarrow$
$multiplicity\ p\ (\prod p \mid 0 < f\ p.\ p\ \hat{}\ f\ p) = f\ p$
⟨*proof*⟩

lemma *multiplicity-characterization-int*: $S = \{p.\ 0 < f\ (p::int)\} \Longrightarrow$
finite $S \Longrightarrow \forall p{\in}S.\ prime\ p \Longrightarrow prime\ p \Longrightarrow n = (\prod p{\in}S.\ p\ \hat{}\ f\ p) \Longrightarrow$
$multiplicity\ p\ n = f\ p$
⟨*proof*⟩

lemma *multiplicity-characterization'-int* [*rule-format*]:
finite $\{p.\ p \geq 0 \wedge 0 < f\ (p::int)\} \Longrightarrow$
$(\forall p.\ 0 < f\ p \longrightarrow prime\ p) \Longrightarrow prime\ p \Longrightarrow$
$multiplicity\ p\ (\prod p \mid p \geq 0 \wedge 0 < f\ p.\ p\ \hat{}\ f\ p) = f\ p$
⟨*proof*⟩

lemma *multiplicity-one-nat* [*simp*]: $multiplicity\ p\ (Suc\ 0) = 0$
⟨*proof*⟩

lemma *multiplicity-eq-nat*:
fixes x **and** $y::nat$
assumes $x > 0\ y > 0\ \bigwedge p.\ prime\ p \Longrightarrow multiplicity\ p\ x = multiplicity\ p\ y$
shows $x = y$
⟨*proof*⟩

lemma *multiplicity-eq-int*:
fixes $x\ y :: int$
assumes $x > 0\ y > 0\ \bigwedge p.\ prime\ p \Longrightarrow multiplicity\ p\ x = multiplicity\ p\ y$
shows $x = y$

⟨*proof*⟩

lemma *multiplicity-prod-prime-powers*:
assumes *finite S* ⋀*x*. *x* ∈ *S* ⟹ *prime x prime p*
shows *multiplicity p* (∏ *p* ∈ *S*. *p* ^ *f p*) = (*if p* ∈ *S then f p else 0*)
⟨*proof*⟩

lemma *prime-factorization-prod-mset*:
assumes *0* ∉# *A*
shows *prime-factorization* (*prod-mset A*) = ⋃#(*image-mset prime-factorization A*)
⟨*proof*⟩

lemma *prime-factors-prod*:
assumes *finite A* **and** *0* ∉ *f* ' *A*
shows *prime-factors* (*prod f A*) = *UNION A* (*prime-factors* ∘ *f*)
⟨*proof*⟩

lemma *prime-factors-fact*:
prime-factors (*fact n*) = {*p* ∈ {*2*..*n*}. *prime p*} (**is** *?M* = *?N*)
⟨*proof*⟩

lemma *prime-dvd-fact-iff*:
assumes *prime p*
shows *p dvd fact n* ⟷ *p* ≤ *n*
⟨*proof*⟩

lemmas *prime-imp-coprime-nat* = *prime-imp-coprime*[**where** *?'a* = *nat*]
lemmas *prime-imp-coprime-int* = *prime-imp-coprime*[**where** *?'a* = *int*]
lemmas *prime-dvd-mult-nat* = *prime-dvd-mult-iff*[**where** *?'a* = *nat*]
lemmas *prime-dvd-mult-int* = *prime-dvd-mult-iff*[**where** *?'a* = *int*]
lemmas *prime-dvd-mult-eq-nat* = *prime-dvd-mult-iff*[**where** *?'a* = *nat*]
lemmas *prime-dvd-mult-eq-int* = *prime-dvd-mult-iff*[**where** *?'a* = *int*]
lemmas *prime-dvd-power-nat* = *prime-dvd-power*[**where** *?'a* = *nat*]
lemmas *prime-dvd-power-int* = *prime-dvd-power*[**where** *?'a* = *int*]
lemmas *prime-dvd-power-nat-iff* = *prime-dvd-power-iff*[**where** *?'a* = *nat*]
lemmas *prime-dvd-power-int-iff* = *prime-dvd-power-iff*[**where** *?'a* = *int*]
lemmas *prime-imp-power-coprime-nat* = *prime-imp-power-coprime*[**where** *?'a* = *nat*]
lemmas *prime-imp-power-coprime-int* = *prime-imp-power-coprime*[**where** *?'a* = *int*]
lemmas *primes-coprime-nat* = *primes-coprime*[**where** *?'a* = *nat*]
lemmas *primes-coprime-int* = *primes-coprime*[**where** *?'a* = *nat*]
lemmas *prime-divprod-pow-nat* = *prime-elem-divprod-pow*[**where** *?'a* = *nat*]
lemmas *prime-exp* = *prime-elem-power-iff*[**where** *?'a* = *nat*]

end

49 Square roots of primes are irrational

theory *Sqrt*
imports *Complex-Main* ~~/*src*/*HOL*/*Number-Theory*/*Primes*
begin

The square root of any prime number (including 2) is irrational.

theorem *sqrt-prime-irrational*:
  assumes *prime* (*p*::*nat*)
  shows *sqrt p* $\notin \mathbb{Q}$
⟨*proof*⟩

corollary *sqrt-2-not-rat*: *sqrt 2* $\notin \mathbb{Q}$
  ⟨*proof*⟩

49.1 Variations

Here is an alternative version of the main proof, using mostly linear forward-reasoning. While this results in less top-down structure, it is probably closer to proofs seen in mathematics.

theorem
  assumes *prime* (*p*::*nat*)
  shows *sqrt p* $\notin \mathbb{Q}$
⟨*proof*⟩

Another old chestnut, which is a consequence of the irrationality of 2.

lemma $\exists\, a\ b$::*real*. $a \notin \mathbb{Q} \wedge b \notin \mathbb{Q} \wedge a$ *powr* $b \in \mathbb{Q}$ (**is** $\exists\, a\ b$. *?P a b*)
⟨*proof*⟩

end

50 Square roots of primes are irrational (script version)

theory *Sqrt-Script*
imports *Complex-Main* ~~/*src*/*HOL*/*Number-Theory*/*Primes*
begin

Contrast this linear Isabelle/Isar script with Markus Wenzel's more mathematical version.

50.1 Preliminaries

lemma *prime-nonzero*: *prime* (*p*::*nat*) $\Longrightarrow p \neq 0$
  ⟨*proof*⟩

lemma *prime-dvd-other-side*:
$(n::nat) * n = p * (k * k) \Longrightarrow prime\ p \Longrightarrow p\ dvd\ n$
⟨*proof*⟩

lemma *reduction*: *prime* $(p::nat) \Longrightarrow$
$0 < k \Longrightarrow k * k = p * (j * j) \Longrightarrow k < p * j \wedge 0 < j$
⟨*proof*⟩

lemma *rearrange*: $(j::nat) * (p * j) = k * k \Longrightarrow k * k = p * (j * j)$
⟨*proof*⟩

lemma *prime-not-square*:
$prime\ (p::nat) \Longrightarrow (\bigwedge k.\ 0 < k \Longrightarrow m * m \neq p * (k * k))$
⟨*proof*⟩

50.2 Main theorem

The square root of any prime number (including *2*) is irrational.

theorem *prime-sqrt-irrational*:
$prime\ (p::nat) \Longrightarrow x * x = real\ p \Longrightarrow 0 \leq x \Longrightarrow x \notin \mathbb{Q}$
⟨*proof*⟩

lemmas *two-sqrt-irrational* =
prime-sqrt-irrational [*OF two-is-prime-nat*]

end

51 Type of finite sets defined as a subtype of sets

theory *FSet*
imports *Main*
begin

51.1 Definition of the type

typedef *'a fset* = $\{A :: 'a\ set.\ finite\ A\}$ **morphisms** *fset Abs-fset*
⟨*proof*⟩

setup-lifting *type-definition-fset*

51.2 Basic operations and type class instantiations

instantiation *fset* :: (*finite*) *finite*
begin
instance ⟨*proof*⟩
end

instantiation *fset* :: (*type*) {*bounded-lattice-bot*, *distrib-lattice*, *minus*}

begin

lift-definition *bot-fset* :: *'a fset* **is** {} **parametric** *empty-transfer* ⟨*proof*⟩

lift-definition *less-eq-fset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* **is** *subset-eq* **parametric** *subset-transfer*
⟨*proof*⟩

definition *less-fset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* **where** *xs* < *ys* ≡ *xs* ≤ *ys* ∧ *xs* ≠ (*ys*::*'a fset*)

lemma *less-fset-transfer*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique A*
shows ((*pcr-fset A*) ===> (*pcr-fset A*) ===> *op* =) *op* ⊂ *op* <
⟨*proof*⟩

lift-definition *sup-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *union* **parametric** *union-transfer*
⟨*proof*⟩

lift-definition *inf-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *inter* **parametric** *inter-transfer*
⟨*proof*⟩

lift-definition *minus-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *minus* **parametric** *Diff-transfer*
⟨*proof*⟩

instance
⟨*proof*⟩

end

abbreviation *fempty* :: *'a fset* ({||}) **where** {||} ≡ *bot*
abbreviation *fsubset-eq* :: *'a fset* ⇒ *'a fset* ⇒ *bool* (**infix** |⊆| *50*) **where** *xs* |⊆| *ys* ≡ *xs* ≤ *ys*
abbreviation *fsubset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* (**infix** |⊂| *50*) **where** *xs* |⊂| *ys* ≡ *xs* < *ys*
abbreviation *funion* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* (**infixl** |∪| *65*) **where** *xs* |∪| *ys* ≡ *sup xs ys*
abbreviation *finter* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* (**infixl** |∩| *65*) **where** *xs* |∩| *ys* ≡ *inf xs ys*
abbreviation *fminus* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* (**infixl** |−| *65*) **where** *xs* |−| *ys* ≡ *minus xs ys*

instantiation *fset* :: (*equal*) *equal*
begin
definition *HOL.equal A B* ⟷ *A* |⊆| *B* ∧ *B* |⊆| *A*
instance ⟨*proof*⟩

end

instantiation *fset* :: (*type*) *conditionally-complete-lattice*
begin

context includes *lifting-syntax*
begin

lemma *right-total-Inf-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *right-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*)
(λ*S*. *if finite* (⋂ *S* ∩ *Collect* (*Domainp A*)) *then* ⋂ *S* ∩ *Collect* (*Domainp A*) *else* {})
(λ*S*. *if finite* (*Inf S*) *then Inf S else* {})
⟨*proof*⟩

lemma *Inf-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *bi-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) (λ*A*. *if finite* (*Inf A*) *then Inf A else* {})
(λ*A*. *if finite* (*Inf A*) *then Inf A else* {})
⟨*proof*⟩

lift-definition *Inf-fset* :: *'a fset set* ⇒ *'a fset* **is** λ*A*. *if finite* (*Inf A*) *then Inf A else* {}
parametric *right-total-Inf-fset-transfer Inf-fset-transfer* ⟨*proof*⟩

lemma *Sup-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) (λ*A*. *if finite* (*Sup A*) *then Sup A else* {})
(λ*A*. *if finite* (*Sup A*) *then Sup A else* {}) ⟨*proof*⟩

lift-definition *Sup-fset* :: *'a fset set* ⇒ *'a fset* **is** λ*A*. *if finite* (*Sup A*) *then Sup A else* {}
parametric *Sup-fset-transfer* ⟨*proof*⟩

lemma *finite-Sup*: ∃ *z*. *finite z* ∧ (∀ *a*. *a* ∈ *X* ⟶ *a* ≤ *z*) ⟹ *finite* (*Sup X*)
⟨*proof*⟩

lemma *transfer-bdd-below*[*transfer-rule*]: (*rel-set* (*pcr-fset op* =) ===> *op* =) *bdd-below bdd-below*
⟨*proof*⟩

end

instance
⟨*proof*⟩
end

instantiation *fset* :: (*finite*) *complete-lattice*
begin

lift-definition *top-fset* :: *'a fset* **is** *UNIV* **parametric** *right-total-UNIV-transfer UNIV-transfer*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *fset* :: (*finite*) *complete-boolean-algebra*
begin

lift-definition *uminus-fset* :: *'a fset* ⇒ *'a fset* **is** *uminus*
parametric *right-total-Compl-transfer Compl-transfer* ⟨*proof*⟩

instance
⟨*proof*⟩

end

abbreviation *fUNIV* :: *'a*::*finite fset* **where** *fUNIV* ≡ *top*
abbreviation *fuminus* :: *'a*::*finite fset* ⇒ *'a fset* (|−| - [*81*] *80*) **where** |−| *x* ≡ *uminus x*

declare *top-fset.rep-eq*[*simp*]

51.3 Other operations

lift-definition *finsert* :: *'a* ⇒ *'a fset* ⇒ *'a fset* **is** *insert* **parametric** *Lifting-Set.insert-transfer*
⟨*proof*⟩

syntax
-insert-fset :: *args* => *'a fset* ({|(-)|})

translations
{|*x*, *xs*|} == *CONST finsert x* {|*xs*|}
{|*x*|} == *CONST finsert x* {||}

lift-definition *fmember* :: *'a* ⇒ *'a fset* ⇒ *bool* (**infix** |∈| *50*) **is** *Set.member*
parametric *member-transfer* ⟨*proof*⟩

abbreviation *notin-fset* :: *'a* ⇒ *'a fset* ⇒ *bool* (**infix** |∉| *50*) **where** *x* |∉| *S* ≡ ¬ (*x* |∈| *S*)

context includes *lifting-syntax*

begin

lift-definition *ffilter* :: $('a \Rightarrow bool) \Rightarrow 'a\ fset \Rightarrow 'a\ fset$ **is** *Set.filter*
parametric *Lifting-Set.filter-transfer* ⟨*proof*⟩

lift-definition *fPow* :: $'a\ fset \Rightarrow 'a\ fset\ fset$ **is** *Pow* **parametric** *Pow-transfer*
⟨*proof*⟩

lift-definition *fcard* :: $'a\ fset \Rightarrow nat$ **is** *card* **parametric** *card-transfer* ⟨*proof*⟩

lift-definition *fimage* :: $('a \Rightarrow 'b) \Rightarrow 'a\ fset \Rightarrow 'b\ fset$ (**infixr** |‘| *90*) **is** *image*
parametric *image-transfer* ⟨*proof*⟩

lift-definition *fthe-elem* :: $'a\ fset \Rightarrow 'a$ **is** *the-elem* ⟨*proof*⟩

lift-definition *fbind* :: $'a\ fset \Rightarrow ('a \Rightarrow 'b\ fset) \Rightarrow 'b\ fset$ **is** *Set.bind* **parametric**
bind-transfer
⟨*proof*⟩

lift-definition *ffUnion* :: $'a\ fset\ fset \Rightarrow 'a\ fset$ **is** *Union* **parametric** *Union-transfer*
⟨*proof*⟩

lift-definition *fBall* :: $'a\ fset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$ **is** *Ball* **parametric** *Ball-transfer*
⟨*proof*⟩
lift-definition *fBex* :: $'a\ fset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$ **is** *Bex* **parametric** *Bex-transfer*
⟨*proof*⟩

lift-definition *ffold* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ fset \Rightarrow 'b$ **is** *Finite-Set.fold*
⟨*proof*⟩

lift-definition *fset-of-list* :: $'a\ list \Rightarrow 'a\ fset$ **is** *set* ⟨*proof*⟩

51.4 Transferred lemmas from Set.thy

lemmas *fset-eqI* = *set-eqI*[*Transfer.transferred*]
lemmas *fset-eq-iff*[*no-atp*] = *set-eq-iff*[*Transfer.transferred*]
lemmas *fBallI*[*intro*!] = *ballI*[*Transfer.transferred*]
lemmas *fbspec*[*dest?*] = *bspec*[*Transfer.transferred*]
lemmas *fBallE*[*elim*] = *ballE*[*Transfer.transferred*]
lemmas *fBexI*[*intro*] = *bexI*[*Transfer.transferred*]
lemmas *rev-fBexI*[*intro?*] = *rev-bexI*[*Transfer.transferred*]
lemmas *fBexCI* = *bexCI*[*Transfer.transferred*]
lemmas *fBexE*[*elim*!] = *bexE*[*Transfer.transferred*]
lemmas *fBall-triv*[*simp*] = *ball-triv*[*Transfer.transferred*]
lemmas *fBex-triv*[*simp*] = *bex-triv*[*Transfer.transferred*]
lemmas *fBex-triv-one-point1*[*simp*] = *bex-triv-one-point1*[*Transfer.transferred*]
lemmas *fBex-triv-one-point2*[*simp*] = *bex-triv-one-point2*[*Transfer.transferred*]
lemmas *fBex-one-point1*[*simp*] = *bex-one-point1*[*Transfer.transferred*]
lemmas *fBex-one-point2*[*simp*] = *bex-one-point2*[*Transfer.transferred*]

lemmas *fBall-one-point1*[*simp*] = *ball-one-point1*[*Transfer.transferred*]
lemmas *fBall-one-point2*[*simp*] = *ball-one-point2*[*Transfer.transferred*]
lemmas *fBall-conj-distrib* = *ball-conj-distrib*[*Transfer.transferred*]
lemmas *fBex-disj-distrib* = *bex-disj-distrib*[*Transfer.transferred*]
lemmas *fBall-cong* = *ball-cong*[*Transfer.transferred*]
lemmas *fBex-cong* = *bex-cong*[*Transfer.transferred*]
lemmas *fsubsetI*[*intro*!] = *subsetI*[*Transfer.transferred*]
lemmas *fsubsetD*[*elim*, *intro?*] = *subsetD*[*Transfer.transferred*]
lemmas *rev-fsubsetD*[*no-atp*,*intro?*] = *rev-subsetD*[*Transfer.transferred*]
lemmas *fsubsetCE*[*no-atp*,*elim*] = *subsetCE*[*Transfer.transferred*]
lemmas *fsubset-eq*[*no-atp*] = *subset-eq*[*Transfer.transferred*]
lemmas *contra-fsubsetD*[*no-atp*] = *contra-subsetD*[*Transfer.transferred*]
lemmas *fsubset-refl* = *subset-refl*[*Transfer.transferred*]
lemmas *fsubset-trans* = *subset-trans*[*Transfer.transferred*]
lemmas *fset-rev-mp* = *set-rev-mp*[*Transfer.transferred*]
lemmas *fset-mp* = *set-mp*[*Transfer.transferred*]
lemmas *fsubset-not-fsubset-eq*[*code*] = *subset-not-subset-eq*[*Transfer.transferred*]
lemmas *eq-fmem-trans* = *eq-mem-trans*[*Transfer.transferred*]
lemmas *fsubset-antisym*[*intro*!] = *subset-antisym*[*Transfer.transferred*]
lemmas *fequalityD1* = *equalityD1*[*Transfer.transferred*]
lemmas *fequalityD2* = *equalityD2*[*Transfer.transferred*]
lemmas *fequalityE* = *equalityE*[*Transfer.transferred*]
lemmas *fequalityCE*[*elim*] = *equalityCE*[*Transfer.transferred*]
lemmas *eqfset-imp-iff* = *eqset-imp-iff*[*Transfer.transferred*]
lemmas *eqfelem-imp-iff* = *eqelem-imp-iff*[*Transfer.transferred*]
lemmas *fempty-iff*[*simp*] = *empty-iff*[*Transfer.transferred*]
lemmas *fempty-fsubsetI*[*iff*] = *empty-subsetI*[*Transfer.transferred*]
lemmas *equalsffemptyI* = *equals0I*[*Transfer.transferred*]
lemmas *equalsffemptyD* = *equals0D*[*Transfer.transferred*]
lemmas *fBall-fempty*[*simp*] = *ball-empty*[*Transfer.transferred*]
lemmas *fBex-fempty*[*simp*] = *bex-empty*[*Transfer.transferred*]
lemmas *fPow-iff*[*iff*] = *Pow-iff*[*Transfer.transferred*]
lemmas *fPowI* = *PowI*[*Transfer.transferred*]
lemmas *fPowD* = *PowD*[*Transfer.transferred*]
lemmas *fPow-bottom* = *Pow-bottom*[*Transfer.transferred*]
lemmas *fPow-top* = *Pow-top*[*Transfer.transferred*]
lemmas *fPow-not-fempty* = *Pow-not-empty*[*Transfer.transferred*]
lemmas *finter-iff*[*simp*] = *Int-iff*[*Transfer.transferred*]
lemmas *finterI*[*intro*!] = *IntI*[*Transfer.transferred*]
lemmas *finterD1* = *IntD1*[*Transfer.transferred*]
lemmas *finterD2* = *IntD2*[*Transfer.transferred*]
lemmas *finterE*[*elim*!] = *IntE*[*Transfer.transferred*]
lemmas *funion-iff*[*simp*] = *Un-iff*[*Transfer.transferred*]
lemmas *funionI1*[*elim?*] = *UnI1*[*Transfer.transferred*]
lemmas *funionI2*[*elim?*] = *UnI2*[*Transfer.transferred*]
lemmas *funionCI*[*intro*!] = *UnCI*[*Transfer.transferred*]
lemmas *funionE*[*elim*!] = *UnE*[*Transfer.transferred*]
lemmas *fminus-iff*[*simp*] = *Diff-iff*[*Transfer.transferred*]
lemmas *fminusI*[*intro*!] = *DiffI*[*Transfer.transferred*]

lemmas *fminusD1* = *DiffD1*[*Transfer.transferred*]
lemmas *fminusD2* = *DiffD2*[*Transfer.transferred*]
lemmas *fminusE*[*elim*!] = *DiffE*[*Transfer.transferred*]
lemmas *finsert-iff*[*simp*] = *insert-iff*[*Transfer.transferred*]
lemmas *finsertI1* = *insertI1*[*Transfer.transferred*]
lemmas *finsertI2* = *insertI2*[*Transfer.transferred*]
lemmas *finsertE*[*elim*!] = *insertE*[*Transfer.transferred*]
lemmas *finsertCI*[*intro*!] = *insertCI*[*Transfer.transferred*]
lemmas *fsubset-finsert-iff* = *subset-insert-iff*[*Transfer.transferred*]
lemmas *finsert-ident* = *insert-ident*[*Transfer.transferred*]
lemmas *fsingletonI*[*intro*!,*no-atp*] = *singletonI*[*Transfer.transferred*]
lemmas *fsingletonD*[*dest*!,*no-atp*] = *singletonD*[*Transfer.transferred*]
lemmas *fsingleton-iff* = *singleton-iff*[*Transfer.transferred*]
lemmas *fsingleton-inject*[*dest*!] = *singleton-inject*[*Transfer.transferred*]
lemmas *fsingleton-finsert-inj-eq*[*iff*,*no-atp*] = *singleton-insert-inj-eq*[*Transfer.transferred*]
lemmas *fsingleton-finsert-inj-eq'*[*iff*,*no-atp*] = *singleton-insert-inj-eq'*[*Transfer.transferred*]
lemmas *fsubset-fsingletonD* = *subset-singletonD*[*Transfer.transferred*]
lemmas *fminus-single-finsert* = *Diff-single-insert*[*Transfer.transferred*]
lemmas *fdoubleton-eq-iff* = *doubleton-eq-iff*[*Transfer.transferred*]
lemmas *funion-fsingleton-iff* = *Un-singleton-iff*[*Transfer.transferred*]
lemmas *fsingleton-funion-iff* = *singleton-Un-iff*[*Transfer.transferred*]
lemmas *fimage-eqI*[*simp*, *intro*] = *image-eqI*[*Transfer.transferred*]
lemmas *fimageI* = *imageI*[*Transfer.transferred*]
lemmas *rev-fimage-eqI* = *rev-image-eqI*[*Transfer.transferred*]
lemmas *fimageE*[*elim*!] = *imageE*[*Transfer.transferred*]
lemmas *Compr-fimage-eq* = *Compr-image-eq*[*Transfer.transferred*]
lemmas *fimage-funion* = *image-Un*[*Transfer.transferred*]
lemmas *fimage-iff* = *image-iff*[*Transfer.transferred*]
lemmas *fimage-fsubset-iff*[*no-atp*] = *image-subset-iff*[*Transfer.transferred*]
lemmas *fimage-fsubsetI* = *image-subsetI*[*Transfer.transferred*]
lemmas *fimage-ident*[*simp*] = *image-ident*[*Transfer.transferred*]
lemmas *if-split-fmem1* = *if-split-mem1*[*Transfer.transferred*]
lemmas *if-split-fmem2* = *if-split-mem2*[*Transfer.transferred*]
lemmas *pfsubsetI*[*intro*!,*no-atp*] = *psubsetI*[*Transfer.transferred*]
lemmas *pfsubsetE*[*elim*!,*no-atp*] = *psubsetE*[*Transfer.transferred*]
lemmas *pfsubset-finsert-iff* = *psubset-insert-iff*[*Transfer.transferred*]
lemmas *pfsubset-eq* = *psubset-eq*[*Transfer.transferred*]
lemmas *pfsubset-imp-fsubset* = *psubset-imp-subset*[*Transfer.transferred*]
lemmas *pfsubset-trans* = *psubset-trans*[*Transfer.transferred*]
lemmas *pfsubsetD* = *psubsetD*[*Transfer.transferred*]
lemmas *pfsubset-fsubset-trans* = *psubset-subset-trans*[*Transfer.transferred*]
lemmas *fsubset-pfsubset-trans* = *subset-psubset-trans*[*Transfer.transferred*]
lemmas *pfsubset-imp-ex-fmem* = *psubset-imp-ex-mem*[*Transfer.transferred*]
lemmas *fimage-fPow-mono* = *image-Pow-mono*[*Transfer.transferred*]
lemmas *fimage-fPow-surj* = *image-Pow-surj*[*Transfer.transferred*]
lemmas *fsubset-finsertI* = *subset-insertI*[*Transfer.transferred*]
lemmas *fsubset-finsertI2* = *subset-insertI2*[*Transfer.transferred*]
lemmas *fsubset-finsert* = *subset-insert*[*Transfer.transferred*]
lemmas *funion-upper1* = *Un-upper1*[*Transfer.transferred*]

lemmas *funion-upper2* = *Un-upper2*[*Transfer.transferred*]
lemmas *funion-least* = *Un-least*[*Transfer.transferred*]
lemmas *finter-lower1* = *Int-lower1*[*Transfer.transferred*]
lemmas *finter-lower2* = *Int-lower2*[*Transfer.transferred*]
lemmas *finter-greatest* = *Int-greatest*[*Transfer.transferred*]
lemmas *fminus-fsubset* = *Diff-subset*[*Transfer.transferred*]
lemmas *fminus-fsubset-conv* = *Diff-subset-conv*[*Transfer.transferred*]
lemmas *fsubset-fempty*[*simp*] = *subset-empty*[*Transfer.transferred*]
lemmas *not-pfsubset-fempty*[*iff*] = *not-psubset-empty*[*Transfer.transferred*]
lemmas *finsert-is-funion* = *insert-is-Un*[*Transfer.transferred*]
lemmas *finsert-not-fempty*[*simp*] = *insert-not-empty*[*Transfer.transferred*]
lemmas *fempty-not-finsert* = *empty-not-insert*[*Transfer.transferred*]
lemmas *finsert-absorb* = *insert-absorb*[*Transfer.transferred*]
lemmas *finsert-absorb2*[*simp*] = *insert-absorb2*[*Transfer.transferred*]
lemmas *finsert-commute* = *insert-commute*[*Transfer.transferred*]
lemmas *finsert-fsubset*[*simp*] = *insert-subset*[*Transfer.transferred*]
lemmas *finsert-inter-finsert*[*simp*] = *insert-inter-insert*[*Transfer.transferred*]
lemmas *finsert-disjoint*[*simp*,*no-atp*] = *insert-disjoint*[*Transfer.transferred*]
lemmas *disjoint-finsert*[*simp*,*no-atp*] = *disjoint-insert*[*Transfer.transferred*]
lemmas *fimage-fempty*[*simp*] = *image-empty*[*Transfer.transferred*]
lemmas *fimage-finsert*[*simp*] = *image-insert*[*Transfer.transferred*]
lemmas *fimage-constant* = *image-constant*[*Transfer.transferred*]
lemmas *fimage-constant-conv* = *image-constant-conv*[*Transfer.transferred*]
lemmas *fimage-fimage* = *image-image*[*Transfer.transferred*]
lemmas *finsert-fimage*[*simp*] = *insert-image*[*Transfer.transferred*]
lemmas *fimage-is-fempty*[*iff*] = *image-is-empty*[*Transfer.transferred*]
lemmas *fempty-is-fimage*[*iff*] = *empty-is-image*[*Transfer.transferred*]
lemmas *fimage-cong* = *image-cong*[*Transfer.transferred*]
lemmas *fimage-finter-fsubset* = *image-Int-subset*[*Transfer.transferred*]
lemmas *fimage-fminus-fsubset* = *image-diff-subset*[*Transfer.transferred*]
lemmas *finter-absorb* = *Int-absorb*[*Transfer.transferred*]
lemmas *finter-left-absorb* = *Int-left-absorb*[*Transfer.transferred*]
lemmas *finter-commute* = *Int-commute*[*Transfer.transferred*]
lemmas *finter-left-commute* = *Int-left-commute*[*Transfer.transferred*]
lemmas *finter-assoc* = *Int-assoc*[*Transfer.transferred*]
lemmas *finter-ac* = *Int-ac*[*Transfer.transferred*]
lemmas *finter-absorb1* = *Int-absorb1*[*Transfer.transferred*]
lemmas *finter-absorb2* = *Int-absorb2*[*Transfer.transferred*]
lemmas *finter-fempty-left* = *Int-empty-left*[*Transfer.transferred*]
lemmas *finter-fempty-right* = *Int-empty-right*[*Transfer.transferred*]
lemmas *disjoint-iff-fnot-equal* = *disjoint-iff-not-equal*[*Transfer.transferred*]
lemmas *finter-funion-distrib* = *Int-Un-distrib*[*Transfer.transferred*]
lemmas *finter-funion-distrib2* = *Int-Un-distrib2*[*Transfer.transferred*]
lemmas *finter-fsubset-iff*[*no-atp*, *simp*] = *Int-subset-iff*[*Transfer.transferred*]
lemmas *funion-absorb* = *Un-absorb*[*Transfer.transferred*]
lemmas *funion-left-absorb* = *Un-left-absorb*[*Transfer.transferred*]
lemmas *funion-commute* = *Un-commute*[*Transfer.transferred*]
lemmas *funion-left-commute* = *Un-left-commute*[*Transfer.transferred*]
lemmas *funion-assoc* = *Un-assoc*[*Transfer.transferred*]

lemmas *funion-ac* = *Un-ac*[*Transfer.transferred*]
lemmas *funion-absorb1* = *Un-absorb1*[*Transfer.transferred*]
lemmas *funion-absorb2* = *Un-absorb2*[*Transfer.transferred*]
lemmas *funion-fempty-left* = *Un-empty-left*[*Transfer.transferred*]
lemmas *funion-fempty-right* = *Un-empty-right*[*Transfer.transferred*]
lemmas *funion-finsert-left*[*simp*] = *Un-insert-left*[*Transfer.transferred*]
lemmas *funion-finsert-right*[*simp*] = *Un-insert-right*[*Transfer.transferred*]
lemmas *finter-finsert-left* = *Int-insert-left*[*Transfer.transferred*]
lemmas *finter-finsert-left-ifffempty*[*simp*] = *Int-insert-left-if0*[*Transfer.transferred*]
lemmas *finter-finsert-left-if1*[*simp*] = *Int-insert-left-if1*[*Transfer.transferred*]
lemmas *finter-finsert-right* = *Int-insert-right*[*Transfer.transferred*]
lemmas *finter-finsert-right-ifffempty*[*simp*] = *Int-insert-right-if0*[*Transfer.transferred*]
lemmas *finter-finsert-right-if1*[*simp*] = *Int-insert-right-if1*[*Transfer.transferred*]
lemmas *funion-finter-distrib* = *Un-Int-distrib*[*Transfer.transferred*]
lemmas *funion-finter-distrib2* = *Un-Int-distrib2*[*Transfer.transferred*]
lemmas *funion-finter-crazy* = *Un-Int-crazy*[*Transfer.transferred*]
lemmas *fsubset-funion-eq* = *subset-Un-eq*[*Transfer.transferred*]
lemmas *funion-fempty*[*iff*] = *Un-empty*[*Transfer.transferred*]
lemmas *funion-fsubset-iff*[*no-atp*, *simp*] = *Un-subset-iff*[*Transfer.transferred*]
lemmas *funion-fminus-finter* = *Un-Diff-Int*[*Transfer.transferred*]
lemmas *fminus-finter2* = *Diff-Int2*[*Transfer.transferred*]
lemmas *funion-finter-assoc-eq* = *Un-Int-assoc-eq*[*Transfer.transferred*]
lemmas *fBall-funion* = *ball-Un*[*Transfer.transferred*]
lemmas *fBex-funion* = *bex-Un*[*Transfer.transferred*]
lemmas *fminus-eq-fempty-iff*[*simp*,*no-atp*] = *Diff-eq-empty-iff*[*Transfer.transferred*]
lemmas *fminus-cancel*[*simp*] = *Diff-cancel*[*Transfer.transferred*]
lemmas *fminus-idemp*[*simp*] = *Diff-idemp*[*Transfer.transferred*]
lemmas *fminus-triv* = *Diff-triv*[*Transfer.transferred*]
lemmas *fempty-fminus*[*simp*] = *empty-Diff*[*Transfer.transferred*]
lemmas *fminus-fempty*[*simp*] = *Diff-empty*[*Transfer.transferred*]
lemmas *fminus-finsertffempty*[*simp*,*no-atp*] = *Diff-insert0*[*Transfer.transferred*]
lemmas *fminus-finsert* = *Diff-insert*[*Transfer.transferred*]
lemmas *fminus-finsert2* = *Diff-insert2*[*Transfer.transferred*]
lemmas *finsert-fminus-if* = *insert-Diff-if*[*Transfer.transferred*]
lemmas *finsert-fminus1*[*simp*] = *insert-Diff1*[*Transfer.transferred*]
lemmas *finsert-fminus-single*[*simp*] = *insert-Diff-single*[*Transfer.transferred*]
lemmas *finsert-fminus* = *insert-Diff*[*Transfer.transferred*]
lemmas *fminus-finsert-absorb* = *Diff-insert-absorb*[*Transfer.transferred*]
lemmas *fminus-disjoint*[*simp*] = *Diff-disjoint*[*Transfer.transferred*]
lemmas *fminus-partition* = *Diff-partition*[*Transfer.transferred*]
lemmas *double-fminus* = *double-diff*[*Transfer.transferred*]
lemmas *funion-fminus-cancel*[*simp*] = *Un-Diff-cancel*[*Transfer.transferred*]
lemmas *funion-fminus-cancel2*[*simp*] = *Un-Diff-cancel2*[*Transfer.transferred*]
lemmas *fminus-funion* = *Diff-Un*[*Transfer.transferred*]
lemmas *fminus-finter* = *Diff-Int*[*Transfer.transferred*]
lemmas *funion-fminus* = *Un-Diff*[*Transfer.transferred*]
lemmas *finter-fminus* = *Int-Diff*[*Transfer.transferred*]
lemmas *fminus-finter-distrib* = *Diff-Int-distrib*[*Transfer.transferred*]
lemmas *fminus-finter-distrib2* = *Diff-Int-distrib2*[*Transfer.transferred*]

lemmas *fUNIV-bool*[*no-atp*] = *UNIV-bool*[*Transfer.transferred*]
lemmas *fPow-fempty*[*simp*] = *Pow-empty*[*Transfer.transferred*]
lemmas *fPow-finsert* = *Pow-insert*[*Transfer.transferred*]
lemmas *funion-fPow-fsubset* = *Un-Pow-subset*[*Transfer.transferred*]
lemmas *fPow-finter-eq*[*simp*] = *Pow-Int-eq*[*Transfer.transferred*]
lemmas *fset-eq-fsubset* = *set-eq-subset*[*Transfer.transferred*]
lemmas *fsubset-iff*[*no-atp*] = *subset-iff*[*Transfer.transferred*]
lemmas *fsubset-iff-pfsubset-eq* = *subset-iff-psubset-eq*[*Transfer.transferred*]
lemmas *all-not-fin-conv*[*simp*] = *all-not-in-conv*[*Transfer.transferred*]
lemmas *ex-fin-conv* = *ex-in-conv*[*Transfer.transferred*]
lemmas *fimage-mono* = *image-mono*[*Transfer.transferred*]
lemmas *fPow-mono* = *Pow-mono*[*Transfer.transferred*]
lemmas *finsert-mono* = *insert-mono*[*Transfer.transferred*]
lemmas *funion-mono* = *Un-mono*[*Transfer.transferred*]
lemmas *finter-mono* = *Int-mono*[*Transfer.transferred*]
lemmas *fminus-mono* = *Diff-mono*[*Transfer.transferred*]
lemmas *fin-mono* = *in-mono*[*Transfer.transferred*]
lemmas *fthe-felem-eq*[*simp*] = *the-elem-eq*[*Transfer.transferred*]
lemmas *fLeast-mono* = *Least-mono*[*Transfer.transferred*]
lemmas *fbind-fbind* = *bind-bind*[*Transfer.transferred*]
lemmas *fempty-fbind*[*simp*] = *empty-bind*[*Transfer.transferred*]
lemmas *nonfempty-fbind-const* = *nonempty-bind-const*[*Transfer.transferred*]
lemmas *fbind-const* = *bind-const*[*Transfer.transferred*]
lemmas *ffmember-filter*[*simp*] = *member-filter*[*Transfer.transferred*]
lemmas *fequalityI* = *equalityI*[*Transfer.transferred*]
lemmas *fset-of-list-simps*[*simp*] = *set-simps*[*Transfer.transferred*]
lemmas *fset-of-list-append*[*simp*] = *set-append*[*Transfer.transferred*]
lemmas *fset-of-list-rev*[*simp*] = *set-rev*[*Transfer.transferred*]
lemmas *fset-of-list-map*[*simp*] = *set-map*[*Transfer.transferred*]

51.5 Additional lemmas

51.5.1 *fsingleton*

lemmas *fsingletonE* = *fsingletonD* [*elim-format*]

51.5.2 *femepty*

lemma *fempty-ffilter*[*simp*]: *ffilter* (λ-. *False*) *A* = {||}
⟨*proof*⟩

lemma *femptyE* [*elim*!]: *a* |∈| {||} ⟹ *P*
⟨*proof*⟩

51.5.3 *fset*

lemmas *fset-simps*[*simp*] = *bot-fset.rep-eq finsert.rep-eq*

lemma *finite-fset* [*simp*]:

shows *finite* (*fset S*)
⟨*proof*⟩

lemmas *fset-cong* = *fset-inject*

lemma *filter-fset* [*simp*]:
shows *fset* (*ffilter P xs*) = *Collect P* ∩ *fset xs*
⟨*proof*⟩

lemma *notin-fset*: *x* |∉| *S* ⟷ *x* ∉ *fset S* ⟨*proof*⟩

lemmas *inter-fset*[*simp*] = *inf-fset.rep-eq*

lemmas *union-fset*[*simp*] = *sup-fset.rep-eq*

lemmas *minus-fset*[*simp*] = *minus-fset.rep-eq*

51.5.4 *ffilter*

lemma *subset-ffilter*:
ffilter P A |⊆| *ffilter Q A* = (∀ *x*. *x* |∈| *A* ⟶ *P x* ⟶ *Q x*)
⟨*proof*⟩

lemma *eq-ffilter*:
(*ffilter P A* = *ffilter Q A*) = (∀ *x*. *x* |∈| *A* ⟶ *P x* = *Q x*)
⟨*proof*⟩

lemma *pfsubset-ffilter*:
(⋀*x*. *x* |∈| *A* ⟹ *P x* ⟹ *Q x*) ⟹ (*x* |∈| *A* & ¬ *P x* & *Q x*) ⟹
ffilter P A |⊂| *ffilter Q A*
⟨*proof*⟩

51.5.5 *fset-of-list*

lemma *fset-of-list-filter*[*simp*]:
fset-of-list (*filter P xs*) = *ffilter P* (*fset-of-list xs*)
⟨*proof*⟩

lemma *fset-of-list-subset*[*intro*]:
set xs ⊆ *set ys* ⟹ *fset-of-list xs* |⊆| *fset-of-list ys*
⟨*proof*⟩

lemma *fset-of-list-elem*: (*x* |∈| *fset-of-list xs*) ⟷ (*x* ∈ *set xs*)
⟨*proof*⟩

51.5.6 *finsert*

lemma *set-finsert*:
assumes *x* |∈| *A*
obtains *B* **where** *A* = *finsert x B* **and** *x* |∉| *B*

⟨*proof*⟩

lemma *mk-disjoint-finsert*: $a \mid\in\mid A \Longrightarrow \exists B.\ A = \mathit{finsert}\ a\ B \wedge a \mid\notin\mid B$
⟨*proof*⟩

51.5.7 *fimage*

lemma *subset-fimage-iff*: $(B \mid\subseteq\mid f\mid`\mid A) = (\exists\ AA.\ AA \mid\subseteq\mid A \wedge B = f\mid`\mid AA)$
⟨*proof*⟩

51.5.8 bounded quantification

lemma *bex-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.\ \mathit{fBex}\ A\ (\lambda x.\ P\ x \wedge Q) = (\mathit{fBex}\ A\ P \wedge Q)$
$\bigwedge A\ P\ Q.\ \mathit{fBex}\ A\ (\lambda x.\ P \wedge Q\ x) = (P \wedge \mathit{fBex}\ A\ Q)$
$\bigwedge P.\ \mathit{fBex}\ \{||\}\ P = \mathit{False}$
$\bigwedge a\ B\ P.\ \mathit{fBex}\ (\mathit{finsert}\ a\ B)\ P = (P\ a \vee \mathit{fBex}\ B\ P)$
$\bigwedge A\ P\ f.\ \mathit{fBex}\ (f\ \mid`\mid\ A)\ P = \mathit{fBex}\ A\ (\lambda x.\ P\ (f\ x))$
$\bigwedge A\ P.\ (\neg\ \mathit{fBex}\ A\ P) = \mathit{fBall}\ A\ (\lambda x.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *ball-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.\ \mathit{fBall}\ A\ (\lambda x.\ P\ x \vee Q) = (\mathit{fBall}\ A\ P \vee Q)$
$\bigwedge A\ P\ Q.\ \mathit{fBall}\ A\ (\lambda x.\ P \vee Q\ x) = (P \vee \mathit{fBall}\ A\ Q)$
$\bigwedge A\ P\ Q.\ \mathit{fBall}\ A\ (\lambda x.\ P \longrightarrow Q\ x) = (P \longrightarrow \mathit{fBall}\ A\ Q)$
$\bigwedge A\ P\ Q.\ \mathit{fBall}\ A\ (\lambda x.\ P\ x \longrightarrow Q) = (\mathit{fBex}\ A\ P \longrightarrow Q)$
$\bigwedge P.\ \mathit{fBall}\ \{||\}\ P = \mathit{True}$
$\bigwedge a\ B\ P.\ \mathit{fBall}\ (\mathit{finsert}\ a\ B)\ P = (P\ a \wedge \mathit{fBall}\ B\ P)$
$\bigwedge A\ P\ f.\ \mathit{fBall}\ (f\ \mid`\mid\ A)\ P = \mathit{fBall}\ A\ (\lambda x.\ P\ (f\ x))$
$\bigwedge A\ P.\ (\neg\ \mathit{fBall}\ A\ P) = \mathit{fBex}\ A\ (\lambda x.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *atomize-fBall*:
$(\bigwedge x.\ x \mid\in\mid A \Longrightarrow P\ x) \equiv \mathit{Trueprop}\ (\mathit{fBall}\ A\ (\lambda x.\ P\ x))$
⟨*proof*⟩

lemma *fBall-mono*[*mono*]: $P \leq Q \Longrightarrow \mathit{fBall}\ S\ P \leq \mathit{fBall}\ S\ Q$
⟨*proof*⟩

end

51.5.9 *fcard*

lemma *fcard-fempty*:
$\mathit{fcard}\ \{||\} = 0$
⟨*proof*⟩

lemma *fcard-finsert-disjoint*:
$x \mid\notin\mid A \Longrightarrow \mathit{fcard}\ (\mathit{finsert}\ x\ A) = \mathit{Suc}\ (\mathit{fcard}\ A)$

⟨*proof*⟩

lemma *fcard-finsert-if*:
fcard (*finsert x A*) = (*if x* |∈| *A then fcard A else Suc* (*fcard A*))
⟨*proof*⟩

lemma *card-0-eq* [*simp*, *no-atp*]:
fcard A = *0* ⟷ *A* = {||}
⟨*proof*⟩

lemma *fcard-Suc-fminus1*:
x |∈| *A* ⟹ *Suc* (*fcard* (*A* |−| {|*x*|})) = *fcard A*
⟨*proof*⟩

lemma *fcard-fminus-fsingleton*:
x |∈| *A* ⟹ *fcard* (*A* |−| {|*x*|}) = *fcard A* − *1*
⟨*proof*⟩

lemma *fcard-fminus-fsingleton-if*:
fcard (*A* |−| {|*x*|}) = (*if x* |∈| *A then fcard A* − *1 else fcard A*)
⟨*proof*⟩

lemma *fcard-fminus-finsert*[*simp*]:
assumes *a* |∈| *A* **and** *a* |∉| *B*
shows *fcard* (*A* |−| *finsert a B*) = *fcard* (*A* |−| *B*) − *1*
⟨*proof*⟩

lemma *fcard-finsert*: *fcard* (*finsert x A*) = *Suc* (*fcard* (*A* |−| {|*x*|}))
⟨*proof*⟩

lemma *fcard-finsert-le*: *fcard A* ≤ *fcard* (*finsert x A*)
⟨*proof*⟩

lemma *fcard-mono*:
A |⊆| *B* ⟹ *fcard A* ≤ *fcard B*
⟨*proof*⟩

lemma *fcard-seteq*: *A* |⊆| *B* ⟹ *fcard B* ≤ *fcard A* ⟹ *A* = *B*
⟨*proof*⟩

lemma *pfsubset-fcard-mono*: *A* |⊂| *B* ⟹ *fcard A* < *fcard B*
⟨*proof*⟩

lemma *fcard-funion-finter*:
fcard A + *fcard B* = *fcard* (*A* |∪| *B*) + *fcard* (*A* |∩| *B*)
⟨*proof*⟩

lemma *fcard-funion-disjoint*:
A |∩| *B* = {||} ⟹ *fcard* (*A* |∪| *B*) = *fcard A* + *fcard B*

⟨*proof*⟩

lemma *fcard-funion-fsubset*:
  $B \mid\subseteq\mid A \Longrightarrow$ *fcard* $(A \mid-\mid B) =$ *fcard* $A -$ *fcard* B
⟨*proof*⟩

lemma *diff-fcard-le-fcard-fminus*:
  fcard $A -$ *fcard* $B \leq$ *fcard*$(A \mid-\mid B)$
⟨*proof*⟩

lemma *fcard-fminus1-less*: $x \mid\in\mid A \Longrightarrow$ *fcard* $(A \mid-\mid \{\mid x \mid\}) <$ *fcard* A
⟨*proof*⟩

lemma *fcard-fminus2-less*:
  $x \mid\in\mid A \Longrightarrow y \mid\in\mid A \Longrightarrow$ *fcard* $(A \mid-\mid \{\mid x \mid\} \mid-\mid \{\mid y \mid\}) <$ *fcard* A
⟨*proof*⟩

lemma *fcard-fminus1-le*: *fcard* $(A \mid-\mid \{\mid x \mid\}) \leq$ *fcard* A
⟨*proof*⟩

lemma *fcard-pfsubset*: $A \mid\subseteq\mid B \Longrightarrow$ *fcard* $A <$ *fcard* $B \Longrightarrow A < B$
⟨*proof*⟩

51.5.10 *ffold*

context *comp-fun-commute*
begin
  lemmas *ffold-empty*[*simp*] = *fold-empty*[*Transfer.transferred*]

  lemma *ffold-finsert* [*simp*]:
    assumes $x \mid\notin\mid A$
    shows *ffold* $f\ z$ (*finsert* $x\ A$) $= f\ x$ (*ffold* $f\ z\ A$)
    ⟨*proof*⟩

  lemma *ffold-fun-left-comm*:
    $f\ x$ (*ffold* $f\ z\ A$) = *ffold* f $(f\ x\ z)\ A$
    ⟨*proof*⟩

  lemma *ffold-finsert2*:
    $x \mid\notin\mid A \Longrightarrow$ *ffold* $f\ z$ (*finsert* $x\ A$) = *ffold* f $(f\ x\ z)\ A$
    ⟨*proof*⟩

  lemma *ffold-rec*:
    assumes $x \mid\in\mid A$
    shows *ffold* $f\ z\ A = f\ x$ (*ffold* $f\ z$ $(A \mid-\mid \{\mid x \mid\})$)
    ⟨*proof*⟩

  lemma *ffold-finsert-fremove*:
    ffold $f\ z$ (*finsert* $x\ A$) $= f\ x$ (*ffold* $f\ z$ $(A \mid-\mid \{\mid x \mid\})$)

⟨*proof*⟩
end

lemma *ffold-fimage*:
assumes *inj-on g* (*fset A*)
shows *ffold f z* (*g* |‘| *A*) = *ffold* (*f* ∘ *g*) *z A*
⟨*proof*⟩

lemma *ffold-cong*:
assumes *comp-fun-commute f comp-fun-commute g*
⋀*x*. *x* |∈| *A* ⟹ *f x* = *g x*
and *s* = *t* **and** *A* = *B*
shows *ffold f s A* = *ffold g t B*
⟨*proof*⟩

context *comp-fun-idem*
begin

lemma *ffold-finsert-idem*:
ffold f z (*finsert x A*) = *f x* (*ffold f z A*)
⟨*proof*⟩

declare *ffold-finsert* [*simp del*] *ffold-finsert-idem* [*simp*]

lemma *ffold-finsert-idem2*:
ffold f z (*finsert x A*) = *ffold f* (*f x z*) *A*
⟨*proof*⟩

end

51.6 Choice in fsets

lemma *fset-choice*:
assumes ∀*x*. *x* |∈| *A* ⟶ (∃*y*. *P x y*)
shows ∃*f*. ∀*x*. *x* |∈| *A* ⟶ *P x* (*f x*)
⟨*proof*⟩

51.7 Induction and Cases rules for fsets

lemma *fset-exhaust* [*case-names empty insert*, *cases type*: *fset*]:
assumes *fempty-case*: *S* = {||} ⟹ *P*
and *finsert-case*: ⋀*x S′*. *S* = *finsert x S′* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *fset-induct* [*case-names empty insert*]:
assumes *fempty-case*: *P* {||}
and *finsert-case*: ⋀*x S*. *P S* ⟹ *P* (*finsert x S*)
shows *P S*
⟨*proof*⟩

lemma *fset-induct-stronger* [*case-names empty insert*, *induct type*: *fset*]:
assumes *empty-fset-case*: $P\ \{||\}$
and *insert-fset-case*: $\bigwedge x\ S.\ [\![x\ |\notin|\ S;\ P\ S]\!] \Longrightarrow P\ (\mathit{finsert}\ x\ S)$
shows $P\ S$
⟨*proof*⟩

lemma *fset-card-induct*:
assumes *empty-fset-case*: $P\ \{||\}$
and *card-fset-Suc-case*: $\bigwedge S\ T.\ \mathit{Suc}\ (\mathit{fcard}\ S) = (\mathit{fcard}\ T) \Longrightarrow P\ S \Longrightarrow P\ T$
shows $P\ S$
⟨*proof*⟩

lemma *fset-strong-cases*:
obtains $xs = \{||\}$
| *ys x* **where** $x\ |\notin|\ ys$ **and** $xs = \mathit{finsert}\ x\ ys$
⟨*proof*⟩

lemma *fset-induct2*:
$P\ \{||\}\ \{||\} \Longrightarrow$
$(\bigwedge x\ xs.\ x\ |\notin|\ xs \Longrightarrow P\ (\mathit{finsert}\ x\ xs)\ \{||\}) \Longrightarrow$
$(\bigwedge y\ ys.\ y\ |\notin|\ ys \Longrightarrow P\ \{||\}\ (\mathit{finsert}\ y\ ys)) \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys.\ [\![P\ xs\ ys;\ x\ |\notin|\ xs;\ y\ |\notin|\ ys]\!] \Longrightarrow P\ (\mathit{finsert}\ x\ xs)\ (\mathit{finsert}\ y\ ys)) \Longrightarrow$
$P\ xsa\ ysa$
⟨*proof*⟩

51.8 Setup for Lifting/Transfer

51.8.1 Relator and predicator properties

lift-definition *rel-fset* :: $('a \Rightarrow 'b \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{fset} \Rightarrow 'b\ \mathit{fset} \Rightarrow \mathit{bool}$ **is** *rel-set*
parametric *rel-set-transfer* ⟨*proof*⟩

lemma *rel-fset-alt-def*: *rel-fset* $R = (\lambda A\ B.\ (\forall x.\exists y.\ x|\in|A \longrightarrow y|\in|B \wedge R\ x\ y)$
$\wedge\ (\forall y.\ \exists x.\ y|\in|B \longrightarrow x|\in|A \wedge R\ x\ y))$
⟨*proof*⟩

lemma *finite-rel-set*:
assumes *fin*: *finite X finite Z*
assumes *R-S*: *rel-set* $(R\ OO\ S)\ X\ Z$
shows $\exists Y.$ *finite* $Y \wedge$ *rel-set* $R\ X\ Y \wedge$ *rel-set* $S\ Y\ Z$
⟨*proof*⟩

51.8.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes *lifting-syntax*
begin

lemmas *fempty-transfer* [*transfer-rule*] = *empty-transfer*[*Transfer.transferred*]

lemma *finsert-transfer* [*transfer-rule*]:
(*A* ===> *rel-fset A* ===> *rel-fset A*) *finsert finsert*
⟨*proof*⟩

lemma *funion-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *funion funion*
⟨*proof*⟩

lemma *ffUnion-transfer* [*transfer-rule*]:
(*rel-fset* (*rel-fset A*) ===> *rel-fset A*) *ffUnion ffUnion*
⟨*proof*⟩

lemma *fimage-transfer* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-fset A* ===> *rel-fset B*) *fimage fimage*
⟨*proof*⟩

lemma *fBall-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *op* =) ===> *op* =) *fBall fBall*
⟨*proof*⟩

lemma *fBex-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *op* =) ===> *op* =) *fBex fBex*
⟨*proof*⟩

lemma *fPow-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset* (*rel-fset A*)) *fPow fPow*
⟨*proof*⟩

lemma *rel-fset-transfer* [*transfer-rule*]:
((*A* ===> *B* ===> *op* =) ===> *rel-fset A* ===> *rel-fset B* ===> *op* =)
rel-fset rel-fset
⟨*proof*⟩

lemma *bind-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *rel-fset B*) ===> *rel-fset B*) *fbind fbind*
⟨*proof*⟩

Rules requiring bi-unique, bi-total or right-total relations

lemma *fmember-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*A* ===> *rel-fset A* ===> *op* =) (*op* |∈|) (*op* |∈|)
⟨*proof*⟩

lemma *finter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *finter finter*

⟨*proof*⟩

lemma *fminus-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) (*op* |−|) (*op* |−|)
⟨*proof*⟩

lemma *fsubset-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *op* =) (*op* |⊆|) (*op* |⊆|)
⟨*proof*⟩

lemma *fSup-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Sup Sup*
⟨*proof*⟩

lemma *fInf-transfer* [*transfer-rule*]:
assumes *bi-unique A* **and** *bi-total A*
shows (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Inf Inf*
⟨*proof*⟩

lemma *ffilter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows ((*A* ===> *op*=) ===> *rel-fset A* ===> *rel-fset A*) *ffilter ffilter*
⟨*proof*⟩

lemma *card-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-fset A* ===> *op* =) *fcard fcard*
⟨*proof*⟩

end

lifting-update *fset.lifting*
lifting-forget *fset.lifting*

51.9 BNF setup

context
includes *fset.lifting*
begin

lemma *rel-fset-alt*:
rel-fset R a b ⟷ (∀ *t* ∈ *fset a*. ∃ *u* ∈ *fset b*. *R t u*) ∧ (∀ *t* ∈ *fset b*. ∃ *u* ∈ *fset a*. *R u t*)
⟨*proof*⟩

lemma *fset-to-fset*: *finite A* ⟹ *fset* (*the-inv fset A*) = *A*

⟨*proof*⟩

lemma *rel-fset-aux*:
$(\forall\, t \in \mathit{fset}\ a.\ \exists\, u \in \mathit{fset}\ b.\ R\ t\ u) \wedge (\forall\, u \in \mathit{fset}\ b.\ \exists\, t \in \mathit{fset}\ a.\ R\ t\ u) \longleftrightarrow$
$((\mathit{BNF\text{-}Def.Grp}\ \{a.\ \mathit{fset}\ a \subseteq \{(a,\ b).\ R\ a\ b\}\}\ (\mathit{fimage}\ \mathit{fst}))^{-1-1}\ \mathit{OO}$
$\mathit{BNF\text{-}Def.Grp}\ \{a.\ \mathit{fset}\ a \subseteq \{(a,\ b).\ R\ a\ b\}\}\ (\mathit{fimage}\ \mathit{snd}))\ a\ b\ (\textbf{is}\ ?L = ?R)$
⟨*proof*⟩

bnf *'a fset*
map: *fimage*
sets: *fset*
bd: *natLeq*
wits: {||}
rel: *rel-fset*
⟨*proof*⟩

lemma *rel-fset-fset*: *rel-set* χ (*fset A1*) (*fset A2*) = *rel-fset* χ *A1 A2*
⟨*proof*⟩

end

lemmas [*simp*] = *fset.map-comp fset.map-id fset.set-map*

51.10 Size setup

context includes *fset.lifting* **begin**
lift-definition *size-fset* :: ('*a* ⇒ *nat*) ⇒ '*a fset* ⇒ *nat* **is** $\lambda f.$ *sum* (*Suc* ∘ *f*) ⟨*proof*⟩
end

instantiation *fset* :: (*type*) *size* **begin**
definition *size-fset* **where**
size-fset-overloaded-def: *size-fset* = *FSet.size-fset* (λ-. *0*)
instance ⟨*proof*⟩
end

lemmas *size-fset-simps*[*simp*] =
size-fset-def[*THEN meta-eq-to-obj-eq*, *THEN fun-cong*, *THEN fun-cong*,
unfolded map-fun-def comp-def id-apply]

lemmas *size-fset-overloaded-simps*[*simp*] =
size-fset-simps[*of* λ-. *0*, *unfolded add-0-left add-0-right*,
folded size-fset-overloaded-def]

lemma *fset-size-o-map*: *inj f* ⟹ *size-fset g* ∘ *fimage f* = *size-fset* (*g* ∘ *f*)
⟨*proof*⟩
including *fset.lifting* ⟨*proof*⟩

⟨*ML*⟩

lifting-update *fset.lifting*
lifting-forget *fset.lifting*

51.11 Advanced relator customization

lemma *rel-set-rel-sum*[*simp*]:
rel-set (*rel-sum* χ φ) *A1 A2* $\longleftrightarrow$
rel-set χ (*Inl* $-$‘ *A1*) (*Inl* $-$‘ *A2*) $\wedge$ *rel-set* φ (*Inr* $-$‘ *A1*) (*Inr* $-$‘ *A2*)
(**is** *?L* $\longleftrightarrow$ *?Rl* $\wedge$ *?Rr*)
⟨*proof*⟩

51.12 Quickcheck setup

Setup adapted from sets.

notation *Quickcheck-Exhaustive.orelse* (**infixr** *orelse 55*)

definition (**in** *term-syntax*) [*code-unfold*]:
valterm-femptyset = *Code-Evaluation.valtermify* ({||} :: (′*a* :: *typerep*) *fset*)

definition (**in** *term-syntax*) [*code-unfold*]:
valtermify-finsert x s = *Code-Evaluation.valtermify finsert* {·} (*x* :: (′*a* :: *typerep* ∗ -)) {·} *s*

instantiation *fset* :: (*exhaustive*) *exhaustive*
begin

fun *exhaustive-fset* **where**
exhaustive-fset f i = (*if i* = *0 then None else* (*f* {||} *orelse exhaustive-fset* (λ*A*. *f A orelse Quickcheck-Exhaustive.exhaustive* (λ*x*. *if x* |∈| *A then None else f* (*finsert x A*)) (*i* − *1*)) (*i* − *1*)))

instance ⟨*proof*⟩

end

instantiation *fset* :: (*full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-fset* **where**
full-exhaustive-fset f i = (*if i* = *0 then None else* (*f valterm-femptyset orelse full-exhaustive-fset* (λ*A*. *f A orelse Quickcheck-Exhaustive.full-exhaustive* (λ*x*. *if fst x* |∈| *fst A then None else f* (*valtermify-finsert x A*)) (*i* − *1*)) (*i* − *1*)))

instance ⟨*proof*⟩

end

no-notation *Quickcheck-Exhaustive.orelse* (**infixr** *orelse 55*)

notation *scomp* (**infixl** $\circ\!\!\rightarrow$ *60*)

instantiation *fset* :: (*random*) *random*
begin

fun *random-aux-fset* :: *natural* ⇒ *natural* ⇒ *natural* × *natural* ⇒ ('*a fset* × (*unit* ⇒ *term*)) × *natural* × *natural* **where**
random-aux-fset 0 j = *Quickcheck-Random.collapse* (*Random.select-weight* [(*1*, *Pair valterm-femptyset*)]) |
random-aux-fset (*Code-Numeral.Suc i*) *j* =
Quickcheck-Random.collapse (*Random.select-weight*
[(*1*, *Pair valterm-femptyset*),
(*Code-Numeral.Suc i*,
Quickcheck-Random.random j $\circ\!\!\rightarrow$ (λ*x*. *random-aux-fset i j* $\circ\!\!\rightarrow$ (λ*s*. *Pair* (*valtermify-finsert x s*))))])

lemma [*code*]:
random-aux-fset i j =
Quickcheck-Random.collapse (*Random.select-weight* [(*1*, *Pair valterm-femptyset*),
(*i*, *Quickcheck-Random.random j* $\circ\!\!\rightarrow$ (λ*x*. *random-aux-fset* (*i* − *1*) *j* $\circ\!\!\rightarrow$ (λ*s*. *Pair* (*valtermify-finsert x s*))))])
⟨*proof*⟩

definition *random-fset i* = *random-aux-fset i i*

instance ⟨*proof*⟩

end

no-notation *scomp* (**infixl** $\circ\!\!\rightarrow$ *60*)

end

theory *Transfer-Debug*
imports *Main* ~~/*src*/*HOL*/*Library*/*FSet*
begin

context
includes *fset.lifting*
begin

51.13 1. A missing transfer rule

declare *fmember.transfer*[*transfer-rule del*] *fmember-transfer*[*transfer-rule del*]

lemma $(A \mid\subseteq\mid B) = \mathit{fBall}\ A\ (\lambda x.\ x \mid\in\mid B)$
⟨*proof*⟩

lemma $(A \mid\subseteq\mid B) = \mathit{fBall}\ A\ (\lambda x.\ x \mid\in\mid B)$
⟨*proof*⟩

lemma [*transfer-rule*]: $\mathit{bi\text{-}unique}\ A \Longrightarrow \mathit{rel\text{-}fun}\ A\ (\mathit{rel\text{-}fun}\ (\mathit{pcr\text{-}fset}\ A)\ \mathit{op} =)\ \mathit{op} \in \mathit{op} \mid\in\mid$
⟨*proof*⟩

lemma $(A \mid\subseteq\mid B) = \mathit{fBall}\ A\ (\lambda x.\ x \mid\in\mid B)$
⟨*proof*⟩

lemma $(A \mid\subseteq\mid B) = \mathit{fBall}\ A\ (\lambda x.\ x \mid\in\mid B)$
⟨*proof*⟩

51.14 2. Unwanted instantiation of a transfer relation variable

lemma *finite* $(\mathit{UNIV} :: 'a{::}\mathit{finite}\ \mathit{fset}\ \mathit{set})$
⟨*proof*⟩

lemma *finite* $(\mathit{UNIV} :: 'a{::}\mathit{finite}\ \mathit{fset}\ \mathit{set})$
⟨*proof*⟩

lemma *finite* $(\mathit{UNIV} :: 'a{::}\mathit{finite}\ \mathit{fset}\ \mathit{set})$
⟨*proof*⟩

end

lifting-forget *fset.lifting*
declare *fmember-transfer*[*transfer-rule*]

end

52 Using the transfer method between nat and int

theory *Transfer-Int-Nat*
imports *GCD*
begin

52.1 Correspondence relation

definition $ZN :: \mathit{int} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$

where $ZN = (\lambda z\ n.\ z = of\text{-}nat\ n)$

52.2 Transfer domain rules

lemma *Domainp-ZN* [*transfer-domain-rule*]: $Domainp\ ZN = (\lambda x.\ x \geq 0)$
⟨*proof*⟩

52.3 Transfer rules

context includes *lifting-syntax*
begin

lemma *bi-unique-ZN* [*transfer-rule*]: *bi-unique ZN*
⟨*proof*⟩

lemma *right-total-ZN* [*transfer-rule*]: *right-total ZN*
⟨*proof*⟩

lemma *ZN-0* [*transfer-rule*]: *ZN 0 0*
⟨*proof*⟩

lemma *ZN-1* [*transfer-rule*]: *ZN 1 1*
⟨*proof*⟩

lemma *ZN-add* [*transfer-rule*]: $(ZN\ ===>\ ZN\ ===>\ ZN)\ (op\ +)\ (op\ +)$
⟨*proof*⟩

lemma *ZN-mult* [*transfer-rule*]: $(ZN\ ===>\ ZN\ ===>\ ZN)\ (op\ *)\ (op\ *)$
⟨*proof*⟩

lemma *ZN-diff* [*transfer-rule*]: $(ZN\ ===>\ ZN\ ===>\ ZN)\ tsub\ (op\ -)$
⟨*proof*⟩

lemma *ZN-power* [*transfer-rule*]: $(ZN\ ===>\ op =\ ===>\ ZN)\ (op\ \hat{})\ (op\ \hat{})$
⟨*proof*⟩

lemma *ZN-nat-id* [*transfer-rule*]: $(ZN\ ===>\ op =)\ nat\ id$
⟨*proof*⟩

lemma *ZN-id-int* [*transfer-rule*]: $(ZN\ ===>\ op =)\ id\ int$
⟨*proof*⟩

lemma *ZN-All* [*transfer-rule*]:
$((ZN\ ===>\ op =)\ ===>\ op =)\ (Ball\ \{0..\})\ All$
⟨*proof*⟩

lemma *ZN-transfer-forall* [*transfer-rule*]:
$((ZN\ ===>\ op =)\ ===>\ op =)\ (transfer\text{-}bforall\ (\lambda x.\ 0 \leq x))\ transfer\text{-}forall$
⟨*proof*⟩

lemma *ZN-Ex* [*transfer-rule*]: ((*ZN* ===> *op* =) ===> *op* =) (*Bex* {*0*..}) *Ex*
⟨*proof*⟩

lemma *ZN-le* [*transfer-rule*]: (*ZN* ===> *ZN* ===> *op* =) (*op* ≤) (*op* ≤)
⟨*proof*⟩

lemma *ZN-less* [*transfer-rule*]: (*ZN* ===> *ZN* ===> *op* =) (*op* <) (*op* <)
⟨*proof*⟩

lemma *ZN-eq* [*transfer-rule*]: (*ZN* ===> *ZN* ===> *op* =) (*op* =) (*op* =)
⟨*proof*⟩

lemma *ZN-Suc* [*transfer-rule*]: (*ZN* ===> *ZN*) (λ*x*. *x* + *1*) *Suc*
⟨*proof*⟩

lemma *ZN-numeral* [*transfer-rule*]:
(*op* = ===> *ZN*) *numeral numeral*
⟨*proof*⟩

lemma *ZN-dvd* [*transfer-rule*]: (*ZN* ===> *ZN* ===> *op* =) (*op dvd*) (*op dvd*)
⟨*proof*⟩

lemma *ZN-div* [*transfer-rule*]: (*ZN* ===> *ZN* ===> *ZN*) (*op div*) (*op div*)
⟨*proof*⟩

lemma *ZN-mod* [*transfer-rule*]: (*ZN* ===> *ZN* ===> *ZN*) (*op mod*) (*op mod*)
⟨*proof*⟩

lemma *ZN-gcd* [*transfer-rule*]: (*ZN* ===> *ZN* ===> *ZN*) *gcd gcd*
⟨*proof*⟩

lemma *ZN-atMost* [*transfer-rule*]:
(*ZN* ===> *rel-set ZN*) (*atLeastAtMost 0*) *atMost*
⟨*proof*⟩

lemma *ZN-atLeastAtMost* [*transfer-rule*]:
(*ZN* ===> *ZN* ===> *rel-set ZN*) *atLeastAtMost atLeastAtMost*
⟨*proof*⟩

lemma *ZN-sum* [*transfer-rule*]:
bi-unique A ⟹ ((*A* ===> *ZN*) ===> *rel-set A* ===> *ZN*) *sum sum*
⟨*proof*⟩

For derived operations, we can use the *transfer-prover* method to help generate transfer rules.

lemma *ZN-sum-list* [*transfer-rule*]: (*list-all2 ZN* ===> *ZN*) *sum-list sum-list*
⟨*proof*⟩

end

52.4 Transfer examples

lemma
assumes $\bigwedge i{::}int.\ 0 \le i \Longrightarrow i + 0 = i$
shows $\bigwedge i{::}nat.\ i + 0 = i$
⟨*proof*⟩

lemma
assumes $\bigwedge i\ k{::}int.\ [\![0 \le i;\ 0 \le k;\ i < k]\!] \Longrightarrow \exists j{\in}\{0..\}.\ i + j = k$
shows $\bigwedge i\ k{::}nat.\ i < k \Longrightarrow \exists j.\ i + j = k$
⟨*proof*⟩

lemma
assumes $\forall x{\in}\{0{::}int..\}.\ \forall y{\in}\{0..\}.\ x * y\ div\ y = x$
shows $\forall x\ y :: nat.\ x * y\ div\ y = x$
⟨*proof*⟩

lemma
assumes $\bigwedge m\ n{::}int.\ [\![0 \le m;\ 0 \le n;\ m * n = 0]\!] \Longrightarrow m = 0 \vee n = 0$
shows $m * n = (0{::}nat) \Longrightarrow m = 0 \vee n = 0$
⟨*proof*⟩

lemma
assumes $\forall x{\in}\{0{::}int..\}.\ \exists y{\in}\{0..\}.\ \exists z{\in}\{0..\}.\ x + 3 * y = 5 * z$
shows $\forall x{::}nat.\ \exists y\ z.\ x + 3 * y = 5 * z$
⟨*proof*⟩

The *fixing* option prevents generalization over the free variable n, allowing the local transfer rule to be used.

lemma
assumes [*transfer-rule*]: *ZN x n*
assumes $\forall i{\in}\{0..\}.\ i < x \longrightarrow 2 * i < 3 * x$
shows $\forall i.\ i < n \longrightarrow 2 * i < 3 * n$
⟨*proof*⟩

lemma
assumes *gcd* (*2^i*) (*3^j*) = (*1*::*int*)
shows *gcd* (*2^i*) (*3^j*) = (*1*::*nat*)
⟨*proof*⟩

lemma
assumes $\bigwedge x\ y\ z{::}int.\ [\![0 \le x;\ 0 \le y;\ 0 \le z]\!] \Longrightarrow$
sum-list [*x*, *y*, *z*] = *0* ⟷ *list-all* (λ*x*. *x* = *0*) [*x*, *y*, *z*]
shows *sum-list* [*x*, *y*, *z*] = (*0*::*nat*) ⟷ *list-all* (λ*x*. *x* = *0*) [*x*, *y*, *z*]
⟨*proof*⟩

Quantifiers over higher types (e.g. *nat list*) are transferred to a readable formula thanks to the transfer domain rule *Domainp ZN = op ≤ 0*

lemma

assumes $\bigwedge xs{::}int\ list.\ list\text{-}all\ (\lambda x.\ x \geq 0)\ xs \Longrightarrow$
$(sum\text{-}list\ xs = 0) = list\text{-}all\ (\lambda x.\ x = 0)\ xs$
shows $sum\text{-}list\ xs = (0{::}nat) \longleftrightarrow list\text{-}all\ (\lambda x.\ x = 0)\ xs$
⟨*proof*⟩

Equality on a higher type can be transferred if the relations involved are bi-unique.

lemma
assumes $\bigwedge xs{::}int\ list.\ [\![list\text{-}all\ (\lambda x.\ x \geq 0)\ xs;\ xs \neq []]\!] \Longrightarrow$
$sum\text{-}list\ xs < sum\text{-}list\ (map\ (\lambda x.\ x + 1)\ xs)$
shows $xs \neq [] \Longrightarrow sum\text{-}list\ xs < sum\text{-}list\ (map\ Suc\ xs)$
⟨*proof*⟩

end

53 Various examples for transfer procedure

theory *Transfer-Ex*
imports *Main Transfer-Int-Nat*
begin

lemma *ex1*: $(x{::}nat) + y = y + x$
⟨*proof*⟩

lemma $0 \leq (y{::}int) \Longrightarrow 0 \leq (x{::}int) \Longrightarrow x + y = y + x$
⟨*proof*⟩

lemma $0 \leq (x{::}int) \Longrightarrow 0 \leq (y{::}int) \Longrightarrow x + y = y + x$
⟨*proof*⟩

lemma *ex2*: $(a{::}nat)\ div\ b * b + a\ mod\ b = a$
⟨*proof*⟩

lemma $0 \leq (b{::}int) \Longrightarrow 0 \leq (a{::}int) \Longrightarrow a\ div\ b * b + a\ mod\ b = a$
⟨*proof*⟩

lemma $0 \leq (a{::}int) \Longrightarrow 0 \leq (b{::}int) \Longrightarrow a\ div\ b * b + a\ mod\ b = a$
⟨*proof*⟩

lemma *ex3*: $ALL\ (x{::}nat).\ ALL\ y.\ EX\ z.\ z >= x + y$
⟨*proof*⟩

lemma $\forall x{\geq}0{::}int.\ \forall y{\geq}0.\ \exists z{\geq}0.\ x + y \leq z$
⟨*proof*⟩

lemma $\forall x{::}int{\in}\{0..\}.\ \forall y{\in}\{0..\}.\ \exists z{\in}\{0..\}.\ x + y \leq z$

⟨*proof*⟩

lemma *ex4*: $(x::nat) >= y \Longrightarrow (x - y) + y = x$
⟨*proof*⟩

lemma $0 \leq (x::int) \Longrightarrow 0 \leq (y::int) \Longrightarrow y \leq x \Longrightarrow tsub\ x\ y + y = x$
⟨*proof*⟩

lemma $0 \leq (y::int) \Longrightarrow 0 \leq (x::int) \Longrightarrow y \leq x \Longrightarrow tsub\ x\ y + y = x$
⟨*proof*⟩

lemma *ex5*: $(2::nat) * \sum \{..n\} = n * (n + 1)$
⟨*proof*⟩

lemma $0 \leq (n::int) \Longrightarrow 2 * \sum \{0..n\} = n * (n + 1)$
⟨*proof*⟩

lemma $0 \leq (n::int) \Longrightarrow 2 * \sum \{0..n\} = n * (n + 1)$
⟨*proof*⟩

lemma $0 \leq (n::nat) \Longrightarrow 2 * \sum \{0..n\} = n * (n + 1)$
⟨*proof*⟩

lemma $0 \leq (n::nat) \Longrightarrow 2 * \sum \{..n\} = n * (n + 1)$
⟨*proof*⟩

end

54 Simple example for table-based implementation of the reflexive transitive closure

theory *Transitive-Closure-Table-Ex*
imports ~~/*src/HOL/Library/Transitive-Closure-Table*
begin

datatype $ty = A \mid B \mid C$

inductive *test* :: $ty \Rightarrow ty \Rightarrow bool$
where
  test A B
| *test B A*
| *test B C*

Invoking with the predicate compiler and the generic code generator

code-pred *test* ⟨*proof*⟩

values $\{x.\ test^{**}\ A\ C\}$
values $\{x.\ test^{**}\ C\ A\}$
values $\{x.\ test^{**}\ A\ x\}$
values $\{x.\ test^{**}\ x\ C\}$

value $test^{**}\ A\ C$
value $test^{**}\ C\ A$

end

55 Divergence of the Harmonic Series

theory *HarmonicSeries*
imports *Complex-Main*
begin

55.1 Abstract

The following document presents a proof of the Divergence of Harmonic Series theorem formalised in the Isabelle/Isar theorem proving system.

Theorem: The series $\sum_{n=1}^{\infty} \frac{1}{n}$ does not converge to any number.

Informal Proof: The informal proof is based on the following auxillary lemmas:

- aux: $\sum_{n=2^m-1}^{2^m} \frac{1}{n} \geq \frac{1}{2}$
- aux2: $\sum_{n=1}^{2^M} \frac{1}{n} = 1 + \sum_{m=1}^{M} \sum_{n=2^m-1}^{2^m} \frac{1}{n}$

From *aux* and *aux2* we can deduce that $\sum_{n=1}^{2^M} \frac{1}{n} \geq 1 + \frac{M}{2}$ for all M. Now for contradiction, assume that $\sum_{n=1}^{\infty} \frac{1}{n} = s$ for some s. Because $\forall n. \frac{1}{n} > 0$ all the partial sums in the series must be less than s. However with our deduction above we can choose $N > 2 * s - 2$ and thus $\sum_{n=1}^{2^N} \frac{1}{n} > s$. This leads to a contradiction and hence $\sum_{n=1}^{\infty} \frac{1}{n}$ is not summable. QED.

55.2 Formal Proof

lemma *two-pow-sub*:
$0 < m \Longrightarrow (2::nat)\hat{}m - 2\hat{}(m - 1) = 2\hat{}(m - 1)$
⟨*proof*⟩

We first prove the following auxillary lemma. This lemma simply states that the finite sums: $\frac{1}{2}$, $\frac{1}{3} + \frac{1}{4}$, $\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}$ etc. are all greater than or equal to $\frac{1}{2}$. We do this by observing that each term in the sum is greater than or equal to the last term, e.g. $\frac{1}{3} > \frac{1}{4}$ and thus $\frac{1}{3} + \frac{1}{4} > \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$.

lemma *harmonic-aux*:

$\forall m{>}0.\ (\sum n{\in}\{(2{::}nat)\ \hat{}\ (m\ -\ 1){+}1..2\ \hat{}\ m\}.\ 1/real\ n) \geq 1/2$
(**is** $\forall m{>}0.\ (\sum n{\in}(?S\ m).\ 1/real\ n) \geq 1/2$)
⟨*proof*⟩

We then show that the sum of a finite number of terms from the harmonic series can be regrouped in increasing powers of 2. For example: $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} = 1 + (\frac{1}{2}) + (\frac{1}{3} + \frac{1}{4}) + (\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8})$.

lemma *harmonic-aux2* [*rule-format*]:
$0{<}M \Longrightarrow (\sum n{\in}\{1..(2{::}nat)\ \hat{}\ M\}.\ 1/real\ n) =$
$(1\ +\ (\sum m{\in}\{1..M\}.\ \sum n{\in}\{(2{::}nat)\ \hat{}\ (m\ -\ 1){+}1..2\ \hat{}\ m\}.\ 1/real\ n))$
(**is** $0{<}M \Longrightarrow ?LHS\ M = ?RHS\ M$)
⟨*proof*⟩

Using *harmonic-aux* and *harmonic-aux2* we now show that each group sum is greater than or equal to $\frac{1}{2}$ and thus the finite sum is bounded below by a value proportional to the number of elements we choose.

lemma *harmonic-aux3* [*rule-format*]:
shows $\forall (M{::}nat).\ (\sum n{\in}\{1..(2{::}nat)\ \hat{}\ M\}.\ 1\ /\ real\ n) \geq 1\ +\ (real\ M)/2$
(**is** $\forall M.\ ?P\ M \geq$ -)
⟨*proof*⟩

The final theorem shows that as we take more and more elements (see *harmonic-aux3*) we get an ever increasing sum. By assuming the sum converges, the lemma *sum-less-suminf* ($[\![summable\ ?f; \forall m{\geq}?n.\ (0{::}?'a) < ?f\ m]\!] \Longrightarrow sum\ ?f\ \{..{<}?n\} < suminf\ ?f$) states that each sum is bounded above by the series' limit. This contradicts our first statement and thus we prove that the harmonic series is divergent.

theorem *DivergenceOfHarmonicSeries*:
shows $\neg summable\ (\lambda n.\ 1/real\ (Suc\ n))$
(**is** $\neg summable\ ?f$)
⟨*proof*⟩

end

56 Examples for the 'refute' command

theory *Refute-Examples*
imports ~~/*src*/*HOL*/*Library*/*Refute*
begin

refute-params [*satsolver* = *cdclite*]

lemma $P \wedge Q$
⟨*proof*⟩

56.1 Examples and Test Cases

56.1.1 Propositional logic

lemma *True*
refute [*expect* = *none*]
⟨*proof*⟩

lemma *False*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma ~ *P*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P* & *Q*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P* | *Q*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P* ⟶ *Q*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma (*P*::*bool*) = *Q*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma (*P* | *Q*) ⟶ (*P* & *Q*)
refute [*expect* = *genuine*]
⟨*proof*⟩

56.1.2 Predicate logic

lemma *P x y z*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P x y* ⟶ *P y x*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ (f\ (f\ x)) \longrightarrow P\ x \longrightarrow P\ (f\ x)$
refute [*expect* = *genuine*]
⟨*proof*⟩

56.1.3 Equality

lemma $P = \mathit{True}$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P = \mathit{False}$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $x = y$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $f\ x = g\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(f{::}'a{\Rightarrow}'b) = g$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(f{::}('d{\Rightarrow}'d){\Rightarrow}('c{\Rightarrow}'d)) = g$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *distinct* [*a*, *b*]

⟨*proof*⟩

56.1.4 First-Order Logic

lemma $\exists\, x.\ P\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $\forall\, x.\ P\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $\exists!x.\ P\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *Ex P*
refute [*expect* = *genuine*]

⟨*proof*⟩

lemma *All P*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *Ex1 P*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(\exists\, x.\ P\ x) \longrightarrow (\forall\, x.\ P\ x)$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(\forall\, x.\ \exists\, y.\ P\ x\ y) \longrightarrow (\exists\, y.\ \forall\, x.\ P\ x\ y)$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(\exists\, x.\ P\ x) \longrightarrow (\exists !x.\ P\ x)$
refute [*expect* = *genuine*]
⟨*proof*⟩

A true statement (also testing names of free and bound variables being identical)

lemma $(\forall\, x\ y.\ P\ x\ y \longrightarrow P\ y\ x) \longrightarrow (\forall\, x.\ P\ x\ y) \longrightarrow P\ y\ x$
refute [*maxsize* = *4*, *expect* = *none*]
⟨*proof*⟩

"A type has at most 4 elements."

lemma $a{=}b \mid a{=}c \mid a{=}d \mid a{=}e \mid b{=}c \mid b{=}d \mid b{=}e \mid c{=}d \mid c{=}e \mid d{=}e$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $\forall\, a\ b\ c\ d\ e.\ a{=}b \mid a{=}c \mid a{=}d \mid a{=}e \mid b{=}c \mid b{=}d \mid b{=}e \mid c{=}d \mid c{=}e \mid d{=}e$
refute [*expect* = *genuine*]
⟨*proof*⟩

"Every reflexive and symmetric relation is transitive."

lemma $[\![\ \forall\, x.\ P\ x\ x;\ \forall\, x\ y.\ P\ x\ y \longrightarrow P\ y\ x\ ]\!] \Longrightarrow P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z$
refute [*expect* = *genuine*]
⟨*proof*⟩

The "Drinker's theorem" ...

lemma $\exists\, x.\ f\ x = g\ x \longrightarrow f = g$
refute [*maxsize* = *4*, *expect* = *none*]
⟨*proof*⟩

... and an incorrect version of it

lemma $(\exists x.\ f\ x = g\ x) \longrightarrow f = g$
refute [*expect* = *genuine*]
⟨*proof*⟩

"Every function has a fixed point."

lemma $\exists x.\ f\ x = x$
refute [*expect* = *genuine*]
⟨*proof*⟩

"Function composition is commutative."

lemma $f\ (g\ x) = g\ (f\ x)$
refute [*expect* = *genuine*]
⟨*proof*⟩

"Two functions that are equivalent wrt. the same predicate 'P' are equal."

lemma $((P{::}('a \Rightarrow 'b) \Rightarrow bool)\ f = P\ g) \longrightarrow (f\ x = g\ x)$
refute [*expect* = *genuine*]
⟨*proof*⟩

56.1.5 Higher-Order Logic

lemma $\exists P.\ P$
refute [*expect* = *none*]
⟨*proof*⟩

lemma $\forall P.\ P$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $\exists! P.\ P$
refute [*expect* = *none*]
⟨*proof*⟩

lemma $\exists! P.\ P\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ Q \mid Q\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $x \neq All$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $x \neq Ex$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $x \neq Ex1$
refute [*expect* = *genuine*]
⟨*proof*⟩

"The transitive closure 'T' of an arbitrary relation 'P' is non-empty."

definition *trans* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
$trans\ P == (ALL\ x\ y\ z.\ P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z)$

definition *subset* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
$subset\ P\ Q == (ALL\ x\ y.\ P\ x\ y \longrightarrow Q\ x\ y)$

definition *trans-closure* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
$trans\text{-}closure\ P\ Q == (subset\ Q\ P)\ \&\ (trans\ P)\ \&\ (ALL\ R.\ subset\ Q\ R \longrightarrow trans\ R \longrightarrow subset\ P\ R)$

lemma $trans\text{-}closure\ T\ P \longrightarrow (\exists\, x\ y.\ T\ x\ y)$
refute [*expect* = *genuine*]
⟨*proof*⟩

"Every surjective function is invertible."

lemma $(\forall\, y.\ \exists\, x.\ y = f\ x) \longrightarrow (\exists\, g.\ \forall\, x.\ g\ (f\ x) = x)$
refute [*expect* = *genuine*]
⟨*proof*⟩

"Every invertible function is surjective."

lemma $(\exists\, g.\ \forall\, x.\ g\ (f\ x) = x) \longrightarrow (\forall\, y.\ \exists\, x.\ y = f\ x)$
refute [*expect* = *genuine*]
⟨*proof*⟩

Every point is a fixed point of some function.

lemma $\exists\, f.\ f\ x = x$
refute [*maxsize* = *4*, *expect* = *none*]
⟨*proof*⟩

Axiom of Choice: first an incorrect version ...

lemma $(\forall\, x.\ \exists\, y.\ P\ x\ y) \longrightarrow (\exists!\, f.\ \forall\, x.\ P\ x\ (f\ x))$
refute [*expect* = *genuine*]
⟨*proof*⟩

... and now two correct ones

lemma $(\forall\, x.\ \exists\, y.\ P\ x\ y) \longrightarrow (\exists\, f.\ \forall\, x.\ P\ x\ (f\ x))$
refute [*maxsize* = *4*, *expect* = *none*]
⟨*proof*⟩

lemma $(\forall\, x.\ \exists!\, y.\ P\ x\ y) \longrightarrow (\exists!\, f.\ \forall\, x.\ P\ x\ (f\ x))$
refute [*maxsize* = *2*, *expect* = *none*]
⟨*proof*⟩

56.1.6 Meta-logic

lemma $!!x.\ P\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $f\ x == g\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P \Longrightarrow Q$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $[\![\ P;\ Q;\ R\ ]\!] \Longrightarrow S$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(x == Pure.all) \Longrightarrow False$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(x == (op ==)) \Longrightarrow False$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(x == (op \Longrightarrow)) \Longrightarrow False$
refute [*expect* = *genuine*]
⟨*proof*⟩

56.1.7 Schematic variables

schematic-goal *?P*
refute [*expect* = *none*]
⟨*proof*⟩

schematic-goal $x = ?y$
refute [*expect* = *none*]
⟨*proof*⟩

56.1.8 Abstractions

lemma $(\lambda x.\ x) = (\lambda x.\ y)$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(\lambda f.\ f\ x) = (\lambda f.\ True)$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(\lambda x.\ x) = (\lambda y.\ y)$
refute
⟨*proof*⟩

56.1.9 Sets

lemma $P\ (A{::}'a\ set)$
refute
⟨*proof*⟩

lemma $P\ (A{::}'a\ set\ set)$
refute
⟨*proof*⟩

lemma $\{x.\ P\ x\} = \{y.\ P\ y\}$
refute
⟨*proof*⟩

lemma $x : \{x.\ P\ x\}$
refute
⟨*proof*⟩

lemma $P\ op$:
refute
⟨*proof*⟩

lemma $P\ (op{:}\ x)$
refute
⟨*proof*⟩

lemma $P\ Collect$
refute
⟨*proof*⟩

lemma $A\ Un\ B = A\ Int\ B$
refute
⟨*proof*⟩

lemma $(A\ Int\ B)\ Un\ C = (A\ Un\ C)\ Int\ B$
refute
⟨*proof*⟩

lemma $Ball\ A\ P \longrightarrow Bex\ A\ P$
refute
⟨*proof*⟩

56.1.10 undefined

lemma *undefined*
refute [*expect* = *genuine*]

⟨*proof*⟩

lemma *P undefined*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *undefined x*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *undefined undefined*
refute [*expect* = *genuine*]
⟨*proof*⟩

56.1.11 The

lemma *The P*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P The*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P* (*The P*)
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma (*THE x*. *x*=*y*) = *z*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *Ex P* ⟶ *P* (*The P*)
refute [*expect* = *genuine*]
⟨*proof*⟩

56.1.12 Eps

lemma *Eps P*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P Eps*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P* (*Eps P*)
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $(SOME\ x.\ x{=}y) = z$
refute [$expect = genuine$]
⟨*proof*⟩

lemma $Ex\ P \longrightarrow P\ (Eps\ P)$
refute [$maxsize = 3$, $expect = none$]
⟨*proof*⟩

56.1.13 Subtypes (typedef), typedecl

A completely unspecified non-empty subset of $'a$:

definition $myTdef = insert\ (undefined{::}'a)\ (undefined{::}'a\ set)$

typedef $'a\ myTdef = myTdef :: 'a\ set$
⟨*proof*⟩

lemma $(x{::}'a\ myTdef) = y$
refute
⟨*proof*⟩

typedecl *myTdecl*

definition $T\text{-}bij = \{(f{::}'a{\Rightarrow}'a).\ \forall\, y.\ \exists!x.\ f\ x = y\}$

typedef $'a\ T\text{-}bij = T\text{-}bij :: ('a \Rightarrow 'a)\ set$
⟨*proof*⟩

lemma $P\ (f{::}(myTdecl\ myTdef)\ T\text{-}bij)$
refute
⟨*proof*⟩

56.1.14 Inductive datatypes

unit

lemma $P\ (x{::}unit)$
refute [$expect = genuine$]
⟨*proof*⟩

lemma $\forall\, x{::}unit.\ P\ x$
refute [$expect = genuine$]
⟨*proof*⟩

lemma $P\ ()$
refute [$expect = genuine$]
⟨*proof*⟩

lemma $P\ (case\ x\ of\ ()\ \Rightarrow u)$
refute [$expect = genuine$]

⟨*proof*⟩

option

lemma $P\ (x::'a\ option)$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $\forall x::'a\ option.\ P\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ None$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ (Some\ x)$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ (case\ x\ of\ None \Rightarrow n \mid Some\ u \Rightarrow s\ u)$
refute [*expect* = *genuine*]
⟨*proof*⟩

*

lemma $P\ (x::'a*'b)$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $\forall x::'a*'b.\ P\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ (x,\ y)$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ (fst\ x)$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ (snd\ x)$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ Pair$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ (case\ x\ of\ Pair\ a\ b \Rightarrow p\ a\ b)$

refute [*expect* = *genuine*]
⟨*proof*⟩

+

lemma *P* (*x*::'*a*+'*b*)
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma ∀ *x*::'*a*+'*b*. *P x*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P* (*Inl x*)
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P* (*Inr x*)
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P Inl*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P* (*case x of Inl a* ⇒ *l a* | *Inr b* ⇒ *r b*)
refute [*expect* = *genuine*]
⟨*proof*⟩

Non-recursive datatypes

datatype *T1* = *A* | *B*

lemma *P* (*x*::*T1*)
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma ∀ *x*::*T1*. *P x*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P A*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *P B*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *rec-T1 a b A* = *a*
refute [*expect* = *none*]

⟨*proof*⟩

lemma *rec-T1 a b B = b*
refute [*expect = none*]
⟨*proof*⟩

lemma *P (rec-T1 a b x)*
refute [*expect = genuine*]
⟨*proof*⟩

lemma *P (case x of A ⇒ a | B ⇒ b)*
refute [*expect = genuine*]
⟨*proof*⟩

datatype *'a T2 = C T1 | D 'a*

lemma *P (x::'a T2)*
refute [*expect = genuine*]
⟨*proof*⟩

lemma *∀x::'a T2. P x*
refute [*expect = genuine*]
⟨*proof*⟩

lemma *P D*
refute [*expect = genuine*]
⟨*proof*⟩

lemma *rec-T2 c d (C x) = c x*
refute [*maxsize = 4, expect = none*]
⟨*proof*⟩

lemma *rec-T2 c d (D x) = d x*
refute [*maxsize = 4, expect = none*]
⟨*proof*⟩

lemma *P (rec-T2 c d x)*
refute [*expect = genuine*]
⟨*proof*⟩

lemma *P (case x of C u ⇒ c u | D v ⇒ d v)*
refute [*expect = genuine*]
⟨*proof*⟩

datatype *('a,'b) T3 = E 'a ⇒ 'b*

lemma *P (x::('a,'b) T3)*
refute [*expect = genuine*]
⟨*proof*⟩

lemma $\forall x{::}('a,'b)\ T3.\ P\ x$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma $P\ E$
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma *rec-T3 e* $(E\ x) = e\ x$
refute [*maxsize* = *2*, *expect* = *none*]
⟨*proof*⟩

lemma P (*rec-T3 e x*)
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma P (*case x of E f* $\Rightarrow$ *e f*)
refute [*expect* = *genuine*]
⟨*proof*⟩

Recursive datatypes

nat

lemma $P\ (x{::}nat)$
refute [*expect* = *potential*]
⟨*proof*⟩

lemma $\forall x{::}nat.\ P\ x$
refute [*expect* = *potential*]
⟨*proof*⟩

lemma P (*Suc 0*)
refute [*expect* = *potential*]
⟨*proof*⟩

lemma *P Suc*
refute [*maxsize* = *3*, *expect* = *none*]
— *Suc* is a partial function (regardless of the size of the model), hence *P Suc* is undefined and no model will be found
⟨*proof*⟩

lemma *rec-nat zero suc 0* = *zero*
refute [*expect* = *none*]
⟨*proof*⟩

lemma *rec-nat zero suc* (*Suc x*) = *suc x* (*rec-nat zero suc x*)
refute [*maxsize* = *2*, *expect* = *none*]
⟨*proof*⟩

lemma P (*rec-nat zero suc x*)
refute [*expect = potential*]
⟨*proof*⟩

lemma P (*case x of 0 ⇒ zero | Suc n ⇒ suc n*)
refute [*expect = potential*]
⟨*proof*⟩

'a list

lemma P (*xs::'a list*)
refute [*expect = potential*]
⟨*proof*⟩

lemma $\forall$ *xs::'a list. P xs*
refute [*expect = potential*]
⟨*proof*⟩

lemma P [*x, y*]
refute [*expect = potential*]
⟨*proof*⟩

lemma *rec-list nil cons [] = nil*
refute [*maxsize = 3, expect = none*]
⟨*proof*⟩

lemma *rec-list nil cons (x#xs) = cons x xs (rec-list nil cons xs)*
refute [*maxsize = 2, expect = none*]
⟨*proof*⟩

lemma P (*rec-list nil cons xs*)
refute [*expect = potential*]
⟨*proof*⟩

lemma P (*case x of Nil ⇒ nil | Cons a b ⇒ cons a b*)
refute [*expect = potential*]
⟨*proof*⟩

lemma (*xs::'a list*) = *ys*
refute [*expect = potential*]
⟨*proof*⟩

lemma *a # xs = b # xs*
refute [*expect = potential*]
⟨*proof*⟩

datatype *BitList = BitListNil | Bit0 BitList | Bit1 BitList*

lemma P (*x::BitList*)
refute [*expect = potential*]

⟨*proof*⟩

lemma ∀ *x*::*BitList*. *P x*
refute [*expect* = *potential*]
⟨*proof*⟩

lemma *P* (*Bit0* (*Bit1 BitListNil*))
refute [*expect* = *potential*]
⟨*proof*⟩

lemma *rec-BitList nil bit0 bit1 BitListNil* = *nil*
refute [*maxsize* = *4*, *expect* = *none*]
⟨*proof*⟩

lemma *rec-BitList nil bit0 bit1* (*Bit0 xs*) = *bit0 xs* (*rec-BitList nil bit0 bit1 xs*)
refute [*maxsize* = *2*, *expect* = *none*]
⟨*proof*⟩

lemma *rec-BitList nil bit0 bit1* (*Bit1 xs*) = *bit1 xs* (*rec-BitList nil bit0 bit1 xs*)
refute [*maxsize* = *2*, *expect* = *none*]
⟨*proof*⟩

lemma *P* (*rec-BitList nil bit0 bit1 x*)
refute [*expect* = *potential*]
⟨*proof*⟩

56.1.15 Examples involving special functions

lemma *card x* = *0*
refute [*expect* = *potential*]
⟨*proof*⟩

lemma *finite x*
refute — no finite countermodel exists
⟨*proof*⟩

lemma (*x*::*nat*) + *y* = *0*
refute [*expect* = *potential*]
⟨*proof*⟩

lemma (*x*::*nat*) = *x* + *x*
refute [*expect* = *potential*]
⟨*proof*⟩

lemma (*x*::*nat*) − *y* + *y* = *x*
refute [*expect* = *potential*]
⟨*proof*⟩

lemma (*x*::*nat*) = *x* ∗ *x*

refute [*expect* = *potential*]
⟨*proof*⟩

lemma (*x*::*nat*) < *x* + *y*
refute [*expect* = *potential*]
⟨*proof*⟩

lemma *xs* @ [] = *ys* @ []
refute [*expect* = *potential*]
⟨*proof*⟩

lemma *xs* @ *ys* = *ys* @ *xs*
refute [*expect* = *potential*]
⟨*proof*⟩

56.1.16 Type classes and overloading

A type class without axioms:

class *classA*

lemma *P* (*x*::'*a*::*classA*)
refute [*expect* = *genuine*]
⟨*proof*⟩

An axiom with a type variable (denoting types which have at least two elements):

class *classC* =
  assumes *classC-ax*: ∃ *x* *y*. *x* ≠ *y*

lemma *P* (*x*::'*a*::*classC*)
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma ∃ *x* *y*. (*x*::'*a*::*classC*) ≠ *y*

⟨*proof*⟩

A type class for which a constant is defined:

class *classD* =
  fixes *classD-const* :: '*a* ⇒ '*a*
  assumes *classD-ax*: *classD-const* (*classD-const x*) = *classD-const x*

lemma *P* (*x*::'*a*::*classD*)
refute [*expect* = *genuine*]
⟨*proof*⟩

A type class with multiple superclasses:

class *classE* = *classC* + *classD*

lemma P (x::$'a$::*classE*)
refute [*expect* = *genuine*]
⟨*proof*⟩

OFCLASS:

lemma *OFCLASS*($'a$::*type*, *type-class*)
refute [*expect* = *none*]
⟨*proof*⟩

lemma *OFCLASS*($'a$::*classC*, *type-class*)
refute [*expect* = *none*]
⟨*proof*⟩

lemma *OFCLASS*($'a$::*type*, *classC-class*)
refute [*expect* = *genuine*]
⟨*proof*⟩

Overloading:

consts *inverse* :: $'a \Rightarrow 'a$

overloading *inverse-bool* ≡ *inverse* :: *bool* ⇒ *bool*
begin
  definition *inverse* (*b*::*bool*) ≡ ¬ *b*
end

overloading *inverse-set* ≡ *inverse* :: $'a$ *set* ⇒ $'a$ *set*
begin
  definition *inverse* (S::$'a$ *set*) ≡ $-S$
end

overloading *inverse-pair* ≡ *inverse* :: $'a \times 'b \Rightarrow 'a \times 'b$
begin
  definition *inverse-pair p* ≡ (*inverse* (*fst p*), *inverse* (*snd p*))
end

lemma *inverse b*
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma P (*inverse* (S::$'a$ *set*))
refute [*expect* = *genuine*]
⟨*proof*⟩

lemma P (*inverse* (p::$'a \times 'b$))
refute [*expect* = *genuine*]
⟨*proof*⟩

Structured proofs

lemma $x = y$
⟨*proof*⟩

refute-params [*satsolver* = *auto*]

end

57 Implementation of Association Lists

theory *AList*
imports *Main*
begin

context
begin

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

57.1 *update* and *updates*

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

lemma *update-conv'*: *map-of* (*update k v al*) = (*map-of al*)(*k*↦*v*)
⟨*proof*⟩

corollary *update-conv*: *map-of* (*update k v al*) *k'* = ((*map-of al*)(*k*↦*v*)) *k'*
⟨*proof*⟩

lemma *dom-update*: *fst* ‘ *set* (*update k v al*) = {*k*} ∪ *fst* ‘ *set al*
⟨*proof*⟩

lemma *update-keys*:
map fst (*update k v al*) =
(*if k* ∈ *set* (*map fst al*) *then map fst al else map fst al* @ [*k*])
⟨*proof*⟩

lemma *distinct-update*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*update k v al*))
⟨*proof*⟩

lemma *update-filter*:
a ≠ *k* ⟹ *update k v* [*q*←*ps*. *fst q* ≠ *a*] = [*q*←*update k v ps*. *fst q* ≠ *a*]

⟨*proof*⟩

lemma *update-triv*: *map-of al k* = *Some v* ⟹ *update k v al* = *al*
⟨*proof*⟩

lemma *update-nonempty* [*simp*]: *update k v al* ≠ []
⟨*proof*⟩

lemma *update-eqD*: *update k v al* = *update k v' al'* ⟹ *v* = *v'*
⟨*proof*⟩

lemma *update-last* [*simp*]: *update k v* (*update k v' al*) = *update k v al*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *image-update* [*simp*]: *x* ∉ *A* ⟹ *map-of* (*update x y al*) ‘ *A* = *map-of al* ‘ *A*
⟨*proof*⟩ **definition** *updates* ::
'key list ⇒ *'val list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *updates ks vs* = *fold* (*case-prod update*) (*zip ks vs*)

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

lemma *updates-key-simp* [*simp*]:
updates (*k* # *ks*) *vs ps* =
(*case vs of* [] ⇒ *ps* | *v* # *vs* ⇒ *updates ks vs* (*update k v ps*))
⟨*proof*⟩

lemma *updates-conv'*: *map-of* (*updates ks vs al*) = (*map-of al*)(*ks*[↦]*vs*)
⟨*proof*⟩

lemma *updates-conv*: *map-of* (*updates ks vs al*) *k* = ((*map-of al*)(*ks*[↦]*vs*)) *k*
⟨*proof*⟩

lemma *distinct-updates*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*updates ks vs al*))
⟨*proof*⟩

lemma *updates-append1*[*simp*]: *size ks* < *size vs* ⟹
updates (*ks*@[*k*]) *vs al* = *update k* (*vs*!*size ks*) (*updates ks vs al*)
⟨*proof*⟩

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

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

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*))
⟨*proof*⟩

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

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

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

57.2 *delete*

qualified definition *delete* :: *'key* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *delete-eq*: *delete k* = *filter* (λ(*k'*, -). *k* ≠ *k'*)

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

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

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

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

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

lemma *delete-id* [*simp*]: *k* ∉ *fst* ‘ *set al* ⟹ *delete k al* = *al*
⟨*proof*⟩

lemma *delete-idem*: *delete k* (*delete k al*) = *delete k al*
⟨*proof*⟩

lemma *map-of-delete* [*simp*]: *k'* ≠ *k* ⟹ *map-of* (*delete k al*) *k'* = *map-of al k'*
⟨*proof*⟩

lemma *delete-notin-dom*: *k* ∉ *fst* ‘ *set* (*delete k al*)
⟨*proof*⟩

lemma *dom-delete-subset*: *fst* ‘ *set* (*delete k al*) ⊆ *fst* ‘ *set al*
⟨*proof*⟩

lemma *delete-update-same*: *delete k* (*update k v al*) = *delete k al*
⟨*proof*⟩

lemma *delete-update*: *k* ≠ *l* ⟹ *delete l* (*update k v al*) = *update k v* (*delete l al*)
⟨*proof*⟩

lemma *delete-twist*: *delete x* (*delete y al*) = *delete y* (*delete x al*)
⟨*proof*⟩

lemma *length-delete-le*: *length* (*delete k al*) ≤ *length al*
⟨*proof*⟩

57.3 *update-with-aux* and *delete-aux*

qualified primrec *update-with-aux* ::
'val ⇒ *'key* ⇒ (*'val* ⇒ *'val*) ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where
update-with-aux v k f [] = [(*k*, *f v*)]
| *update-with-aux v k f* (*p* # *ps*) =
(*if* (*fst p* = *k*) *then* (*k*, *f* (*snd p*)) # *ps else p* # *update-with-aux v k f ps*)

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

qualified fun *delete-aux* :: *'key* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where
delete-aux k [] = []
| *delete-aux k* ((*k'*, *v*) # *xs*) = (*if k* = *k' then xs else* (*k'*, *v*) # *delete-aux k xs*)

lemma *map-of-update-with-aux'*:
map-of (*update-with-aux v k f ps*) *k'* =
((*map-of ps*)(*k* ↦ (*case map-of ps k of None* ⇒ *f v* | *Some v* ⇒ *f v*))) *k'*
⟨*proof*⟩

lemma *map-of-update-with-aux*:
map-of (*update-with-aux v k f ps*) =
(*map-of ps*)(*k* ↦ (*case map-of ps k of None* ⇒ *f v* | *Some v* ⇒ *f v*))
⟨*proof*⟩

lemma *dom-update-with-aux*: *fst* ‘ *set* (*update-with-aux v k f ps*) = {*k*} ∪ *fst* ‘ *set ps*
⟨*proof*⟩

lemma *distinct-update-with-aux* [*simp*]:
distinct (*map fst* (*update-with-aux v k f ps*)) = *distinct* (*map fst ps*)
⟨*proof*⟩

lemma *set-update-with-aux*:
distinct (*map fst xs*) ⟹
set (*update-with-aux v k f xs*) =
(*set xs* − {*k*} × *UNIV* ∪ {(*k*, *f* (*case map-of xs k of None* ⇒ *v* | *Some v* ⇒ *v*))})
⟨*proof*⟩

lemma *set-delete-aux*: *distinct* (*map fst xs*) ⟹ *set* (*delete-aux k xs*) = *set xs* − {*k*} × *UNIV*
⟨*proof*⟩

lemma *dom-delete-aux*: *distinct* (*map fst ps*) ⟹ *fst* ‘ *set* (*delete-aux k ps*) = *fst* ‘ *set ps* − {*k*}
⟨*proof*⟩

lemma *distinct-delete-aux* [*simp*]: *distinct* (*map fst ps*) ⟹ *distinct* (*map fst* (*delete-aux k ps*))
⟨*proof*⟩

lemma *map-of-delete-aux'*:
distinct (*map fst xs*) ⟹ *map-of* (*delete-aux k xs*) = (*map-of xs*)(*k* := *None*)
⟨*proof*⟩

lemma *map-of-delete-aux*:
distinct (*map fst xs*) $\Longrightarrow$ *map-of* (*delete-aux k xs*) *k'* = ((*map-of xs*)(*k* := *None*)) *k'*
⟨*proof*⟩

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

57.4 *restrict*

qualified definition *restrict* :: *'key set* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *restrict-eq*: *restrict A* = *filter* (λ(*k*, *v*). *k* ∈ *A*)

lemma *restr-simps* [*simp*]:
restrict A [] = []
restrict A (*p*#*ps*) = (*if fst p* ∈ *A then p* # *restrict A ps else restrict A ps*)
⟨*proof*⟩

lemma *restr-conv'*: *map-of* (*restrict A al*) = ((*map-of al*)|‘ *A*)
⟨*proof*⟩

corollary *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k*
⟨*proof*⟩

lemma *distinct-restr*: *distinct* (*map fst al*) $\Longrightarrow$ *distinct* (*map fst* (*restrict A al*))
⟨*proof*⟩

lemma *restr-empty* [*simp*]:
restrict {} *al* = []
restrict A [] = []
⟨*proof*⟩

lemma *restr-in* [*simp*]: *x* ∈ *A* $\Longrightarrow$ *map-of* (*restrict A al*) *x* = *map-of al x*
⟨*proof*⟩

lemma *restr-out* [*simp*]: *x* ∉ *A* $\Longrightarrow$ *map-of* (*restrict A al*) *x* = *None*
⟨*proof*⟩

lemma *dom-restr* [*simp*]: *fst* ‘ *set* (*restrict A al*) = *fst* ‘ *set al* ∩ *A*
⟨*proof*⟩

lemma *restr-upd-same* [*simp*]: *restrict* (−{*x*}) (*update x y al*) = *restrict* (−{*x*}) *al*
⟨*proof*⟩

lemma *restr-restr* [*simp*]: *restrict A* (*restrict B al*) = *restrict* (*A*∩*B*) *al*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *restr-delete* [*simp*]:
delete x (*restrict D al*) = (*if x* ∈ *D then restrict* (*D* − {*x*}) *al else restrict D al*)
⟨*proof*⟩

lemma *update-restr*:
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D* − {*x*}) *al*))
⟨*proof*⟩

lemma *update-restr-conv* [*simp*]:
x ∈ *D* ⟹
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D* − {*x*}) *al*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *restr-delete-twist*: (*restrict A* (*delete a ps*)) = *delete a* (*restrict A ps*)
⟨*proof*⟩

57.5 *clearjunk*

qualified function *clearjunk* :: ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list*
where
clearjunk [] = []
| *clearjunk* (*p*#*ps*) = *p* # *clearjunk* (*delete* (*fst p*) *ps*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *map-of-clearjunk*: *map-of* (*clearjunk al*) = *map-of al*
⟨*proof*⟩

lemma *clearjunk-keys-set*: *set* (*map fst* (*clearjunk al*)) = *set* (*map fst al*)
⟨*proof*⟩

lemma *dom-clearjunk*: *fst* ‘ *set* (*clearjunk al*) = *fst* ‘ *set al*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *ran-clearjunk*: *ran* (*map-of* (*clearjunk al*)) = *ran* (*map-of al*)
⟨*proof*⟩

lemma *ran-map-of*: *ran* (*map-of al*) = *snd* ‘ *set* (*clearjunk al*)
⟨*proof*⟩

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

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

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

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

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

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

lemma *length-clearjunk*: *length* (*clearjunk al*) ≤ *length al*
⟨*proof*⟩

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

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

57.6 *map-ran*

definition *map-ran* :: (*'key* ⇒ *'val* ⇒ *'val*) ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *map-ran f* = *map* (λ(*k*, *v*). (*k*, *f k v*))

lemma *map-ran-simps* [*simp*]:
map-ran f [] = []
map-ran f ((*k*, *v*) # *ps*) = (*k*, *f k v*) # *map-ran f ps*

⟨*proof*⟩

lemma *dom-map-ran*: *fst ‘ set* (*map-ran f al*) = *fst ‘ set al*
⟨*proof*⟩

lemma *map-ran-conv*: *map-of* (*map-ran f al*) *k* = *map-option* (*f k*) (*map-of al k*)
⟨*proof*⟩

lemma *distinct-map-ran*: *distinct* (*map fst al*) ⟹ *distinct* (*map fst* (*map-ran f al*))
⟨*proof*⟩

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

lemma *clearjunk-map-ran*: *clearjunk* (*map-ran f al*) = *map-ran f* (*clearjunk al*)
⟨*proof*⟩

57.7 *merge*

qualified definition *merge* :: (′*key* × ′*val*) *list* ⇒ (′*key* × ′*val*) *list* ⇒ (′*key* × ′*val*) *list*
where *merge qs ps* = *foldr* (λ(*k*, *v*). *update k v*) *ps qs*

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

lemma *merge-updates*: *merge qs ps* = *updates* (*rev* (*map fst ps*)) (*rev* (*map snd ps*)) *qs*
⟨*proof*⟩

lemma *dom-merge*: *fst ‘ set* (*merge xs ys*) = *fst ‘ set xs* ∪ *fst ‘ set ys*
⟨*proof*⟩

lemma *distinct-merge*: *distinct* (*map fst xs*) ⟹ *distinct* (*map fst* (*merge xs ys*))
⟨*proof*⟩

lemma *clearjunk-merge*: *clearjunk* (*merge xs ys*) = *merge* (*clearjunk xs*) *ys*
⟨*proof*⟩

lemma *merge-conv′*: *map-of* (*merge xs ys*) = *map-of xs* ++ *map-of ys*
⟨*proof*⟩

corollary *merge-conv*: *map-of* (*merge xs ys*) *k* = (*map-of xs* ++ *map-of ys*) *k*
⟨*proof*⟩

lemma *merge-empty*: *map-of* (*merge* [] *ys*) = *map-of ys*
⟨*proof*⟩

lemma *merge-assoc* [*simp*]: *map-of* (*merge m1* (*merge m2 m3*)) = *map-of* (*merge* (*merge m1 m2*) *m3*)
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *merge-find-right* [*simp*]: *map-of n k* = *Some v* ⟹ *map-of* (*merge m n*) *k* = *Some v*
⟨*proof*⟩

lemma *merge-None* [*iff*]: (*map-of* (*merge m n*) *k* = *None*) = (*map-of n k* = *None* ∧ *map-of m k* = *None*)
⟨*proof*⟩

lemma *merge-upd* [*simp*]: *map-of* (*merge m* (*update k v n*)) = *map-of* (*update k v* (*merge m n*))
⟨*proof*⟩

lemma *merge-updatess* [*simp*]:
map-of (*merge m* (*updates xs ys n*)) = *map-of* (*updates xs ys* (*merge m n*))
⟨*proof*⟩

lemma *merge-append*: *map-of* (*xs* @ *ys*) = *map-of* (*merge ys xs*)
⟨*proof*⟩

57.8 *compose*

qualified function *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*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *compose-first-None* [*simp*]: *map-of xs k* = *None* ⟹ *map-of* (*compose xs ys*) *k* = *None*
⟨*proof*⟩

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

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

lemma *compose-first-Some* [*simp*]: *map-of xs k* = *Some v* ⟹ *map-of* (*compose xs ys*) *k* = *map-of ys v*
⟨*proof*⟩

lemma *dom-compose*: *fst* ‘ *set* (*compose xs ys*) ⊆ *fst* ‘ *set xs*
⟨*proof*⟩

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

lemma *compose-delete-twist*: *compose* (*delete k xs*) *ys* = *delete k* (*compose xs ys*)
⟨*proof*⟩

lemma *compose-clearjunk*: *compose xs* (*clearjunk ys*) = *compose xs ys*
⟨*proof*⟩

lemma *clearjunk-compose*: *clearjunk* (*compose xs ys*) = *compose* (*clearjunk xs*) *ys*
⟨*proof*⟩

lemma *compose-empty* [*simp*]: *compose xs* [] = []
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

57.9 *map-entry*

qualified fun *map-entry* :: *'key* ⇒ (*'val* ⇒ *'val*) ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where
map-entry k f [] = []
| *map-entry k f* (*p* # *ps*) =
(*if fst p* = *k then* (*k*, *f* (*snd p*)) # *ps else p* # *map-entry k f ps*)

lemma *map-of-map-entry*:
map-of (*map-entry k f xs*) =
(*map-of xs*)(*k* := *case map-of xs k of None* ⇒ *None* | *Some v′* ⇒ *Some* (*f v′*))
⟨*proof*⟩

lemma *dom-map-entry*: *fst* ‘ *set* (*map-entry k f xs*) = *fst* ‘ *set xs*
⟨*proof*⟩

lemma *distinct-map-entry*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*map-entry k f xs*))
⟨*proof*⟩

57.10 *map-default*

fun *map-default* :: *′key* ⇒ *′val* ⇒ (*′val* ⇒ *′val*) ⇒ (*′key* × *′val*) *list* ⇒ (*′key* × *′val*) *list*
where
map-default k v f [] = [(*k*, *v*)]
| *map-default k v f* (*p* # *ps*) =
(*if fst p* = *k then* (*k*, *f* (*snd p*)) # *ps else p* # *map-default k v f ps*)

lemma *map-of-map-default*:
map-of (*map-default k v f xs*) =
(*map-of xs*)(*k* := *case map-of xs k of None* ⇒ *Some v* | *Some v′* ⇒ *Some* (*f v′*))
⟨*proof*⟩

lemma *dom-map-default*: *fst* ‘ *set* (*map-default k v f xs*) = *insert k* (*fst* ‘ *set xs*)
⟨*proof*⟩

lemma *distinct-map-default*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*map-default k v f xs*))
⟨*proof*⟩

end

end

58 An abstract view on maps for code generation.

theory *Mapping*
imports *Main*
begin

58.1 Parametricity transfer rules

lemma *map-of-foldr*: *map-of xs* = *foldr* (λ(*k*, *v*) *m*. *m*(*k* ↦ *v*)) *xs Map.empty*
⟨*proof*⟩

context includes *lifting-syntax*
begin

lemma *empty-parametric*: (*A* ===> *rel-option B*) *Map.empty Map.empty*
⟨*proof*⟩

lemma *lookup-parametric*: ((*A* ===> *B*) ===> *A* ===> *B*) (λ*m k*. *m k*) (λ*m k*. *m k*)
⟨*proof*⟩

lemma *update-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *B* ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
(λ*k v m*. *m*(*k* ↦ *v*)) (λ*k v m*. *m*(*k* ↦ *v*))
⟨*proof*⟩

lemma *delete-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
(λ*k m*. *m*(*k* := *None*)) (λ*k m*. *m*(*k* := *None*))
⟨*proof*⟩

lemma *is-none-parametric* [*transfer-rule*]:
(*rel-option A* ===> *HOL.eq*) *Option.is-none Option.is-none*
⟨*proof*⟩

lemma *dom-parametric*:
assumes [*transfer-rule*]: *bi-total A*
shows ((*A* ===> *rel-option B*) ===> *rel-set A*) *dom dom*
⟨*proof*⟩

lemma *map-of-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique R1*
shows (*list-all2* (*rel-prod R1 R2*) ===> *R1* ===> *rel-option R2*) *map-of map-of*
⟨*proof*⟩

lemma *map-entry-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> (*B* ===> *B*) ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
(λ*k f m*. (*case m k of None* ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*)))) (λ*k f m*. (*case m k of None* ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*))))

⟨*proof*⟩

lemma *tabulate-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> (*A* ===> *B*) ===> *A* ===> *rel-option B*)
(λ*ks f*. (*map-of* (*map* (λ*k*. (*k*, *f k*)) *ks*))) (λ*ks f*. (*map-of* (*map* (λ*k*. (*k*, *f k*)) *ks*)))
⟨*proof*⟩

lemma *bulkload-parametric*:
(*list-all2 A* ===> *HOL.eq* ===> *rel-option A*)
(λ*xs k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None*)
(λ*xs k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None*)
⟨*proof*⟩

lemma *map-parametric*:
((*A* ===> *B*) ===> (*C* ===> *D*) ===> (*B* ===> *rel-option C*) ===> *A* ===> *rel-option D*)
(λ*f g m*. (*map-option g* ∘ *m* ∘ *f*)) (λ*f g m*. (*map-option g* ∘ *m* ∘ *f*))
⟨*proof*⟩

lemma *combine-with-key-parametric*:
((*A* ===> *B* ===> *B* ===> *B*) ===> (*A* ===> *rel-option B*) ===> (*A* ===> *rel-option B*) ===>
(*A* ===> *rel-option B*)) (λ*f m1 m2 x*. *combine-options* (*f x*) (*m1 x*) (*m2 x*))
(λ*f m1 m2 x*. *combine-options* (*f x*) (*m1 x*) (*m2 x*))
⟨*proof*⟩

lemma *combine-parametric*:
((*B* ===> *B* ===> *B*) ===> (*A* ===> *rel-option B*) ===> (*A* ===> *rel-option B*) ===>
(*A* ===> *rel-option B*)) (λ*f m1 m2 x*. *combine-options f* (*m1 x*) (*m2 x*))
(λ*f m1 m2 x*. *combine-options f* (*m1 x*) (*m2 x*))
⟨*proof*⟩

end

58.2 Type definition and primitive operations

typedef ('*a*, '*b*) *mapping* = *UNIV* :: ('*a* ⇀ '*b*) *set*
morphisms *rep Mapping* ⟨*proof*⟩

setup-lifting *type-definition-mapping*

lift-definition *empty* :: ('*a*, '*b*) *mapping*
is *Map.empty* **parametric** *empty-parametric* ⟨*proof*⟩

lift-definition *lookup* :: ('*a*, '*b*) *mapping* ⇒ '*a* ⇒ '*b option*
is λ*m k*. *m k* **parametric** *lookup-parametric* ⟨*proof*⟩

definition *lookup-default d m k* = (*case Mapping.lookup m k of None* ⇒ *d* | *Some v* ⇒ *v*)

declare [[*code drop*: *Mapping.lookup*]]
⟨*ML*⟩

lift-definition *update* :: *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping*
is λ*k v m*. *m*(*k* ↦ *v*) **parametric** *update-parametric* ⟨*proof*⟩

lift-definition *delete* :: *'a* ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping*
is λ*k m*. *m*(*k* := *None*) **parametric** *delete-parametric* ⟨*proof*⟩

lift-definition *filter* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping*
is λ*P m k*. *case m k of None* ⇒ *None* | *Some v* ⇒ *if P k v then Some v else None* ⟨*proof*⟩

lift-definition *keys* :: (*'a*, *'b*) *mapping* ⇒ *'a set*
is *dom* **parametric** *dom-parametric* ⟨*proof*⟩

lift-definition *tabulate* :: *'a list* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a*, *'b*) *mapping*
is λ*ks f*. (*map-of* (*List.map* (λ*k*. (*k*, *f k*)) *ks*)) **parametric** *tabulate-parametric* ⟨*proof*⟩

lift-definition *bulkload* :: *'a list* ⇒ (*nat*, *'a*) *mapping*
is λ*xs k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None* **parametric** *bulkload-parametric* ⟨*proof*⟩

lift-definition *map* :: (*'c* ⇒ *'a*) ⇒ (*'b* ⇒ *'d*) ⇒ (*'a*, *'b*) *mapping* ⇒ (*'c*, *'d*) *mapping*
is λ*f g m*. (*map-option g* ∘ *m* ∘ *f*) **parametric** *map-parametric* ⟨*proof*⟩

lift-definition *map-values* :: (*'c* ⇒ *'a* ⇒ *'b*) ⇒ (*'c*, *'a*) *mapping* ⇒ (*'c*, *'b*) *mapping*
is λ*f m x*. *map-option* (*f x*) (*m x*) ⟨*proof*⟩

lift-definition *combine-with-key* ::
(*'a* ⇒ *'b* ⇒ *'b* ⇒ *'b*) ⇒ (*'a*,*'b*) *mapping* ⇒ (*'a*,*'b*) *mapping* ⇒ (*'a*,*'b*) *mapping*
is λ*f m1 m2 x*. *combine-options* (*f x*) (*m1 x*) (*m2 x*) **parametric** *combine-with-key-parametric* ⟨*proof*⟩

lift-definition *combine* ::
(*'b* ⇒ *'b* ⇒ *'b*) ⇒ (*'a*,*'b*) *mapping* ⇒ (*'a*,*'b*) *mapping* ⇒ (*'a*,*'b*) *mapping*
is λ*f m1 m2 x*. *combine-options f* (*m1 x*) (*m2 x*) **parametric** *combine-parametric* ⟨*proof*⟩

definition *All-mapping m P* ⟷
(∀ *x*. *case Mapping.lookup m x of None* ⇒ *True* | *Some y* ⇒ *P x y*)

declare [[*code drop*: *map*]]

58.3 Functorial structure

functor *map*: *map*
⟨*proof*⟩

58.4 Derived operations

definition *ordered-keys* :: ('*a*::*linorder*, '*b*) *mapping* ⇒ '*a list*
where *ordered-keys m* = (*if finite* (*keys m*) *then sorted-list-of-set* (*keys m*) *else* [])

definition *is-empty* :: ('*a*, '*b*) *mapping* ⇒ *bool*
where *is-empty m* ⟷ *keys m* = {}

definition *size* :: ('*a*, '*b*) *mapping* ⇒ *nat*
where *size m* = (*if finite* (*keys m*) *then card* (*keys m*) *else 0*)

definition *replace* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
where *replace k v m* = (*if k* ∈ *keys m then update k v m else m*)

definition *default* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
where *default k v m* = (*if k* ∈ *keys m then m else update k v m*)

Manual derivation of transfer rule is non-trivial

lift-definition *map-entry* :: '*a* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
is
λk f m.
(*case m k of*
None ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*))) **parametric** *map-entry-parametric* ⟨*proof*⟩

lemma *map-entry-code* [*code*]:
map-entry k f m =
(*case lookup m k of*
None ⇒ *m*
| *Some v* ⇒ *update k* (*f v*) *m*)
⟨*proof*⟩

definition *map-default* :: '*a* ⇒ '*b* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
where *map-default k v f m* = *map-entry k f* (*default k v m*)

definition *of-alist* :: ('*k* × '*v*) *list* ⇒ ('*k*, '*v*) *mapping*
where *of-alist xs* = *foldr* (*λ*(*k*, *v*) *m*. *update k v m*) *xs empty*

instantiation *mapping* :: (*type*, *type*) *equal*
begin

definition *HOL.equal m1 m2* ⟷ (∀ *k*. *lookup m1 k* = *lookup m2 k*)

instance
⟨*proof*⟩

end

context includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
and [*transfer-rule*]: *bi-unique B*
shows (*pcr-mapping A B* ===> *pcr-mapping A B* ===> *op=*) *HOL.eq HOL.equal*
⟨*proof*⟩

lemma *of-alist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique R1*
shows (*list-all2* (*rel-prod R1 R2*) ===> *pcr-mapping R1 R2*) *map-of of-alist*
⟨*proof*⟩

end

58.5 Properties

lemma *mapping-eqI*: $(\bigwedge x.\ lookup\ m\ x = lookup\ m'\ x) \Longrightarrow m = m'$
⟨*proof*⟩

lemma *mapping-eqI'*:
assumes $\bigwedge x.\ x \in Mapping.keys\ m \Longrightarrow Mapping.lookup\text{-}default\ d\ m\ x = Mapping.lookup\text{-}default\ d\ m'\ x$
and $Mapping.keys\ m = Mapping.keys\ m'$
shows $m = m'$
⟨*proof*⟩

lemma *lookup-update*: $lookup\ (update\ k\ v\ m)\ k = Some\ v$
⟨*proof*⟩

lemma *lookup-update-neq*: $k \neq k' \Longrightarrow lookup\ (update\ k\ v\ m)\ k' = lookup\ m\ k'$
⟨*proof*⟩

lemma *lookup-update'*: $Mapping.lookup\ (update\ k\ v\ m)\ k' = (if\ k = k'\ then\ Some\ v\ else\ lookup\ m\ k')$
⟨*proof*⟩

lemma *lookup-empty*: $lookup\ empty\ k = None$
⟨*proof*⟩

lemma *lookup-filter*:
$lookup\ (filter\ P\ m)\ k =$
(*case lookup m k of*

None ⇒ *None*
| *Some v* ⇒ *if P k v then Some v else None*)
⟨*proof*⟩

lemma *lookup-map-values*: *lookup* (*map-values f m*) *k* = *map-option* (*f k*) (*lookup m k*)
⟨*proof*⟩

lemma *lookup-default-empty*: *lookup-default d empty k* = *d*
⟨*proof*⟩

lemma *lookup-default-update*: *lookup-default d* (*update k v m*) *k* = *v*
⟨*proof*⟩

lemma *lookup-default-update-neq*:
k ≠ *k'* ⟹ *lookup-default d* (*update k v m*) *k'* = *lookup-default d m k'*
⟨*proof*⟩

lemma *lookup-default-update'*:
lookup-default d (*update k v m*) *k'* = (*if k* = *k' then v else lookup-default d m k'*)
⟨*proof*⟩

lemma *lookup-default-filter*:
lookup-default d (*filter P m*) *k* =
(*if P k* (*lookup-default d m k*) *then lookup-default d m k else d*)
⟨*proof*⟩

lemma *lookup-default-map-values*:
lookup-default (*f k d*) (*map-values f m*) *k* = *f k* (*lookup-default d m k*)
⟨*proof*⟩

lemma *lookup-combine-with-key*:
Mapping.lookup (*combine-with-key f m1 m2*) *x* =
combine-options (*f x*) (*Mapping.lookup m1 x*) (*Mapping.lookup m2 x*)
⟨*proof*⟩

lemma *combine-altdef*: *combine f m1 m2* = *combine-with-key* (λ-. *f*) *m1 m2*
⟨*proof*⟩

lemma *lookup-combine*:
Mapping.lookup (*combine f m1 m2*) *x* =
combine-options f (*Mapping.lookup m1 x*) (*Mapping.lookup m2 x*)
⟨*proof*⟩

lemma *lookup-default-neutral-combine-with-key*:
assumes ⋀*x*. *f k d x* = *x* ⋀*x*. *f k x d* = *x*
shows *Mapping.lookup-default d* (*combine-with-key f m1 m2*) *k* =
f k (*Mapping.lookup-default d m1 k*) (*Mapping.lookup-default d m2 k*)
⟨*proof*⟩

lemma *lookup-default-neutral-combine*:
assumes $\bigwedge x.\ f\ d\ x = x\ \bigwedge x.\ f\ x\ d = x$
shows *Mapping.lookup-default d* (*combine f m1 m2*) *x* =
f (*Mapping.lookup-default d m1 x*) (*Mapping.lookup-default d m2 x*)
⟨*proof*⟩

lemma *lookup-map-entry*: *lookup* (*map-entry x f m*) *x* = *map-option f* (*lookup m x*)
⟨*proof*⟩

lemma *lookup-map-entry-neq*: $x \neq y \Longrightarrow$ *lookup* (*map-entry x f m*) *y* = *lookup m y*
⟨*proof*⟩

lemma *lookup-map-entry'*:
lookup (*map-entry x f m*) *y* =
(*if x* = *y then map-option f* (*lookup m y*) *else lookup m y*)
⟨*proof*⟩

lemma *lookup-default*: *lookup* (*default x d m*) *x* = *Some* (*lookup-default d m x*)
⟨*proof*⟩

lemma *lookup-default-neq*: $x \neq y \Longrightarrow$ *lookup* (*default x d m*) *y* = *lookup m y*
⟨*proof*⟩

lemma *lookup-default'*:
lookup (*default x d m*) *y* =
(*if x* = *y then Some* (*lookup-default d m x*) *else lookup m y*)
⟨*proof*⟩

lemma *lookup-map-default*: *lookup* (*map-default x d f m*) *x* = *Some* (*f* (*lookup-default d m x*))
⟨*proof*⟩

lemma *lookup-map-default-neq*: $x \neq y \Longrightarrow$ *lookup* (*map-default x d f m*) *y* = *lookup m y*
⟨*proof*⟩

lemma *lookup-map-default'*:
lookup (*map-default x d f m*) *y* =
(*if x* = *y then Some* (*f* (*lookup-default d m x*)) *else lookup m y*)
⟨*proof*⟩

lemma *lookup-tabulate*:
assumes *distinct xs*
shows *Mapping.lookup* (*Mapping.tabulate xs f*) *x* = (*if x* ∈ *set xs then Some* (*f x*) *else None*)
⟨*proof*⟩

lemma *lookup-of-alist*: *Mapping.lookup* (*Mapping.of-alist xs*) *k* = *map-of xs k*
⟨*proof*⟩

lemma *keys-is-none-rep* [*code-unfold*]: *k* ∈ *keys m* ⟷ ¬ (*Option.is-none* (*lookup m k*))
⟨*proof*⟩

lemma *update-update*:
update k v (*update k w m*) = *update k v m*
k ≠ *l* ⟹ *update k v* (*update l w m*) = *update l w* (*update k v m*)
⟨*proof*⟩

lemma *update-delete* [*simp*]: *update k v* (*delete k m*) = *update k v m*
⟨*proof*⟩

lemma *delete-update*:
delete k (*update k v m*) = *delete k m*
k ≠ *l* ⟹ *delete k* (*update l v m*) = *update l v* (*delete k m*)
⟨*proof*⟩

lemma *delete-empty* [*simp*]: *delete k empty* = *empty*
⟨*proof*⟩

lemma *replace-update*:
k ∉ *keys m* ⟹ *replace k v m* = *m*
k ∈ *keys m* ⟹ *replace k v m* = *update k v m*
⟨*proof*⟩

lemma *map-values-update*: *map-values f* (*update k v m*) = *update k* (*f k v*) (*map-values f m*)
⟨*proof*⟩

lemma *size-mono*: *finite* (*keys m'*) ⟹ *keys m* ⊆ *keys m'* ⟹ *size m* ≤ *size m'*
⟨*proof*⟩

lemma *size-empty* [*simp*]: *size empty* = *0*
⟨*proof*⟩

lemma *size-update*:
finite (*keys m*) ⟹ *size* (*update k v m*) =
(*if k* ∈ *keys m then size m else Suc* (*size m*))
⟨*proof*⟩

lemma *size-delete*: *size* (*delete k m*) = (*if k* ∈ *keys m then size m* − *1 else size m*)
⟨*proof*⟩

lemma *size-tabulate* [*simp*]: *size* (*tabulate ks f*) = *length* (*remdups ks*)

⟨*proof*⟩

lemma *keys-filter*: *keys* (*filter* *P* *m*) ⊆ *keys* *m*
⟨*proof*⟩

lemma *size-filter*: *finite* (*keys* *m*) ⟹ *size* (*filter* *P* *m*) ≤ *size* *m*
⟨*proof*⟩

lemma *bulkload-tabulate*: *bulkload* *xs* = *tabulate* [*0*..<*length* *xs*] (*nth* *xs*)
⟨*proof*⟩

lemma *is-empty-empty* [*simp*]: *is-empty* *empty*
⟨*proof*⟩

lemma *is-empty-update* [*simp*]: ¬ *is-empty* (*update* *k* *v* *m*)
⟨*proof*⟩

lemma *is-empty-delete*: *is-empty* (*delete* *k* *m*) ⟷ *is-empty* *m* ∨ *keys* *m* = {*k*}
⟨*proof*⟩

lemma *is-empty-replace* [*simp*]: *is-empty* (*replace* *k* *v* *m*) ⟷ *is-empty* *m*
⟨*proof*⟩

lemma *is-empty-default* [*simp*]: ¬ *is-empty* (*default* *k* *v* *m*)
⟨*proof*⟩

lemma *is-empty-map-entry* [*simp*]: *is-empty* (*map-entry* *k* *f* *m*) ⟷ *is-empty* *m*
⟨*proof*⟩

lemma *is-empty-map-values* [*simp*]: *is-empty* (*map-values* *f* *m*) ⟷ *is-empty* *m*
⟨*proof*⟩

lemma *is-empty-map-default* [*simp*]: ¬ *is-empty* (*map-default* *k* *v* *f* *m*)
⟨*proof*⟩

lemma *keys-dom-lookup*: *keys* *m* = *dom* (*Mapping.lookup* *m*)
⟨*proof*⟩

lemma *keys-empty* [*simp*]: *keys* *empty* = {}
⟨*proof*⟩

lemma *keys-update* [*simp*]: *keys* (*update* *k* *v* *m*) = *insert* *k* (*keys* *m*)
⟨*proof*⟩

lemma *keys-delete* [*simp*]: *keys* (*delete* *k* *m*) = *keys* *m* − {*k*}
⟨*proof*⟩

lemma *keys-replace* [*simp*]: *keys* (*replace* *k* *v* *m*) = *keys* *m*
⟨*proof*⟩

lemma *keys-default* [*simp*]: *keys* (*default k v m*) = *insert k* (*keys m*)
⟨*proof*⟩

lemma *keys-map-entry* [*simp*]: *keys* (*map-entry k f m*) = *keys m*
⟨*proof*⟩

lemma *keys-map-default* [*simp*]: *keys* (*map-default k v f m*) = *insert k* (*keys m*)
⟨*proof*⟩

lemma *keys-map-values* [*simp*]: *keys* (*map-values f m*) = *keys m*
⟨*proof*⟩

lemma *keys-combine-with-key* [*simp*]:
Mapping.keys (*combine-with-key f m1 m2*) = *Mapping.keys m1* ∪ *Mapping.keys m2*
⟨*proof*⟩

lemma *keys-combine* [*simp*]: *Mapping.keys* (*combine f m1 m2*) = *Mapping.keys m1* ∪ *Mapping.keys m2*
⟨*proof*⟩

lemma *keys-tabulate* [*simp*]: *keys* (*tabulate ks f*) = *set ks*
⟨*proof*⟩

lemma *keys-of-alist* [*simp*]: *keys* (*of-alist xs*) = *set* (*List.map fst xs*)
⟨*proof*⟩

lemma *keys-bulkload* [*simp*]: *keys* (*bulkload xs*) = {*0..<length xs*}
⟨*proof*⟩

lemma *distinct-ordered-keys* [*simp*]: *distinct* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-infinite* [*simp*]: ¬ *finite* (*keys m*) ⟹ *ordered-keys m* = []
⟨*proof*⟩

lemma *ordered-keys-empty* [*simp*]: *ordered-keys empty* = []
⟨*proof*⟩

lemma *ordered-keys-update* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*update k v m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹
ordered-keys (*update k v m*) = *insort k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-delete* [*simp*]: *ordered-keys* (*delete k m*) = *remove1 k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-replace* [*simp*]: *ordered-keys* (*replace k v m*) = *ordered-keys m*
⟨*proof*⟩

lemma *ordered-keys-default* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*default k v m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*default k v m*) = *insort k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-map-entry* [*simp*]: *ordered-keys* (*map-entry k f m*) = *ordered-keys m*
⟨*proof*⟩

lemma *ordered-keys-map-default* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*map-default k v f m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*map-default k v f m*) = *insort k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-tabulate* [*simp*]: *ordered-keys* (*tabulate ks f*) = *sort* (*remdups ks*)
⟨*proof*⟩

lemma *ordered-keys-bulkload* [*simp*]: *ordered-keys* (*bulkload ks*) = [*0*..<*length ks*]
⟨*proof*⟩

lemma *tabulate-fold*: *tabulate xs f* = *fold* (λ*k m*. *update k* (*f k*) *m*) *xs empty*
⟨*proof*⟩

lemma *All-mapping-mono*:
(⋀*k v*. *k* ∈ *keys m* ⟹ *P k v* ⟹ *Q k v*) ⟹ *All-mapping m P* ⟹ *All-mapping m Q*
⟨*proof*⟩

lemma *All-mapping-empty* [*simp*]: *All-mapping Mapping.empty P*
⟨*proof*⟩

lemma *All-mapping-update-iff*:
All-mapping (*Mapping.update k v m*) *P* ⟷ *P k v* ∧ *All-mapping m* (λ*k' v'*. *k* = *k'* ∨ *P k' v'*)
⟨*proof*⟩

lemma *All-mapping-update*:
P k v ⟹ *All-mapping m* (λ*k' v'*. *k* = *k'* ∨ *P k' v'*) ⟹ *All-mapping* (*Mapping.update k v m*) *P*
⟨*proof*⟩

lemma *All-mapping-filter-iff*: *All-mapping* (*filter P m*) *Q* ⟷ *All-mapping m* (λ*k*

$v.\ P\ k\ v \longrightarrow Q\ k\ v)$
⟨*proof*⟩

lemma *All-mapping-filter*: *All-mapping m Q* ⟹ *All-mapping* (*filter P m*) *Q*
⟨*proof*⟩

lemma *All-mapping-map-values*: *All-mapping* (*map-values f m*) *P* ⟷ *All-mapping m* (λ*k v*. *P k* (*f k v*))
⟨*proof*⟩

lemma *All-mapping-tabulate*: (∀ *x*∈*set xs*. *P x* (*f x*)) ⟹ *All-mapping* (*Mapping.tabulate xs f*) *P*
⟨*proof*⟩

lemma *All-mapping-alist*:
(⋀*k v*. (*k*, *v*) ∈ *set xs* ⟹ *P k v*) ⟹ *All-mapping* (*Mapping.of-alist xs*) *P*
⟨*proof*⟩

lemma *combine-empty* [*simp*]: *combine f Mapping.empty y* = *y combine f y Mapping.empty* = *y*
⟨*proof*⟩

lemma (**in** *abel-semigroup*) *comm-monoid-set-combine*: *comm-monoid-set* (*combine f*) *Mapping.empty*
⟨*proof*⟩

locale *combine-mapping-abel-semigroup* = *abel-semigroup*
begin

sublocale *combine*: *comm-monoid-set combine f Mapping.empty*
⟨*proof*⟩

lemma *fold-combine-code*:
combine.F g (*set xs*) = *foldr* (λ*x*. *combine f* (*g x*)) (*remdups xs*) *Mapping.empty*
⟨*proof*⟩

lemma *keys-fold-combine*: *finite A* ⟹ *Mapping.keys* (*combine.F g A*) = (⋃*x*∈*A*. *Mapping.keys* (*g x*))
⟨*proof*⟩

end

58.6 Code generator setup

hide-const (**open**) *empty is-empty rep lookup lookup-default filter update delete ordered-keys*
keys size replace default map-entry map-default tabulate bulkload map map-values combine of-alist

end

59 Implementation of mappings with Association Lists

theory *AList-Mapping*
imports *AList Mapping*
begin

lift-definition *Mapping* :: $('a \times 'b)$ *list* $\Rightarrow$ $('a, 'b)$ *mapping* **is** *map-of* ⟨*proof*⟩

code-datatype *Mapping*

lemma *lookup-Mapping* [*simp*, *code*]: *Mapping.lookup* (*Mapping xs*) = *map-of xs*
⟨*proof*⟩

lemma *keys-Mapping* [*simp*, *code*]: *Mapping.keys* (*Mapping xs*) = *set* (*map fst xs*)
⟨*proof*⟩

lemma *empty-Mapping* [*code*]: *Mapping.empty* = *Mapping* []
⟨*proof*⟩

lemma *is-empty-Mapping* [*code*]: *Mapping.is-empty* (*Mapping xs*) ⟷ *List.null xs*
⟨*proof*⟩

lemma *update-Mapping* [*code*]: *Mapping.update k v* (*Mapping xs*) = *Mapping* (*AList.update k v xs*)
⟨*proof*⟩

lemma *delete-Mapping* [*code*]: *Mapping.delete k* (*Mapping xs*) = *Mapping* (*AList.delete k xs*)
⟨*proof*⟩

lemma *ordered-keys-Mapping* [*code*]:
Mapping.ordered-keys (*Mapping xs*) = *sort* (*remdups* (*map fst xs*))
⟨*proof*⟩

lemma *size-Mapping* [*code*]: *Mapping.size* (*Mapping xs*) = *length* (*remdups* (*map fst xs*))
⟨*proof*⟩

lemma *tabulate-Mapping* [*code*]: *Mapping.tabulate ks f* = *Mapping* (*map* (λ*k*. (*k*, *f k*)) *ks*)
⟨*proof*⟩

lemma *bulkload-Mapping* [*code*]:
Mapping.bulkload vs = *Mapping* (*map* (λ*n*. (*n*, *vs* ! *n*)) [*0*..<*length vs*])

⟨*proof*⟩

lemma *equal-Mapping* [*code*]:
HOL.equal (*Mapping xs*) (*Mapping ys*) ⟷
(*let ks* = *map fst xs*; *ls* = *map fst ys*
in (∀ *l*∈*set ls*. *l* ∈ *set ks*) ∧ (∀ *k*∈*set ks*. *k* ∈ *set ls* ∧ *map-of xs k* = *map-of ys k*))
⟨*proof*⟩

lemma *map-values-Mapping* [*code*]:
Mapping.map-values f (*Mapping xs*) = *Mapping* (*map* (λ(*x*,*y*). (*x*, *f x y*)) *xs*)
for *f* :: *'c* ⇒ *'a* ⇒ *'b* **and** *xs* :: (*'c* × *'a*) *list*
⟨*proof*⟩

lemma *combine-with-key-code* [*code*]:
Mapping.combine-with-key f (*Mapping xs*) (*Mapping ys*) =
Mapping.tabulate (*remdups* (*map fst xs* @ *map fst ys*))
(λ*x*. *the* (*combine-options* (*f x*) (*map-of xs x*) (*map-of ys x*)))
⟨*proof*⟩

lemma *combine-code* [*code*]:
Mapping.combine f (*Mapping xs*) (*Mapping ys*) =
Mapping.tabulate (*remdups* (*map fst xs* @ *map fst ys*))
(λ*x*. *the* (*combine-options f* (*map-of xs x*) (*map-of ys x*)))
⟨*proof*⟩

lemma *map-of-filter-distinct*:
assumes *distinct* (*map fst xs*)
shows *map-of* (*filter P xs*) *x* =
(*case map-of xs x of*
None ⇒ *None*
| *Some y* ⇒ *if P* (*x*,*y*) *then Some y else None*)
⟨*proof*⟩

lemma *filter-Mapping* [*code*]:
Mapping.filter P (*Mapping xs*) = *Mapping* (*filter* (λ(*k*,*v*). *P k v*) (*AList.clearjunk xs*))
⟨*proof*⟩

lemma [*code nbe*]: *HOL.equal* (*x* :: (*'a*, *'b*) *mapping*) *x* ⟷ *True*
⟨*proof*⟩

end

60 A simple cookbook example how to eliminate choice in programs.

theory *Execute-Choice*

imports *Main* ~~/*src/HOL/Library/AList-Mapping*
begin

A trivial example:

definition *valuesum* :: ('*a*, '*b* :: *ab-group-add*) *mapping* ⇒ '*b* **where**
valuesum m = (∑ *k* ∈ *Mapping.keys m*. *the* (*Mapping.lookup m k*))

Not that instead of defining *valuesum* with choice, we define it directly and derive a description involving choice afterwards:

lemma *valuesum-rec*:
assumes *fin*: *finite* (*dom* (*Mapping.lookup m*))
shows *valuesum m* = (*if Mapping.is-empty m then 0 else*
let l = (*SOME l*. *l* ∈ *Mapping.keys m*) *in the* (*Mapping.lookup m l*) + *valuesum* (*Mapping.delete l m*))
⟨*proof*⟩

In the context of the else-branch we can show that the exact choice is irrelevant; in practice, finding this point where choice becomes irrelevant is the most difficult thing!

lemma *valuesum-choice*:
finite (*Mapping.keys M*) ⟹ *x* ∈ *Mapping.keys M* ⟹ *y* ∈ *Mapping.keys M* ⟹
the (*Mapping.lookup M x*) + *valuesum* (*Mapping.delete x M*) =
the (*Mapping.lookup M y*) + *valuesum* (*Mapping.delete y M*)
⟨*proof*⟩

Given *valuesum-rec* as initial description, we stepwise refine it to something executable; first, we formally insert the constructor *AList-Mapping.Mapping* and split the one equation into two, where the second one provides the necessary context:

lemma *valuesum-rec-Mapping*:
shows [*code*]: *valuesum* (*Mapping* []) = *0*
and *valuesum* (*Mapping* (*x* # *xs*)) = (*let l* = (*SOME l*. *l* ∈ *Mapping.keys* (*Mapping* (*x* # *xs*))) *in*
the (*Mapping.lookup* (*Mapping* (*x* # *xs*)) *l*) + *valuesum* (*Mapping.delete l* (*Mapping* (*x* # *xs*))))
⟨*proof*⟩

As a side effect the precondition disappears (but note this has nothing to do with choice!). The first equation deals with the uncritical empty case and can already be used for code generation.

Using *valuesum-choice*, we are able to prove an executable version of *valuesum*:

lemma *valuesum-rec-exec* [*code*]:
valuesum (*Mapping* (*x* # *xs*)) = (*let l* = *fst* (*hd* (*x* # *xs*)) *in*
the (*Mapping.lookup* (*Mapping* (*x* # *xs*)) *l*) + *valuesum* (*Mapping.delete l* (*Mapping* (*x* # *xs*))))
⟨*proof*⟩

See how it works:

value *valuesum* (*Mapping* [(*''abc''*, (*42*::*int*)), (*''def''*, *1705*)])

end

61 Theory of Integration on real intervals

theory *Gauge-Integration*
imports *Complex-Main*
begin

Attention: This theory defines the Integration on real intervals. This is just a example theory for historical / expository interests. A better replacement is found in the Multivariate Analysis library. This defines the gauge integral on real vector spaces and in the Real Integral theory is a specialization to the integral on arbitrary real intervals. The Multivariate Analysis package also provides a better support for analysis on integrals.

We follow John Harrison in formalizing the Gauge integral.

61.1 Gauges

definition
  gauge :: [*real set*, *real* => *real*] => *bool* **where**
  gauge E g = (∀ *x*∈*E*. *0* < *g*(*x*))

61.2 Gauge-fine divisions

inductive
  fine :: [*real* ⇒ *real*, *real* × *real*, (*real* × *real* × *real*) *list*] ⇒ *bool*
for
  δ :: *real* ⇒ *real*
where
  fine-Nil:
    fine δ (*a*, *a*) []
| *fine-Cons*:
    ⟦*fine δ* (*b*, *c*) *D*; *a* < *b*; *a* ≤ *x*; *x* ≤ *b*; *b* − *a* < *δ x*⟧
      ⟹ *fine δ* (*a*, *c*) ((*a*, *x*, *b*) # *D*)

lemmas *fine-induct* [*induct set*: *fine*] =
  fine.induct [*of δ* (*a*,*b*) *D case-prod P*, *unfolded split-conv*] **for** *δ a b D P*

lemma *fine-single*:
  ⟦*a* < *b*; *a* ≤ *x*; *x* ≤ *b*; *b* − *a* < *δ x*⟧ ⟹ *fine δ* (*a*, *b*) [(*a*, *x*, *b*)]
⟨*proof*⟩

lemma *fine-append*:
  ⟦*fine δ* (*a*, *b*) *D*; *fine δ* (*b*, *c*) *D'*⟧ ⟹ *fine δ* (*a*, *c*) (*D* @ *D'*)

⟨*proof*⟩

lemma *fine-imp-le*: *fine* δ $(a,\ b)$ $D \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *nonempty-fine-imp-less*: $[\![$*fine* δ $(a,\ b)$ $D;\ D \neq []]\!] \Longrightarrow a < b$
⟨*proof*⟩

lemma *fine-Nil-iff*: *fine* δ $(a,\ b)$ $[] \longleftrightarrow a = b$
⟨*proof*⟩

lemma *fine-same-iff*: *fine* δ $(a,\ a)$ $D \longleftrightarrow D = []$
⟨*proof*⟩

lemma *empty-fine-imp-eq*: $[\![$*fine* δ $(a,\ b)$ $D;\ D = []]\!] \Longrightarrow a = b$
⟨*proof*⟩

lemma *mem-fine*:
$[\![$*fine* δ $(a,\ b)$ $D;\ (u,\ x,\ v) \in$ *set* $D]\!] \Longrightarrow u < v \wedge u \leq x \wedge x \leq v$
⟨*proof*⟩

lemma *mem-fine2*: $[\![$*fine* δ $(a,\ b)$ $D;\ (u,\ z,\ v) \in$ *set* $D]\!] \Longrightarrow a \leq u \wedge v \leq b$
⟨*proof*⟩

lemma *mem-fine3*: $[\![$*fine* δ $(a,\ b)$ $D;\ (u,\ z,\ v) \in$ *set* $D]\!] \Longrightarrow v - u < \delta\ z$
⟨*proof*⟩

lemma *BOLZANO*:
fixes P :: *real* ⇒ *real* ⇒ *bool*
assumes *1*: $a \leq b$
assumes *2*: $\bigwedge a\ b\ c.\ [\![P\ a\ b;\ P\ b\ c;\ a \leq b;\ b \leq c]\!] \Longrightarrow P\ a\ c$
assumes *3*: $\bigwedge x.\ \exists\, d{>}0.\ \forall\, a\ b.\ a \leq x\ \&\ x \leq b\ \&\ (b{-}a) < d \longrightarrow P\ a\ b$
shows $P\ a\ b$
⟨*proof*⟩

We can always find a division that is fine wrt any gauge

lemma *fine-exists*:
assumes $a \leq b$ **and** *gauge* $\{a..b\}$ δ **shows** $\exists\, D.$ *fine* δ $(a,\ b)$ D
⟨*proof*⟩

lemma *fine-covers-all*:
assumes *fine* δ $(a,\ c)$ D **and** $a < x$ **and** $x \leq c$
shows $\exists\ N <$ *length* $D.\ \forall\ d\ t\ e.\ D\ !\ N = (d,t,e) \longrightarrow d < x \wedge x \leq e$
⟨*proof*⟩

lemma *fine-append-split*:
assumes *fine* δ (a,b) D **and** $D2 \neq []$ **and** $D = D1\ @\ D2$
shows *fine* δ $(a,fst\ (hd\ D2))$ $D1$ (**is** *?fine1*)
and *fine* δ $(fst\ (hd\ D2),\ b)$ $D2$ (**is** *?fine2*)

⟨*proof*⟩

lemma *fine-δ-expand*:
assumes *fine* δ (*a*,*b*) *D*
and ⋀*x*. *a* ≤ *x* ⟹ *x* ≤ *b* ⟹ δ *x* ≤ δ′ *x*
shows *fine* δ′ (*a*,*b*) *D*
⟨*proof*⟩

lemma *fine-single-boundaries*:
assumes *fine* δ (*a*,*b*) *D* **and** *D* = [(*d*, *t*, *e*)]
shows *a* = *d* ∧ *b* = *e*
⟨*proof*⟩

lemma *fine-sum-list-eq-diff*:
fixes *f* :: *real* ⇒ *real*
shows *fine* δ (*a*, *b*) *D* ⟹ (∑ (*u*, *x*, *v*)←*D*. *f v* − *f u*) = *f b* − *f a*
⟨*proof*⟩

Lemmas about combining gauges

lemma *gauge-min*:
[| *gauge*(*E*) *g1*; *gauge*(*E*) *g2* |]
==> *gauge*(*E*) (%*x*. *min* (*g1*(*x*)) (*g2*(*x*)))
⟨*proof*⟩

lemma *fine-min*:
fine (%*x*. *min* (*g1*(*x*)) (*g2*(*x*))) (*a*,*b*) *D*
==> *fine*(*g1*) (*a*,*b*) *D* & *fine*(*g2*) (*a*,*b*) *D*
⟨*proof*⟩

61.3 Riemann sum

definition
rsum :: [(*real* × *real* × *real*) *list*, *real* ⇒ *real*] ⇒ *real* **where**
rsum D f = (∑ (*u*, *x*, *v*)←*D*. *f x* ∗ (*v* − *u*))

lemma *rsum-Nil* [*simp*]: *rsum* [] *f* = *0*
⟨*proof*⟩

lemma *rsum-Cons* [*simp*]: *rsum* ((*u*, *x*, *v*) # *D*) *f* = *f x* ∗ (*v* − *u*) + *rsum D f*
⟨*proof*⟩

lemma *rsum-zero* [*simp*]: *rsum D* (λ*x*. *0*) = *0*
⟨*proof*⟩

lemma *rsum-left-distrib*: *rsum D f* ∗ *c* = *rsum D* (λ*x*. *f x* ∗ *c*)
⟨*proof*⟩

lemma *rsum-right-distrib*: *c* ∗ *rsum D f* = *rsum D* (λ*x*. *c* ∗ *f x*)
⟨*proof*⟩

lemma *rsum-add*: *rsum D* ($\lambda x.\ f\ x\ +\ g\ x$) = *rsum D f* + *rsum D g*
⟨*proof*⟩

lemma *rsum-append*: *rsum* (*D1* @ *D2*) *f* = *rsum D1 f* + *rsum D2 f*
⟨*proof*⟩

61.4 Gauge integrability (definite)

definition
Integral :: [(*real*∗*real*),*real*=>*real*,*real*] => *bool* **where**
Integral = (%(*a*,*b*) *f k*. ∀ *e* > *0*.
(∃ δ. *gauge* {*a* .. *b*} δ &
(∀ *D*. *fine* δ (*a*,*b*) *D* −−>
|*rsum D f* − *k*| < *e*)))

lemma *Integral-eq*:
Integral (*a*, *b*) *f k* ⟷
(∀ *e*>*0*. ∃ δ. *gauge* {*a*..*b*} δ ∧ (∀ *D*. *fine* δ (*a*,*b*) *D* ⟶ |*rsum D f* − *k*| < *e*))
⟨*proof*⟩

lemma *IntegralI*:
assumes ⋀*e*. *0* < *e* ⟹
∃ δ. *gauge* {*a*..*b*} δ ∧ (∀ *D*. *fine* δ (*a*, *b*) *D* ⟶ |*rsum D f* − *k*| < *e*)
shows *Integral* (*a*, *b*) *f k*
⟨*proof*⟩

lemma *IntegralE*:
assumes *Integral* (*a*, *b*) *f k* **and** *0* < *e*
obtains δ **where** *gauge* {*a*..*b*} δ **and** ∀ *D*. *fine* δ (*a*, *b*) *D* ⟶ |*rsum D f* − *k*|
< *e*
⟨*proof*⟩

lemma *Integral-def2*:
Integral = (%(*a*,*b*) *f k*. ∀ *e*>*0*. (∃ δ. *gauge* {*a*..*b*} δ &
(∀ *D*. *fine* δ (*a*,*b*) *D* −−>
|*rsum D f* − *k*| ≤ *e*)))
⟨*proof*⟩

The integral is unique if it exists

lemma *Integral-unique*:
assumes *le*: *a* ≤ *b*
assumes *1*: *Integral* (*a*, *b*) *f k1*
assumes *2*: *Integral* (*a*, *b*) *f k2*
shows *k1* = *k2*
⟨*proof*⟩

lemma *Integral-zero*: *Integral*(*a*,*a*) *f 0*
⟨*proof*⟩

lemma *Integral-same-iff* [*simp*]: *Integral* $(a,\ a)\ f\ k \longleftrightarrow k = 0$
⟨*proof*⟩

lemma *Integral-zero-fun*: *Integral* $(a,b)\ (\lambda x.\ 0)\ 0$
⟨*proof*⟩

lemma *fine-rsum-const*: *fine* $\delta\ (a,b)\ D \Longrightarrow rsum\ D\ (\lambda x.\ c) = (c * (b - a))$
⟨*proof*⟩

lemma *Integral-mult-const*: $a \leq b \Longrightarrow Integral(a,b)\ (\lambda x.\ c)\ (c * (b - a))$
⟨*proof*⟩

lemma *Integral-eq-diff-bounds*: $a \leq b \Longrightarrow Integral(a,b)\ (\lambda x.\ 1)\ (b - a)$
⟨*proof*⟩

lemma *Integral-mult*:
$[|\ a \leq b;\ Integral(a,b)\ f\ k\ |] ==> Integral(a,b)\ (\%x.\ c * f\ x)\ (c * k)$
⟨*proof*⟩

lemma *Integral-add*:
assumes *Integral* $(a,\ b)\ f\ x1$
assumes *Integral* $(b,\ c)\ f\ x2$
assumes $a \leq b$ **and** $b \leq c$
shows *Integral* $(a,\ c)\ f\ (x1 + x2)$
⟨*proof*⟩

Fundamental theorem of calculus (Part I)

"Straddle Lemma" : Swartz and Thompson: AMM 95(7) 1988

lemma *strad1*:
fixes $z\ x\ s\ e$ :: *real*
assumes *P*: $(\bigwedge z.\ z \neq x \Longrightarrow |z - x| < s \Longrightarrow |(f\ z - f\ x)\ /\ (z - x) - f'\ x| < e\ /\ 2)$
assumes $|z - x| < s$
shows $|f\ z - f\ x - f'\ x * (z - x)| \leq e\ /\ 2 * |z - x|$
⟨*proof*⟩

lemma *lemma-straddle*:
assumes f': $\forall x.\ a \leq x\ \&\ x \leq b$ −−> *DERIV* $f\ x$:> $f'(x)$ **and** $0 < e$
shows $\exists g.\ gauge\ \{a..b\}\ g\ \&$
$(\forall x\ u\ v.\ a \leq u\ \&\ u \leq x\ \&\ x \leq v\ \&\ v \leq b\ \&\ (v - u) < g(x)$
−−> $|(f(v) - f(u)) - (f'(x) * (v - u))| \leq e * (v - u))$
⟨*proof*⟩

lemma *fundamental-theorem-of-calculus*:
assumes $a \leq b$
assumes f': $\forall x.\ a \leq x \wedge x \leq b \longrightarrow$ *DERIV* $f\ x$:> $f'(x)$
shows *Integral* $(a,\ b)\ f'\ (f(b) - f(a))$
⟨*proof*⟩

end

theory *Dedekind-Real*
imports *Complex-Main*
begin

62 Positive real numbers

Could be generalized and moved to *Groups*

lemma *add-eq-exists*: $\exists\, x.\ a+x = (b::rat)$
⟨*proof*⟩

definition
cut :: *rat set* => *bool* **where**
cut A = ({} ⊂ *A* &
A < {*r*. *0* < *r*} &
(∀ *y* ∈ *A*. ((∀ *z*. *0*<*z* & *z* < *y* −−> *z* ∈ *A*) & (∃ *u* ∈ *A*. *y* < *u*))))

lemma *interval-empty-iff*:
{*y*. (*x*::'*a*::*unbounded-dense-linorder*) < *y* ∧ *y* < *z*} = {} ⟷ ¬ *x* < *z*
⟨*proof*⟩

lemma *cut-of-rat*:
assumes *q*: *0* < *q* **shows** *cut* {*r*::*rat*. *0* < *r* & *r* < *q*} (**is** *cut ?A*)
⟨*proof*⟩

typedef *preal* = *Collect cut*
⟨*proof*⟩

lemma *Abs-preal-induct* [*induct type*: *preal*]:
(⋀*x*. *cut x* ⟹ *P* (*Abs-preal x*)) ⟹ *P x*
⟨*proof*⟩

lemma *Rep-preal*:
cut (*Rep-preal x*)
⟨*proof*⟩

definition
psup :: *preal set* => *preal* **where**
psup P = *Abs-preal* (⋃ *X* ∈ *P*. *Rep-preal X*)

definition
add-set :: [*rat set*,*rat set*] => *rat set* **where**
add-set A B = {*w*. ∃ *x* ∈ *A*. ∃ *y* ∈ *B*. *w* = *x* + *y*}

definition
diff-set :: [*rat set*,*rat set*] => *rat set* **where**
diff-set A B = {*w*. ∃ *x*. *0* < *w* & *0* < *x* & *x* ∉ *B* & *x* + *w* ∈ *A*}

definition
mult-set :: [*rat set*,*rat set*] => *rat set* **where**
mult-set A B = {*w*. ∃ *x* ∈ *A*. ∃ *y* ∈ *B*. *w* = *x* * *y*}

definition
inverse-set :: *rat set* => *rat set* **where**
inverse-set A = {*x*. ∃ *y*. *0* < *x* & *x* < *y* & *inverse y* ∉ *A*}

instantiation *preal* :: {*ord*, *plus*, *minus*, *times*, *inverse*, *one*}
begin

definition
preal-less-def:
R < *S* == *Rep-preal R* < *Rep-preal S*

definition
preal-le-def:
R ≤ *S* == *Rep-preal R* ⊆ *Rep-preal S*

definition
preal-add-def:
R + *S* == *Abs-preal* (*add-set* (*Rep-preal R*) (*Rep-preal S*))

definition
preal-diff-def:
R − *S* == *Abs-preal* (*diff-set* (*Rep-preal R*) (*Rep-preal S*))

definition
preal-mult-def:
R * *S* == *Abs-preal* (*mult-set* (*Rep-preal R*) (*Rep-preal S*))

definition
preal-inverse-def:
inverse R == *Abs-preal* (*inverse-set* (*Rep-preal R*))

definition *R div S* = *R* * *inverse* (*S*::*preal*)

definition
preal-one-def:
1 == *Abs-preal* {*x*. *0* < *x* & *x* < *1*}

instance ⟨*proof*⟩

end

Reduces equality on abstractions to equality on representatives

declare *Abs-preal-inject* [*simp*]
declare *Abs-preal-inverse* [*simp*]

lemma *rat-mem-preal*: $0 < q \Longrightarrow cut\ \{r::rat.\ 0 < r\ \&\ r < q\}$
⟨*proof*⟩

lemma *preal-nonempty*: $cut\ A \Longrightarrow \exists x{\in}A.\ 0 < x$
⟨*proof*⟩

lemma *preal-Ex-mem*: $cut\ A \Longrightarrow \exists x.\ x \in A$
⟨*proof*⟩

lemma *preal-imp-psubset-positives*: $cut\ A \Longrightarrow A < \{r.\ 0 < r\}$
⟨*proof*⟩

lemma *preal-exists-bound*: $cut\ A \Longrightarrow \exists x.\ 0 < x\ \&\ x \notin A$
⟨*proof*⟩

lemma *preal-exists-greater*: $[|\ cut\ A;\ y \in A\ |] \Longrightarrow \exists u \in A.\ y < u$
⟨*proof*⟩

lemma *preal-downwards-closed*: $[|\ cut\ A;\ y \in A;\ 0 < z;\ z < y\ |] \Longrightarrow z \in A$
⟨*proof*⟩

Relaxing the final premise

lemma *preal-downwards-closed′*:
$[|\ cut\ A;\ y \in A;\ 0 < z;\ z \leq y\ |] \Longrightarrow z \in A$
⟨*proof*⟩

A positive fraction not in a positive real is an upper bound. Gleason p. 122 - Remark (1)

lemma *not-in-preal-ub*:
assumes *A*: *cut A*
and *notx*: $x \notin A$
and *y*: $y \in A$
and *pos*: $0 < x$
shows $y < x$
⟨*proof*⟩

preal lemmas instantiated to *Rep-preal X*

lemma *mem-Rep-preal-Ex*: $\exists x.\ x \in Rep\text{-}preal\ X$
thm *preal-Ex-mem*
⟨*proof*⟩

lemma *Rep-preal-exists-bound*: $\exists x{>}0.\ x \notin Rep\text{-}preal\ X$
⟨*proof*⟩

lemmas *not-in-Rep-preal-ub* = *not-in-preal-ub* [*OF Rep-preal*]

62.1 Properties of Ordering

instance *preal* :: *order*
⟨*proof*⟩

lemma *preal-imp-pos*: [|*cut A*; $r \in A$|] ==> $0 < r$
⟨*proof*⟩

instance *preal* :: *linorder*
⟨*proof*⟩

instantiation *preal* :: *distrib-lattice*
begin

definition
(*inf* :: *preal* ⇒ *preal* ⇒ *preal*) = *min*

definition
(*sup* :: *preal* ⇒ *preal* ⇒ *preal*) = *max*

instance
⟨*proof*⟩

end

62.2 Properties of Addition

lemma *preal-add-commute*: (*x*::*preal*) + *y* = *y* + *x*
⟨*proof*⟩

Lemmas for proving that addition of two positive reals gives a positive real

Part 1 of Dedekind sections definition

lemma *add-set-not-empty*:
[|*cut A*; *cut B*|] ==> {} ⊂ *add-set A B*
⟨*proof*⟩

Part 2 of Dedekind sections definition. A structured version of this proof is *preal-not-mem-mult-set-Ex* below.

lemma *preal-not-mem-add-set-Ex*:
[|*cut A*; *cut B*|] ==> ∃ *q*>*0*. *q* ∉ *add-set A B*
⟨*proof*⟩

lemma *add-set-not-rat-set*:
assumes *A*: *cut A*
and *B*: *cut B*
shows *add-set A B* < {*r*. *0* < *r*}
⟨*proof*⟩

Part 3 of Dedekind sections definition

lemma *add-set-lemma3*:
[|*cut A*; *cut B*; *u* ∈ *add-set A B*; *0* < *z*; *z* < *u*|]
==> *z* ∈ *add-set A B*
⟨*proof*⟩

Part 4 of Dedekind sections definition

lemma *add-set-lemma4*:
[|*cut A*; *cut B*; *y* ∈ *add-set A B*|] ==> ∃ *u* ∈ *add-set A B*. *y* < *u*
⟨*proof*⟩

lemma *mem-add-set*:
[|*cut A*; *cut B*|] ==> *cut* (*add-set A B*)
⟨*proof*⟩

lemma *preal-add-assoc*: ((*x*::*preal*) + *y*) + *z* = *x* + (*y* + *z*)
⟨*proof*⟩

instance *preal* :: *ab-semigroup-add*
⟨*proof*⟩

62.3 Properties of Multiplication

Proofs essentially same as for addition

lemma *preal-mult-commute*: (*x*::*preal*) * *y* = *y* * *x*
⟨*proof*⟩

Multiplication of two positive reals gives a positive real.

Lemmas for proving positive reals multiplication set in *preal*

Part 1 of Dedekind sections definition

lemma *mult-set-not-empty*:
[|*cut A*; *cut B*|] ==> {} ⊂ *mult-set A B*
⟨*proof*⟩

Part 2 of Dedekind sections definition

lemma *preal-not-mem-mult-set-Ex*:
assumes *A*: *cut A*
and *B*: *cut B*
shows ∃ *q*. *0* < *q* & *q* ∉ *mult-set A B*
⟨*proof*⟩

lemma *mult-set-not-rat-set*:
assumes *A*: *cut A*
and *B*: *cut B*
shows *mult-set A B* < {*r*. *0* < *r*}
⟨*proof*⟩

Part 3 of Dedekind sections definition

lemma *mult-set-lemma3*:
[|*cut A*; *cut B*; *u* ∈ *mult-set A B*; *0* < *z*; *z* < *u*|]
==> *z* ∈ *mult-set A B*
⟨*proof*⟩

Part 4 of Dedekind sections definition

lemma *mult-set-lemma4*:
[|*cut A*; *cut B*; *y* ∈ *mult-set A B*|] ==> ∃ *u* ∈ *mult-set A B*. *y* < *u*
⟨*proof*⟩

lemma *mem-mult-set*:
[|*cut A*; *cut B*|] ==> *cut* (*mult-set A B*)
⟨*proof*⟩

lemma *preal-mult-assoc*: ((*x*::*preal*) * *y*) * *z* = *x* * (*y* * *z*)
⟨*proof*⟩

instance *preal* :: *ab-semigroup-mult*
⟨*proof*⟩

Positive real 1 is the multiplicative identity element

lemma *preal-mult-1*: (*1*::*preal*) * *z* = *z*
⟨*proof*⟩

instance *preal* :: *comm-monoid-mult*
⟨*proof*⟩

62.4 Distribution of Multiplication across Addition

lemma *mem-Rep-preal-add-iff*:
(*z* ∈ *Rep-preal*(*R*+*S*)) = (∃ *x* ∈ *Rep-preal R*. ∃ *y* ∈ *Rep-preal S*. *z* = *x* + *y*)
⟨*proof*⟩

lemma *mem-Rep-preal-mult-iff*:
(*z* ∈ *Rep-preal*(*R***S*)) = (∃ *x* ∈ *Rep-preal R*. ∃ *y* ∈ *Rep-preal S*. *z* = *x* * *y*)
⟨*proof*⟩

lemma *distrib-subset1*:
Rep-preal (*w* * (*x* + *y*)) ⊆ *Rep-preal* (*w* * *x* + *w* * *y*)
⟨*proof*⟩

lemma *preal-add-mult-distrib-mean*:
assumes *a*: *a* ∈ *Rep-preal w*
and *b*: *b* ∈ *Rep-preal w*
and *d*: *d* ∈ *Rep-preal x*
and *e*: *e* ∈ *Rep-preal y*
shows ∃ *c* ∈ *Rep-preal w*. *a* * *d* + *b* * *e* = *c* * (*d* + *e*)
⟨*proof*⟩

lemma *distrib-subset2*:
Rep-preal $(w * x + w * y) \subseteq$ *Rep-preal* $(w * (x + y))$
⟨*proof*⟩

lemma *preal-add-mult-distrib2*: $(w * ((x::preal) + y)) = (w * x) + (w * y)$
⟨*proof*⟩

lemma *preal-add-mult-distrib*: $(((x::preal) + y) * w) = (x * w) + (y * w)$
⟨*proof*⟩

instance *preal* :: *comm-semiring*
⟨*proof*⟩

62.5 Existence of Inverse, a Positive Real

lemma *mem-inv-set-ex*:
assumes *A*: *cut A* **shows** $\exists x\ y.\ 0 < x\ \&\ x < y\ \&$ *inverse y* $\notin A$
⟨*proof*⟩

Part 1 of Dedekind sections definition

lemma *inverse-set-not-empty*:
cut A ==> $\{\} \subset$ *inverse-set A*
⟨*proof*⟩

Part 2 of Dedekind sections definition

lemma *preal-not-mem-inverse-set-Ex*:
assumes *A*: *cut A* **shows** $\exists q.\ 0 < q\ \&\ q \notin$ *inverse-set A*
⟨*proof*⟩

lemma *inverse-set-not-rat-set*:
assumes *A*: *cut A* **shows** *inverse-set A* $< \{r.\ 0 < r\}$
⟨*proof*⟩

Part 3 of Dedekind sections definition

lemma *inverse-set-lemma3*:
[|*cut A*; $u \in$ *inverse-set A*; $0 < z$; $z < u$|]
==> $z \in$ *inverse-set A*
⟨*proof*⟩

Part 4 of Dedekind sections definition

lemma *inverse-set-lemma4*:
[|*cut A*; $y \in$ *inverse-set A*|] ==> $\exists u \in$ *inverse-set A*. $y < u$
⟨*proof*⟩

lemma *mem-inverse-set*:
cut A ==> *cut* (*inverse-set A*)
⟨*proof*⟩

62.6 Gleason's Lemma 9-3.4, page 122

lemma *Gleason9-34-exists*:
assumes A: *cut* A
and $\forall x{\in}A.\ x + u \in A$
and $0 \leq z$
shows $\exists b{\in}A.\ b + (\textit{of-int}\ z) * u \in A$
⟨*proof*⟩

lemma *Gleason9-34-contra*:
assumes A: *cut* A
shows $[|\forall x{\in}A.\ x + u \in A;\ 0 < u;\ 0 < y;\ y \notin A|] ==> \textit{False}$
⟨*proof*⟩

lemma *Gleason9-34*:
assumes A: *cut* A
and *upos*: $0 < u$
shows $\exists r \in A.\ r + u \notin A$
⟨*proof*⟩

62.7 Gleason's Lemma 9-3.6

lemma *lemma-gleason9-36*:
assumes A: *cut* A
and x: $1 < x$
shows $\exists r \in A.\ r{*}x \notin A$
⟨*proof*⟩

62.8 Existence of Inverse: Part 2

lemma *mem-Rep-preal-inverse-iff*:
$(z \in \textit{Rep-preal}(\textit{inverse}\ R)) =$
$(0 < z \wedge (\exists y.\ z < y \wedge \textit{inverse}\ y \notin \textit{Rep-preal}\ R))$
⟨*proof*⟩

lemma *Rep-preal-one*:
$\textit{Rep-preal}\ 1 = \{x.\ 0 < x \wedge x < 1\}$
⟨*proof*⟩

lemma *subset-inverse-mult-lemma*:
assumes *xpos*: $0 < x$ **and** *xless*: $x < 1$
shows $\exists r\ u\ y.\ 0 < r\ \&\ r < y\ \&\ \textit{inverse}\ y \notin \textit{Rep-preal}\ R\ \&$
$u \in \textit{Rep-preal}\ R\ \&\ x = r * u$
⟨*proof*⟩

lemma *subset-inverse-mult*:
$\textit{Rep-preal}\ 1 \subseteq \textit{Rep-preal}(\textit{inverse}\ R * R)$
⟨*proof*⟩

lemma *inverse-mult-subset-lemma*:
  assumes *rpos*: $0 < r$
    and *rless*: $r < y$
    and *notin*: $inverse\ y \notin Rep\text{-}preal\ R$
    and *q*: $q \in Rep\text{-}preal\ R$
  shows $r*q < 1$
⟨*proof*⟩

lemma *inverse-mult-subset*:
    $Rep\text{-}preal(inverse\ R * R) \subseteq Rep\text{-}preal\ 1$
⟨*proof*⟩

lemma *preal-mult-inverse*: $inverse\ R * R = (1::preal)$
⟨*proof*⟩

lemma *preal-mult-inverse-right*: $R * inverse\ R = (1::preal)$
⟨*proof*⟩

Theorems needing *Gleason9-34*

lemma *Rep-preal-self-subset*: $Rep\text{-}preal\ (R) \subseteq Rep\text{-}preal(R + S)$
⟨*proof*⟩

lemma *Rep-preal-sum-not-subset*: $\sim\ Rep\text{-}preal\ (R + S) \subseteq Rep\text{-}preal(R)$
⟨*proof*⟩

lemma *Rep-preal-sum-not-eq*: $Rep\text{-}preal\ (R + S) \neq Rep\text{-}preal(R)$
⟨*proof*⟩

at last, Gleason prop. 9-3.5(iii) page 123

lemma *preal-self-less-add-left*: $(R::preal) < R + S$
⟨*proof*⟩

62.9 Subtraction for Positive Reals

Gleason prop. 9-3.5(iv), page 123: proving $A < B \Longrightarrow \exists\, D.\ A + D = B$. We define the claimed D and show that it is a positive real

Part 1 of Dedekind sections definition

lemma *diff-set-not-empty*:
    $R < S \Longrightarrow \{\} \subset diff\text{-}set\ (Rep\text{-}preal\ S)\ (Rep\text{-}preal\ R)$
⟨*proof*⟩

Part 2 of Dedekind sections definition

lemma *diff-set-nonempty*:
    $\exists\, q.\ 0 < q\ \&\ q \notin diff\text{-}set\ (Rep\text{-}preal\ S)\ (Rep\text{-}preal\ R)$
⟨*proof*⟩

lemma *diff-set-not-rat-set*:

diff-set (*Rep-preal S*) (*Rep-preal R*) < {*r*. *0* < *r*} (**is** *?lhs* < *?rhs*)
⟨*proof*⟩

Part 3 of Dedekind sections definition

lemma *diff-set-lemma3*:
[|*R* < *S*; *u* ∈ *diff-set* (*Rep-preal S*) (*Rep-preal R*); *0* < *z*; *z* < *u*|]
==> *z* ∈ *diff-set* (*Rep-preal S*) (*Rep-preal R*)
⟨*proof*⟩

Part 4 of Dedekind sections definition

lemma *diff-set-lemma4*:
[|*R* < *S*; *y* ∈ *diff-set* (*Rep-preal S*) (*Rep-preal R*)|]
==> ∃ *u* ∈ *diff-set* (*Rep-preal S*) (*Rep-preal R*). *y* < *u*
⟨*proof*⟩

lemma *mem-diff-set*:
R < *S* ==> *cut* (*diff-set* (*Rep-preal S*) (*Rep-preal R*))
⟨*proof*⟩

lemma *mem-Rep-preal-diff-iff*:
R < *S* ==>
(*z* ∈ *Rep-preal*(*S*−*R*)) =
(∃ *x*. *0* < *x* & *0* < *z* & *x* ∉ *Rep-preal R* & *x* + *z* ∈ *Rep-preal S*)
⟨*proof*⟩

proving that $R + D \leq S$

lemma *less-add-left-lemma*:
assumes *Rless*: *R* < *S*
and *a*: *a* ∈ *Rep-preal R*
and *cb*: *c* + *b* ∈ *Rep-preal S*
and *c* ∉ *Rep-preal R*
and *0* < *b*
and *0* < *c*
shows *a* + *b* ∈ *Rep-preal S*
⟨*proof*⟩

lemma *less-add-left-le1*:
R < (*S*::*preal*) ==> *R* + (*S*−*R*) ≤ *S*
⟨*proof*⟩

62.10 proving that $S \leq R + D$ — trickier

lemma *lemma-sum-mem-Rep-preal-ex*:
x ∈ *Rep-preal S* ==> ∃ *e*. *0* < *e* & *x* + *e* ∈ *Rep-preal S*
⟨*proof*⟩

lemma *less-add-left-lemma2*:
assumes *Rless*: *R* < *S*
and *x*: *x* ∈ *Rep-preal S*

and *xnot*: $x \notin$ *Rep-preal R*
shows $\exists u\ v\ z.\ 0 < v\ \&\ 0 < z\ \&\ u \in$ *Rep-preal R* $\&\ z \notin$ *Rep-preal R* $\&$
$z + v \in$ *Rep-preal S* $\&\ x = u + v$
⟨*proof*⟩

lemma *less-add-left-le2*: $R < (S::preal) \Longrightarrow S \leq R + (S-R)$
⟨*proof*⟩

lemma *less-add-left*: $R < (S::preal) \Longrightarrow R + (S-R) = S$
⟨*proof*⟩

lemma *less-add-left-Ex*: $R < (S::preal) \Longrightarrow \exists D.\ R + D = S$
⟨*proof*⟩

lemma *preal-add-less2-mono1*: $R < (S::preal) \Longrightarrow R + T < S + T$
⟨*proof*⟩

lemma *preal-add-less2-mono2*: $R < (S::preal) \Longrightarrow T + R < T + S$
⟨*proof*⟩

lemma *preal-add-right-less-cancel*: $R + T < S + T \Longrightarrow R < (S::preal)$
⟨*proof*⟩

lemma *preal-add-left-less-cancel*: $T + R < T + S \Longrightarrow R < (S::preal)$
⟨*proof*⟩

lemma *preal-add-less-cancel-left* [*simp*]: $(T + (R::preal) < T + S) = (R < S)$
⟨*proof*⟩

lemma *preal-add-less-cancel-right* [*simp*]: $((R::preal) + T < S + T) = (R < S)$
⟨*proof*⟩

lemma *preal-add-le-cancel-left* [*simp*]: $(T + (R::preal) \leq T + S) = (R \leq S)$
⟨*proof*⟩

lemma *preal-add-le-cancel-right* [*simp*]: $((R::preal) + T \leq S + T) = (R \leq S)$
⟨*proof*⟩

lemma *preal-add-right-cancel*: $(R::preal) + T = S + T \Longrightarrow R = S$
⟨*proof*⟩

lemma *preal-add-left-cancel*: $C + A = C + B \Longrightarrow A = (B::preal)$
⟨*proof*⟩

instance *preal* :: *linordered-ab-semigroup-add*
⟨*proof*⟩

62.11 Completeness of type *preal*

Prove that supremum is a cut

Part 1 of Dedekind sections definition

lemma *preal-sup-set-not-empty*:
$P \neq \{\} \Longrightarrow \{\} \subset (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X))$
⟨*proof*⟩

Part 2 of Dedekind sections definition

lemma *preal-sup-not-exists*:
$\forall X \in P.\ X \leq Y \Longrightarrow \exists q.\ 0 < q\ \&\ q \notin (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X))$
⟨*proof*⟩

lemma *preal-sup-set-not-rat-set*:
$\forall X \in P.\ X \leq Y \Longrightarrow (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X)) < \{r.\ 0 < r\}$
⟨*proof*⟩

Part 3 of Dedekind sections definition

lemma *preal-sup-set-lemma3*:
$[|P \neq \{\}; \forall X \in P.\ X \leq Y;\ u \in (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X));\ 0 < z;\ z < u|]$
$\Longrightarrow z \in (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X))$
⟨*proof*⟩

Part 4 of Dedekind sections definition

lemma *preal-sup-set-lemma4*:
$[|P \neq \{\}; \forall X \in P.\ X \leq Y;\ y \in (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X))\ |]$
$\Longrightarrow \exists u \in (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X)).\ y < u$
⟨*proof*⟩

lemma *preal-sup*:
$[|P \neq \{\}; \forall X \in P.\ X \leq Y|] \Longrightarrow \mathit{cut}\ (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X))$
⟨*proof*⟩

lemma *preal-psup-le*:
$[|\ \forall X \in P.\ X \leq Y;\ \ x \in P\ |] \Longrightarrow x \leq \mathit{psup}\ P$
⟨*proof*⟩

lemma *psup-le-ub*: $[|\ P \neq \{\}; \forall X \in P.\ X \leq Y\ |] \Longrightarrow \mathit{psup}\ P \leq Y$
⟨*proof*⟩

Supremum property

lemma *preal-complete*:
$[|\ P \neq \{\}; \forall X \in P.\ X \leq Y\ |] \Longrightarrow (\exists X \in P.\ Z < X) = (Z < \mathit{psup}\ P)$
⟨*proof*⟩

63 Defining the Reals from the Positive Reals

definition

realrel :: $((preal * preal) * (preal * preal))$ *set* **where**
realrel = $\{p.\ \exists x1\ y1\ x2\ y2.\ p = ((x1,y1),(x2,y2))\ \&\ x1+y2 = x2+y1\}$

definition *Real* = *UNIV*//*realrel*

typedef *real* = *Real*
morphisms *Rep-Real Abs-Real*
⟨*proof*⟩

definition

real-of-preal :: *preal* => *real* **where**
real-of-preal m = *Abs-Real* (*realrel* `` $\{(m + 1, 1)\}$)

instantiation *real* :: {*zero*, *one*, *plus*, *minus*, *uminus*, *times*, *inverse*, *ord*, *abs*, *sgn*}
begin

definition
real-zero-def: *0* = *Abs-Real*(*realrel*``$\{(1, 1)\}$)

definition
real-one-def: *1* = *Abs-Real*(*realrel*``$\{(1 + 1, 1)\}$)

definition
real-add-def: $z + w =$
the-elem ($\bigcup(x,y) \in Rep\text{-}Real(z).\ \bigcup(u,v) \in Rep\text{-}Real(w).$
$\{$ *Abs-Real*(*realrel*``$\{(x+u, y+v)\}$) $\}$)

definition
real-minus-def: $- r =$ *the-elem* ($\bigcup(x,y) \in Rep\text{-}Real(r).\ \{$ *Abs-Real*(*realrel*``$\{(y,x)\}$) $\}$)

definition
real-diff-def: $r - (s::real) = r + - s$

definition
real-mult-def:
$z * w =$
the-elem ($\bigcup(x,y) \in Rep\text{-}Real(z).\ \bigcup(u,v) \in Rep\text{-}Real(w).$
$\{$ *Abs-Real*(*realrel*``$\{(x*u + y*v, x*v + y*u)\}$) $\}$)

definition
real-inverse-def: *inverse* $(R::real) = (THE\ S.\ (R = 0\ \&\ S = 0) \mid S * R = 1)$

definition
real-divide-def: R *div* $(S::real) = R * inverse\ S$

definition

real-le-def: $z \leq (w::real) \longleftrightarrow$
$(\exists x\ y\ u\ v.\ x+v \leq u+y\ \&\ (x,y) \in Rep\text{-}Real\ z\ \&\ (u,v) \in Rep\text{-}Real\ w)$

definition
real-less-def: $x < (y::real) \longleftrightarrow x \leq y \wedge x \neq y$

definition
real-abs-def: $|r::real| = (if\ r < 0\ then\ -\ r\ else\ r)$

definition
real-sgn-def: $sgn\ (x::real) = (if\ x=0\ then\ 0\ else\ if\ 0<x\ then\ 1\ else\ -\ 1)$

instance ⟨*proof*⟩

end

63.1 Equivalence relation over positive reals

lemma *preal-trans-lemma*:
assumes $x + y1 = x1 + y$
and $x + y2 = x2 + y$
shows $x1 + y2 = x2 + (y1::preal)$
⟨*proof*⟩

lemma *realrel-iff* [*simp*]: $(((x1,y1),(x2,y2)) \in realrel) = (x1 + y2 = x2 + y1)$
⟨*proof*⟩

lemma *equiv-realrel*: *equiv UNIV realrel*
⟨*proof*⟩

Reduces equality of equivalence classes to the *Dedekind-Real.realrel* relation: $(Dedekind\text{-}Real.realrel\ ``\ \{x\} = Dedekind\text{-}Real.realrel\ ``\ \{y\}) = ((x,\ y) \in Dedekind\text{-}Real.realrel)$

lemmas *equiv-realrel-iff* =
eq-equiv-class-iff [*OF equiv-realrel UNIV-I UNIV-I*]

declare *equiv-realrel-iff* [*simp*]

lemma *realrel-in-real* [*simp*]: $realrel``\{(x,y)\}: Real$
⟨*proof*⟩

declare *Abs-Real-inject* [*simp*]
declare *Abs-Real-inverse* [*simp*]

Case analysis on the representation of a real number as an equivalence class of pairs of positive reals.

lemma *eq-Abs-Real* [*case-names Abs-Real*, *cases type*: *real*]:

(!!x y. z = *Abs-Real*(*realrel*``{(x,y)}) ==> P) ==> P
⟨*proof*⟩

63.2 Addition and Subtraction

lemma *real-add-congruent2-lemma*:
[|a + ba = aa + b; ab + bc = ac + bb|]
==> a + ab + (ba + bc) = aa + ac + (b + (bb::*preal*))
⟨*proof*⟩

lemma *real-add*:
Abs-Real (*realrel*``{($x$,$y$)}) + *Abs-Real* (*realrel*``{(u,v)}) =
Abs-Real (*realrel*``{(x+u, y+v)})
⟨*proof*⟩

lemma *real-minus*: − *Abs-Real*(*realrel*``{($x$,$y$)}) = *Abs-Real*(*realrel* `` {(y,x)})
⟨*proof*⟩

instance *real* :: *ab-group-add*
⟨*proof*⟩

63.3 Multiplication

lemma *real-mult-congruent2-lemma*:
!!(*x1*::*preal*). [| *x1* + *y2* = *x2* + *y1* |] ==>
x ∗ *x1* + *y* ∗ *y1* + (*x* ∗ *y2* + *y* ∗ *x2*) =
x ∗ *x2* + *y* ∗ *y2* + (*x* ∗ *y1* + *y* ∗ *x1*)
⟨*proof*⟩

lemma *real-mult-congruent2*:
(%*p1* *p2*.
(%(*x1*,*y1*). (%(*x2*,*y2*).
{ *Abs-Real* (*realrel*``{(*x1*∗*x2* + *y1*∗*y2*, *x1*∗*y2*+*y1*∗*x2*)}) }) *p2*) *p1*)
respects2 realrel
⟨*proof*⟩

lemma *real-mult*:
Abs-Real((*realrel*``{(*x1*,*y1*)})) ∗ *Abs-Real*((*realrel*``{(*x2*,*y2*)})) =
Abs-Real(*realrel* `` {(*x1*∗*x2*+*y1*∗*y2*,*x1*∗*y2*+*y1*∗*x2*)})
⟨*proof*⟩

lemma *real-mult-commute*: (*z*::*real*) ∗ *w* = *w* ∗ *z*
⟨*proof*⟩

lemma *real-mult-assoc*: ((*z1*::*real*) ∗ *z2*) ∗ *z3* = *z1* ∗ (*z2* ∗ *z3*)
⟨*proof*⟩

lemma *real-mult-1*: (*1*::*real*) ∗ *z* = *z*
⟨*proof*⟩

lemma *real-add-mult-distrib*: $((z1::real) + z2) * w = (z1 * w) + (z2 * w)$
⟨*proof*⟩

one and zero are distinct

lemma *real-zero-not-eq-one*: $0 \neq (1::real)$
⟨*proof*⟩

instance *real* :: *comm-ring-1*
⟨*proof*⟩

63.4 Inverse and Division

lemma *real-zero-iff*: *Abs-Real* (*realrel* `` $\{(x, x)\}$) = 0
⟨*proof*⟩

Instead of using an existential quantifier and constructing the inverse within the proof, we could define the inverse explicitly.

lemma *real-mult-inverse-left-ex*: $x \neq 0 \Longrightarrow \exists y.\ y*x = (1::real)$
⟨*proof*⟩

lemma *real-mult-inverse-left*: $x \neq 0 \Longrightarrow inverse(x)*x = (1::real)$
⟨*proof*⟩

63.5 The Real Numbers form a Field

instance *real* :: *field*
⟨*proof*⟩

63.6 The $\leq$ Ordering

lemma *real-le-refl*: $w \leq (w::real)$
⟨*proof*⟩

The arithmetic decision procedure is not set up for type preal. This lemma is currently unused, but it could simplify the proofs of the following two lemmas.

lemma *preal-eq-le-imp-le*:
assumes *eq*: $a+b = c+d$ **and** *le*: $c \leq a$
shows $b \leq (d::preal)$
⟨*proof*⟩

lemma *real-le-lemma*:
assumes *l*: $u1 + v2 \leq u2 + v1$
and $x1 + v1 = u1 + y1$
and $x2 + v2 = u2 + y2$
shows $x1 + y2 \leq x2 + (y1::preal)$
⟨*proof*⟩

lemma *real-le*:

$(Abs\text{-}Real(realrel``\{(x1,y1)\}) \leq Abs\text{-}Real(realrel``\{(x2,y2)\})) =$
$(x1 + y2 \leq x2 + y1)$
⟨*proof*⟩

lemma *real-le-antisym*: $[\!|\ z \leq w;\ w \leq z\ |\!] ==> z = (w::real)$
⟨*proof*⟩

lemma *real-trans-lemma*:
assumes $x + v \leq u + y$
and $u + v' \leq u' + v$
and $x2 + v2 = u2 + y2$
shows $x + v' \leq u' + (y::preal)$
⟨*proof*⟩

lemma *real-le-trans*: $[\!|\ i \leq j;\ j \leq k\ |\!] ==> i \leq (k::real)$
⟨*proof*⟩

instance *real* :: *order*
⟨*proof*⟩

lemma *real-le-linear*: $(z::real) \leq w \mid w \leq z$
⟨*proof*⟩

instance *real* :: *linorder*
⟨*proof*⟩

lemma *real-le-eq-diff*: $(x \leq y) = (x-y \leq (0::real))$
⟨*proof*⟩

lemma *real-add-left-mono*:
assumes *le*: $x \leq y$ **shows** $z + x \leq z + (y::real)$
⟨*proof*⟩

lemma *real-sum-gt-zero-less*: $(0 < S + (-W::real)) ==> (W < S)$
⟨*proof*⟩

lemma *real-less-sum-gt-zero*: $(W < S) ==> (0 < S + (-W::real))$
⟨*proof*⟩

lemma *real-mult-order*: $[\!|\ 0 < x;\ 0 < y\ |\!] ==> (0::real) < x * y$
⟨*proof*⟩

lemma *real-mult-less-mono2*: $[\!|\ (0::real) < z;\ x < y\ |\!] ==> z * x < z * y$
⟨*proof*⟩

instantiation *real* :: *distrib-lattice*
begin

definition
(*inf* :: *real* ⇒ *real* ⇒ *real*) = *min*

definition
(*sup* :: *real* ⇒ *real* ⇒ *real*) = *max*

instance
⟨*proof*⟩

end

63.7 The Reals Form an Ordered Field

instance *real* :: *linordered-field*
⟨*proof*⟩

The function *real-of-preal* requires many proofs, but it seems to be essential for proving completeness of the reals from that of the positive reals.

lemma *real-of-preal-add*:
real-of-preal ((*x*::*preal*) + *y*) = *real-of-preal x* + *real-of-preal y*
⟨*proof*⟩

lemma *real-of-preal-mult*:
real-of-preal ((*x*::*preal*) ∗ *y*) = *real-of-preal x*∗ *real-of-preal y*
⟨*proof*⟩

Gleason prop 9-4.4 p 127

lemma *real-of-preal-trichotomy*:
∃ *m*. (*x*::*real*) = *real-of-preal m* | *x* = *0* | *x* = −(*real-of-preal m*)
⟨*proof*⟩

lemma *real-of-preal-leD*:
real-of-preal m1 ≤ *real-of-preal m2* ==> *m1* ≤ *m2*
⟨*proof*⟩

lemma *real-of-preal-lessI*: *m1* < *m2* ==> *real-of-preal m1* < *real-of-preal m2*
⟨*proof*⟩

lemma *real-of-preal-lessD*:
real-of-preal m1 < *real-of-preal m2* ==> *m1* < *m2*
⟨*proof*⟩

lemma *real-of-preal-less-iff* [*simp*]:
(*real-of-preal m1* < *real-of-preal m2*) = (*m1* < *m2*)
⟨*proof*⟩

lemma *real-of-preal-le-iff*:
(*real-of-preal m1* ≤ *real-of-preal m2*) = (*m1* ≤ *m2*)
⟨*proof*⟩

lemma *real-of-preal-zero-less*: $0 < \text{real-of-preal } m$
⟨*proof*⟩

lemma *real-of-preal-minus-less-zero*: $- \text{real-of-preal } m < 0$
⟨*proof*⟩

lemma *real-of-preal-not-minus-gt-zero*: $\sim\ 0 < - \text{real-of-preal } m$
⟨*proof*⟩

63.8 Theorems About the Ordering

lemma *real-gt-zero-preal-Ex*: $(0 < x) = (\exists\, y.\ x = \text{real-of-preal } y)$
⟨*proof*⟩

lemma *real-gt-preal-preal-Ex*:
$\text{real-of-preal } z < x \Longrightarrow \exists\, y.\ x = \text{real-of-preal } y$
⟨*proof*⟩

lemma *real-ge-preal-preal-Ex*:
$\text{real-of-preal } z \leq x \Longrightarrow \exists\, y.\ x = \text{real-of-preal } y$
⟨*proof*⟩

lemma *real-less-all-preal*: $y \leq 0 \Longrightarrow \forall\, x.\ y < \text{real-of-preal } x$
⟨*proof*⟩

lemma *real-less-all-real2*: $\sim\ 0 < y \Longrightarrow \forall\, x.\ y < \text{real-of-preal } x$
⟨*proof*⟩

63.9 Completeness of Positive Reals

Supremum property for the set of positive reals

Let P be a non-empty set of positive reals, with an upper bound y. Then P has a least upper bound (written S).

FIXME: Can the premise be weakened to $\forall\, x \in P.\ x \leq y$?

lemma *posreal-complete*:
assumes *positive-P*: $\forall\, x \in P.\ (0::real) < x$
and *not-empty-P*: $\exists\, x.\ x \in P$
and *upper-bound-Ex*: $\exists\, y.\ \forall\, x \in P.\ x<y$
shows $\exists\, S.\ \forall\, y.\ (\exists\, x \in P.\ y < x) = (y < S)$
⟨*proof*⟩

Completeness

lemma *reals-complete*:
fixes S :: *real set*
assumes *notempty-S*: $\exists\, X.\ X \in S$
and *exists-Ub*: *bdd-above S*

shows $\exists x.\ (\forall s{\in}S.\ s \leq x) \wedge (\forall y.\ (\forall s{\in}S.\ s \leq y) \longrightarrow x \leq y)$
⟨*proof*⟩

63.10 The Archimedean Property of the Reals

theorem *reals-Archimedean*:
fixes x :: *real*
assumes *x-pos*: $0 < x$
shows $\exists n.$ *inverse* (*of-nat* (*Suc* n)) $< x$
⟨*proof*⟩

There must be other proofs, e.g. *Suc* of the largest integer in the cut representing x.

lemma *reals-Archimedean2*: $\exists n.$ (x::*real*) $<$ *of-nat* (n::*nat*)
⟨*proof*⟩

instance *real* :: *archimedean-field*
⟨*proof*⟩

end

64 Quicksort with function package

theory *Quicksort*
imports ~~/*src*/*HOL*/*Library*/*Multiset*
begin

context *linorder*
begin

fun *quicksort* :: 'a *list* ⇒ 'a *list* **where**
  quicksort [] = []
| *quicksort* (*x*#*xs*) = *quicksort* [*y*←*xs*. ¬ *x*≤*y*] @ [*x*] @ *quicksort* [*y*←*xs*. *x*≤*y*]

lemma [*code*]:
  quicksort [] = []
  quicksort (*x*#*xs*) = *quicksort* [*y*←*xs*. *y*<*x*] @ [*x*] @ *quicksort* [*y*←*xs*. *x*≤*y*]
  ⟨*proof*⟩

lemma *quicksort-permutes* [*simp*]:
  mset (*quicksort xs*) = *mset xs*
  ⟨*proof*⟩

lemma *set-quicksort* [*simp*]: *set* (*quicksort xs*) = *set xs*
⟨*proof*⟩

lemma *sorted-quicksort*: *sorted* (*quicksort xs*)
  ⟨*proof*⟩

theorem *sort-quicksort*:
sort = *quicksort*
⟨*proof*⟩

end

end

65 A Formulation of the Birthday Paradox

theory *Birthday-Paradox*
imports *Main* ~~/*src*/*HOL*/*Binomial* ~~/*src*/*HOL*/*Library*/*FuncSet*
begin

66 Cardinality

lemma *card-product-dependent*:
assumes *finite S*
assumes ∀ *x* ∈ *S*. *finite* (*T x*)
shows *card* {(*x*, *y*). *x* ∈ *S* ∧ *y* ∈ *T x*} = (∑ *x* ∈ *S*. *card* (*T x*))
⟨*proof*⟩

lemma *card-extensional-funcset-inj-on*:
assumes *finite S finite T card S* ≤ *card T*
shows *card* {*f* ∈ *extensional-funcset S T*. *inj-on f S*} = *fact* (*card T*) *div* (*fact* (*card T* − *card S*))
⟨*proof*⟩

lemma *card-extensional-funcset-not-inj-on*:
assumes *finite S finite T card S* ≤ *card T*
shows *card* {*f* ∈ *extensional-funcset S T*. ¬ *inj-on f S*} = (*card T*) ^ (*card S*) − (*fact* (*card T*)) *div* (*fact* (*card T* − *card S*))
⟨*proof*⟩

lemma *prod-upto-nat-unfold*:
prod f {*m*..(*n*::*nat*)} = (*if n* < *m then 1 else* (*if n* = *0 then f 0 else f n* ∗ *prod f* {*m*..(*n* − *1*)}))
⟨*proof*⟩

67 Birthday paradox

lemma *birthday-paradox*:
assumes *card S* = *23 card T* = *365*
shows *2* ∗ *card* {*f* ∈ *extensional-funcset S T*. ¬ *inj-on f S*} ≥ *card* (*extensional-funcset S T*)
⟨*proof*⟩

end

68 Examples for the list comprehension to set comprehension simproc

theory *List-to-Set-Comprehension-Examples*
imports *Main*
begin

Examples that show and test the simproc that rewrites list comprehensions applied to List.set to set comprehensions.

68.1 Some own examples for set (case ..) simpproc

lemma *set* $[(x,\ xs).\ x\ \#\ xs <-\ as] = \{(x,\ xs).\ x\ \#\ xs \in set\ as\}$
⟨*proof*⟩

lemma *set* $[(x,\ y,\ ys).\ x\ \#\ y\ \#\ ys <-\ as] = \{(x,\ y,\ ys).\ x\ \#\ y\ \#\ ys \in set\ as\}$
⟨*proof*⟩

lemma *set* $[(x,\ y,\ z,\ zs).\ x\ \#\ y\ \#\ z\ \#\ zs <-\ as] = \{(x,\ y,\ z,\ zs).\ x\ \#\ y\ \#\ z\ \#\ zs \in set\ as\}$
⟨*proof*⟩

lemma *set* $[(zs,\ x,\ z,\ y).\ x\ \#\ (y,\ z)\ \#\ zs <-\ as] = \{(zs,\ x,\ z,\ y).\ x\ \#\ (y,\ z)\ \#\ zs \in set\ as\}$
⟨*proof*⟩

lemma *set* $[(x,\ y).\ x\ \#\ y <-\ zs,\ x = x'] = \{(x,\ y).\ x\ \#\ y \in set\ zs\ \&\ x = x'\}$
⟨*proof*⟩

68.2 Existing examples from the List theory

lemma *set* $[(x,y,z).\ b] = \{(x',\ y',\ z').\ x = x'\ \&\ y = y'\ \&\ z = z'\ \&\ b\}$
⟨*proof*⟩

lemma *set* $[(x,y,z).\ x{\leftarrow}xs] = \{(x,\ y',\ z').\ x \in set\ xs\ \&\ y' = y\ \&\ z = z'\}$
⟨*proof*⟩

lemma *set* $[e\ x\ y.\ x{\leftarrow}xs,\ y{\leftarrow}ys] = \{z.\ \exists\ x\ y.\ z = e\ x\ y\ \&\ x \in set\ xs\ \&\ y \in set\ ys\}$
⟨*proof*⟩

lemma *set* $[(x,y,z).\ x<a,\ x>b] = \{(x',\ y',\ z').\ x' = x\ \&\ y' = y\ \&\ z = z'\ \&\ x < a\ \&\ x>b\}$
⟨*proof*⟩

lemma *set* $[(x,y,z).\ x{\leftarrow}xs,\ x>b] = \{(x',\ y',\ z').\ y = y'\ \&\ z = z'\ \&\ x' \in set\ xs\ \&\ x' > b\}$
⟨*proof*⟩

lemma *set* [(*x*,*y*,*z*). *x*<*a*, *x*←*xs*] = {(*x'*, *y'*, *z'*). *y* = *y'* & *z* = *z'* & *x'* ∈ *set xs* & *x* < *a*}
⟨*proof*⟩

lemma *set* [(*x*,*y*). *Cons True x* ← *xs*] = {(*x*, *y'*). *y* = *y'* & *Cons True x* ∈ *set xs*}
⟨*proof*⟩

lemma *set* [(*x*,*y*,*z*). *Cons x* [] ← *xs*] = {(*x*, *y'*, *z'*). *y* = *y'* & *z* = *z'* & *Cons x* [] ∈ *set xs*}
⟨*proof*⟩

lemma *set* [(*x*,*y*,*z*). *x*<*a*, *x*>*b*, *x*=*d*] = {(*x'*, *y'*, *z'*). *x* = *x'* & *y* = *y'* & *z* = *z'* & *x* < *a* & *x* > *b* & *x* = *d*}
⟨*proof*⟩

lemma *set* [(*x*,*y*,*z*). *x*<*a*, *x*>*b*, *y*←*ys*] = {(*x'*, *y*, *z'*). *x* = *x'* & *y* ∈ *set ys* & *z* = *z'* & *x* < *a* & *x* > *b*}
⟨*proof*⟩

lemma *set* [(*x*,*y*,*z*). *x*<*a*, *x*←*xs*,*y*>*b*] = {(*x'*,*y'*,*z'*). *x'* ∈ *set xs* & *y* = *y'* & *z* = *z'* & *x* < *a* & *y* > *b*}
⟨*proof*⟩

lemma *set* [(*x*,*y*,*z*). *x*<*a*, *x*←*xs*, *y*←*ys*] = {(*x'*, *y'*, *z'*). *z* = *z'* & *x* < *a* & *x'* ∈ *set xs* & *y'* ∈ *set ys*}
⟨*proof*⟩

lemma *set* [(*x*,*y*,*z*). *x*←*xs*, *x*>*b*, *y*<*a*] = {(*x'*, *y'*, *z'*). *y* = *y'* & *z* = *z'* & *x'* ∈ *set xs* & *x'* > *b* & *y* < *a*}
⟨*proof*⟩

lemma *set* [(*x*,*y*,*z*). *x*←*xs*, *x*>*b*, *y*←*ys*] = {(*x'*, *y'*, *z'*). *z* = *z'* & *x'* ∈ *set xs* & *x'* > *b* & *y'* ∈ *set ys*}
⟨*proof*⟩

lemma *set* [(*x*,*y*,*z*). *x*←*xs*, *y*←*ys*,*y*>*x*] = {(*x'*, *y'*, *z'*). *z* = *z'* & *x'* ∈ *set xs* & *y'* ∈ *set ys* & *y'* > *x'*}
⟨*proof*⟩

lemma *set* [(*x*,*y*,*z*). *x*←*xs*, *y*←*ys*,*z*←*zs*] = {(*x'*, *y'*, *z'*). *x'* ∈ *set xs* & *y'* ∈ *set ys* & *z'* ∈ *set zs*}
⟨*proof*⟩

end

69 Finite sequences

theory *Seq*
imports *Main*

begin

datatype *'a seq = Empty | Seq 'a 'a seq*

fun *conc* :: *'a seq ⇒ 'a seq ⇒ 'a seq*
where
  conc Empty ys = ys
| *conc (Seq x xs) ys = Seq x (conc xs ys)*

fun *reverse* :: *'a seq ⇒ 'a seq*
where
  reverse Empty = Empty
| *reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)*

lemma *conc-empty*: *conc xs Empty = xs*
  ⟨*proof*⟩

lemma *conc-assoc*: *conc (conc xs ys) zs = conc xs (conc ys zs)*
  ⟨*proof*⟩

lemma *reverse-conc*: *reverse (conc xs ys) = conc (reverse ys) (reverse xs)*
  ⟨*proof*⟩

lemma *reverse-reverse*: *reverse (reverse xs) = xs*
  ⟨*proof*⟩

end

70 Testing of arithmetic simprocs

theory *Simproc-Tests*
imports *Main*
begin

This theory tests the various simprocs defined in `~~/src/HOL/Nat.thy` and `~~/src/HOL/Numeral_Simprocs.thy`. Many of the tests are derived from commented-out code originally found in `~~/src/HOL/Tools/numeral_simprocs.ML`.

70.1 ML bindings

⟨*ML*⟩

70.2 Cancellation simprocs from *Nat.thy*

notepad begin
  ⟨*proof*⟩
end

schematic-goal ⋀(*y*::*?'b*::*size*). *size* (*?x*::*?'a*::*size*) ≤ *size y* + *size ?x*

⟨*proof*⟩

70.3 Abelian group cancellation simprocs

notepad begin
⟨*proof*⟩
end

70.4 *int-combine-numerals*

notepad begin
⟨*proof*⟩
end

70.5 *inteq-cancel-numerals*

notepad begin
⟨*proof*⟩
end

70.6 *intless-cancel-numerals*

notepad begin
⟨*proof*⟩
end

70.7 *ring-eq-cancel-numeral-factor*

notepad begin
⟨*proof*⟩
end

70.8 *int-div-cancel-numeral-factors*

notepad begin
⟨*proof*⟩
end

70.9 *ring-less-cancel-numeral-factor*

notepad begin
⟨*proof*⟩
end

70.10 *ring-le-cancel-numeral-factor*

notepad begin
⟨*proof*⟩
end

70.11 *divide-cancel-numeral-factor*

notepad begin
⟨*proof*⟩
end

70.12 *ring-eq-cancel-factor*

notepad begin
⟨*proof*⟩
end

70.13 *int-div-cancel-factor*

notepad begin
⟨*proof*⟩
end

lemma shows $a*(b*c)/(y*z) = d*(b::'a::linordered\text{-}field)*(x*a)/z$
⟨*proof*⟩

70.14 *divide-cancel-factor*

notepad begin
⟨*proof*⟩
end

lemma
fixes $a\ b\ c\ d\ x\ y\ z :: 'a::linordered\text{-}field$
shows $a*(b*c)/(y*z) = d*(b)*(x*a)/z$
⟨*proof*⟩

70.15 *linordered-ring-less-cancel-factor*

notepad begin
⟨*proof*⟩
end

70.16 *linordered-ring-le-cancel-factor*

notepad begin
⟨*proof*⟩
end

70.17 *field-combine-numerals*

notepad begin
⟨*proof*⟩
end

lemma

fixes x :: $'a$::{*linordered-field*}
shows $2/3 * x + x / 3 = uu$
⟨*proof*⟩

70.18 *nat-combine-numerals*

notepad begin
⟨*proof*⟩
end

70.19 *nateq-cancel-numerals*

notepad begin
⟨*proof*⟩
end

70.20 *natless-cancel-numerals*

notepad begin
⟨*proof*⟩
end

70.21 *natle-cancel-numerals*

notepad begin
⟨*proof*⟩
end

70.22 *natdiff-cancel-numerals*

notepad begin
⟨*proof*⟩
end

70.23 Factor-cancellation simprocs for type *nat*

nat-eq-cancel-factor, *nat-less-cancel-factor*, *nat-le-cancel-factor*, *nat-divide-cancel-factor*, and *nat-dvd-cancel-factor*.

notepad begin
⟨*proof*⟩
end

70.24 Numeral-cancellation simprocs for type *nat*

notepad begin
⟨*proof*⟩
end

70.25 Integer numeral div/mod simprocs

notepad begin
⟨*proof*⟩
end

end
theory *Executable-Relation*
imports *Main*
begin

70.26 A dedicated type for relations

70.26.1 Definition of the dedicated type for relations

typedef *'a rel = UNIV :: (('a * 'a) set) set*
morphisms *set-of-rel rel-of-set* ⟨*proof*⟩

setup-lifting *type-definition-rel*

lift-definition *Rel :: 'a set => ('a * 'a) set => 'a rel* **is** *λ X R. Id-on X Un R*
⟨*proof*⟩

70.26.2 Constant definitions on relations

hide-const (**open**) *converse relcomp rtrancl Image*

lift-definition *member :: 'a * 'a => 'a rel => bool* **is** *Set.member* ⟨*proof*⟩

lift-definition *converse :: 'a rel => 'a rel* **is** *Relation.converse* ⟨*proof*⟩

lift-definition *union :: 'a rel => 'a rel => 'a rel* **is** *Set.union* ⟨*proof*⟩

lift-definition *relcomp :: 'a rel => 'a rel => 'a rel* **is** *Relation.relcomp* ⟨*proof*⟩

lift-definition *rtrancl :: 'a rel => 'a rel* **is** *Transitive-Closure.rtrancl* ⟨*proof*⟩

lift-definition *Image :: 'a rel => 'a set => 'a set* **is** *Relation.Image* ⟨*proof*⟩

70.26.3 Code generation

code-datatype *Rel*

lemma [*code*]:
member (x, y) (Rel X R) = ((x = y ∧ x : X) ∨ (x, y) : R)
⟨*proof*⟩

lemma [*code*]:
converse (Rel X R) = Rel X (R^−1)
⟨*proof*⟩

lemma [*code*]:
union (*Rel X R*) (*Rel Y S*) = *Rel* (*X Un Y*) (*R Un S*)
⟨*proof*⟩

lemma [*code*]:
relcomp (*Rel X R*) (*Rel Y S*) = *Rel* (*X Int Y*) (*Set.filter* (%(*x*, *y*). *y* : *Y*) *R Un* (*Set.filter* (%(*x*, *y*). *x* : *X*) *S Un R O S*))
⟨*proof*⟩

lemma [*code*]:
rtrancl (*Rel X R*) = *Rel UNIV* (*R*^+)
⟨*proof*⟩

lemma [*code*]:
Image (*Rel X R*) *S* = (*X Int S*) *Un* (*R `` S*)
⟨*proof*⟩

quickcheck-generator *rel constructors*: *Rel*

lemma
member (*x*, (*y* :: *nat*)) (*rtrancl* (*union R S*)) ⟹ *member* (*x*, *y*) (*union* (*rtrancl R*) (*rtrancl S*))
quickcheck[*exhaustive*, *expect* = *counterexample*]
⟨*proof*⟩

end

71 A generic phantom type

theory *Phantom-Type*
imports *Main*
begin

datatype ('*a*, '*b*) *phantom* = *phantom* (*of-phantom*: '*b*)

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

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

syntax -*Phantom* :: *type* ⇒ *logic* ((*1Phantom*/(*1* '(-'))))
translations
Phantom('*t*) => *CONST phantom* :: - ⇒ ('*t*, -) *phantom*

⟨*ML*⟩

lemma *of-phantom-inject* [*simp*]:

of-phantom x = of-phantom y ⟷ x = y
⟨*proof*⟩

end

72 Cardinality of types

theory *Cardinality*
imports *Phantom-Type*
begin

72.1 Preliminary lemmas

lemma (**in** *type-definition*) *univ*:
UNIV = Abs ' A
⟨*proof*⟩

lemma (**in** *type-definition*) *card*: *card* (*UNIV* :: *'b set*) = *card A*
⟨*proof*⟩

lemma *finite-range-Some*: *finite* (*range* (*Some* :: *'a* ⇒ *'a option*)) = *finite* (*UNIV* :: *'a set*)
⟨*proof*⟩

lemma *infinite-literal*: ¬ *finite* (*UNIV* :: *String.literal set*)
⟨*proof*⟩

72.2 Cardinalities of types

syntax *-type-card* :: *type* => *nat* ((*1CARD/(1'(-')*)))

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

⟨*ML*⟩

lemma *card-prod* [*simp*]: *CARD('a × 'b) = CARD('a) * CARD('b)*
⟨*proof*⟩

lemma *card-UNIV-sum*: *CARD('a + 'b) = (if CARD('a) ≠ 0 ∧ CARD('b) ≠ 0 then CARD('a) + CARD('b) else 0)*
⟨*proof*⟩

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

lemma *card-UNIV-option*: *CARD('a option) = (if CARD('a) = 0 then 0 else CARD('a) + 1)*
⟨*proof*⟩

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

lemma *card-UNIV-set*: $CARD('a\ set) = (if\ CARD('a) = 0\ then\ 0\ else\ 2\ \hat{}\ CARD('a))$
⟨*proof*⟩

lemma *card-set* [*simp*]: $CARD('a\ set) = 2\ \hat{}\ CARD('a::finite)$
⟨*proof*⟩

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

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

corollary *finite-UNIV-fun*:
$finite\ (UNIV :: ('a \Rightarrow 'b)\ set) \longleftrightarrow$
$finite\ (UNIV :: 'a\ set) \wedge finite\ (UNIV :: 'b\ set) \vee CARD('b) = 1$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

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

72.3 Classes with at least 1 and 2

Class finite already captures "at least 1"

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

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

Class for cardinality "at least 2"

class *card2* = *finite* +
assumes *two-le-card*: $2 \leq CARD('a)$

lemma *one-less-card*: $Suc\ 0 < CARD('a::card2)$
⟨*proof*⟩

lemma *one-less-int-card*: $1 < int\ CARD('a::card2)$
⟨*proof*⟩

72.4 A type class for deciding finiteness of types

type-synonym *'a finite-UNIV* = $('a, bool)\ phantom$

class *finite-UNIV* =

fixes *finite-UNIV* :: ('*a*, *bool*) *phantom*
assumes *finite-UNIV*: *finite-UNIV* = *Phantom*('*a*) (*finite* (*UNIV* :: '*a set*))

lemma *finite-UNIV-code* [*code-unfold*]:
finite (*UNIV* :: '*a* :: *finite-UNIV set*)
⟷ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*)
⟨*proof*⟩

72.5 A type class for computing the cardinality of types

definition *is-list-UNIV* :: '*a list* ⇒ *bool*
where *is-list-UNIV xs* = (*let c* = *CARD*('*a*) *in if c* = *0 then False else size* (*remdups xs*) = *c*)

lemma *is-list-UNIV-iff*: *is-list-UNIV xs* ⟷ *set xs* = *UNIV*
⟨*proof*⟩

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

class *card-UNIV* = *finite-UNIV* +
fixes *card-UNIV* :: '*a card-UNIV*
assumes *card-UNIV*: *card-UNIV* = *Phantom*('*a*) *CARD*('*a*)

72.6 Instantiations for *card-UNIV*

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

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

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

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

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

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

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

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

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

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

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

instantiation *sum* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* + '*b*)
(*let ca* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*);

```
      cb = of-phantom (card-UNIV :: 'b card-UNIV)
   in if ca ≠ 0 ∧ cb ≠ 0 then ca + cb else 0)
instance ⟨proof⟩
end

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

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

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

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

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

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

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

end

lemma *UNIV-finite-1*: *UNIV* = *set* [*finite-1*.a_1]
⟨*proof*⟩

lemma *UNIV-finite-2*: *UNIV* = *set* [*finite-2*.a_1, *finite-2*.a_2]
⟨*proof*⟩

lemma *UNIV-finite-3*: *UNIV* = *set* [*finite-3*.a_1, *finite-3*.a_2, *finite-3*.a_3]
⟨*proof*⟩

lemma *UNIV-finite-4*: *UNIV* = *set* [*finite-4*.a_1, *finite-4*.a_2, *finite-4*.a_3, *finite-4*.a_4]
⟨*proof*⟩

lemma *UNIV-finite-5*:
UNIV = *set* [*finite-5*.a_1, *finite-5*.a_2, *finite-5*.a_3, *finite-5*.a_4, *finite-5*.a_5]
⟨*proof*⟩

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

instantiation *Enum.finite-2* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-2*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-2*) *2*
instance
⟨*proof*⟩
end

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

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

instantiation *Enum.finite-5* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-5*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-5*) *5*

instance
⟨*proof*⟩
end

72.7 Code setup for sets

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

context
begin

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

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

lemma *card-UNIV'-code* [*code*]:
card-UNIV' = *card-UNIV*
⟨*proof*⟩

end

lemma *card-Compl*:
finite A ⟹ *card* (− *A*) = *card* (*UNIV* :: '*a set*) − *card* (*A* :: '*a set*)
⟨*proof*⟩

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

qualified definition *finite'* :: '*a set* ⇒ *bool*
where [*simp*, *code del*, *code-abbrev*]: *finite'* = *finite*

lemma *finite'-code* [*code*]:
finite' (*set xs*) ⟷ *True*
finite' (*List.coset xs*) ⟷ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*)
⟨*proof*⟩

end

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

qualified definition *card'* :: *'a set* ⇒ *nat*
where [*simp*, *code del*, *code-abbrev*]: *card'* = *card*

lemma *card'-code* [*code*]:
card' (*set xs*) = *length* (*remdups xs*)
card' (*List.coset xs*) = *of-phantom* (*card-UNIV* :: *'a card-UNIV*) − *length* (*remdups xs*)
⟨*proof*⟩ **definition** *subset'* :: *'a set* ⇒ *'a set* ⇒ *bool*
where [*simp*, *code del*, *code-abbrev*]: *subset'* = *op* ⊆

lemma *subset'-code* [*code*]:
subset' *A* (*List.coset ys*) ⟷ (∀ *y* ∈ *set ys*. *y* ∉ *A*)
subset' (*set ys*) *B* ⟷ (∀ *y* ∈ *set ys*. *y* ∈ *B*)
subset' (*List.coset xs*) (*set ys*) ⟷ (*let n* = *CARD*(*'a*) *in n* > *0* ∧ *card*(*set* (*xs* @ *ys*)) = *n*)
⟨*proof*⟩ **definition** *eq-set* :: *'a set* ⇒ *'a set* ⇒ *bool*
where [*simp*, *code del*, *code-abbrev*]: *eq-set* = *op* =

lemma *eq-set-code* [*code*]:
fixes *ys*
defines *rhs* ≡
let n = *CARD*(*'a*)
in if n = *0 then False else*
let xs' = *remdups xs*; *ys'* = *remdups ys*
in length xs' + *length ys'* = *n* ∧ (∀ *x* ∈ *set xs'*. *x* ∉ *set ys'*) ∧ (∀ *y* ∈ *set ys'*. *y* ∉ *set xs'*)
shows *eq-set* (*List.coset xs*) (*set ys*) ⟷ *rhs*
and *eq-set* (*set ys*) (*List.coset xs*) ⟷ *rhs*
and *eq-set* (*set xs*) (*set ys*) ⟷ (∀ *x* ∈ *set xs*. *x* ∈ *set ys*) ∧ (∀ *y* ∈ *set ys*. *y* ∈ *set xs*)
and *eq-set* (*List.coset xs*) (*List.coset ys*) ⟷ (∀ *x* ∈ *set xs*. *x* ∈ *set ys*) ∧ (∀ *y* ∈ *set ys*. *y* ∈ *set xs*)
⟨*proof*⟩

end

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

lemma *card-coset-error* [*code*]:
card (*List.coset xs*) =
Code.abort (*STR ''card* (*List.coset* -) *requires type class instance card-UNIV''*)
(λ-. *card* (*List.coset xs*))
⟨*proof*⟩

lemma *coset-subseteq-set-code* [*code*]:
List.coset xs ⊆ *set ys* ⟷

(*if* $xs = [] \wedge ys = []$ *then False*
else Code.abort
(*STR ''subset-eq (List.coset -) (List.set -) requires type class instance card-UNIV ''*)
(λ-. *List.coset xs* $\subseteq$ *set ys*))
⟨*proof*⟩

notepad begin — test code setup
⟨*proof*⟩
end

end

73 Almost everywhere constant functions

theory *FinFun*
imports *Cardinality*
begin

This theory defines functions which are constant except for finitely many points (FinFun) and introduces a type finfin along with a number of operators for them. The code generator is set up such that such functions can be represented as data in the generated code and all operators are executable.

For details, see Formalising FinFuns - Generating Code for Functions as Data by A. Lochbihler in TPHOLs 2009.

73.1 The *map-default* operation

definition *map-default* :: $'b \Rightarrow ('a \rightharpoonup 'b) \Rightarrow 'a \Rightarrow 'b$
where *map-default b f a* $\equiv$ *case f a of None* $\Rightarrow$ *b* | *Some b'* $\Rightarrow$ *b'*

lemma *map-default-delete* [*simp*]:
map-default b (*f*(*a* := *None*)) = (*map-default b f*)(*a* := *b*)
⟨*proof*⟩

lemma *map-default-insert*:
map-default b (*f*(*a* $\mapsto$ *b'*)) = (*map-default b f*)(*a* := *b'*)
⟨*proof*⟩

lemma *map-default-empty* [*simp*]: *map-default b empty* = (λ*a*. *b*)
⟨*proof*⟩

lemma *map-default-inject*:
fixes *g g'* :: $'a \rightharpoonup 'b$
assumes *infin-eq*: $\neg$ *finite* (*UNIV* :: *'a set*) $\vee$ *b* = *b'*
and *fin*: *finite* (*dom g*) **and** *b*: *b* $\notin$ *ran g*
and *fin'*: *finite* (*dom g'*) **and** *b'*: *b'* $\notin$ *ran g'*
and *eq'*: *map-default b g* = *map-default b' g'*
shows *b* = *b' g* = *g'*

⟨*proof*⟩

73.2 The finfun type

definition *finfun* = {*f*::'*a*⇒'*b*. ∃ *b*. *finite* {*a*. *f a* ≠ *b*}}

typedef ('*a*,'*b*) *finfun* ((- ⇒*f* /-) [*22*, *21*] *21*) = *finfun* :: ('*a* => '*b*) *set*
  morphisms *finfun-apply Abs-finfun*
⟨*proof*⟩

type-notation *finfun* ((- ⇒*f* /-) [*22*, *21*] *21*)

setup-lifting *type-definition-finfun*

lemma *fun-upd-finfun*: *y*(*a* := *b*) ∈ *finfun* ⟷ *y* ∈ *finfun*
⟨*proof*⟩

lemma *const-finfun*: (λ*x*. *a*) ∈ *finfun*
⟨*proof*⟩

lemma *finfun-left-compose*:
  assumes *y* ∈ *finfun*
  shows *g* ∘ *y* ∈ *finfun*
⟨*proof*⟩

lemma assumes *y* ∈ *finfun*
  shows *fst-finfun*: *fst* ∘ *y* ∈ *finfun*
  and *snd-finfun*: *snd* ∘ *y* ∈ *finfun*
⟨*proof*⟩

lemma *map-of-finfun*: *map-of xs* ∈ *finfun*
⟨*proof*⟩

lemma *Diag-finfun*: (λ*x*. (*f x*, *g x*)) ∈ *finfun* ⟷ *f* ∈ *finfun* ∧ *g* ∈ *finfun*
⟨*proof*⟩

lemma *finfun-right-compose*:
  assumes *g*: *g* ∈ *finfun* **and** *inj*: *inj f*
  shows *g o f* ∈ *finfun*
⟨*proof*⟩

lemma *finfun-curry*:
  assumes *fin*: *f* ∈ *finfun*
  shows *curry f* ∈ *finfun curry f a* ∈ *finfun*
⟨*proof*⟩

bundle *finfun*
begin

lemmas [*simp*] =
fst-finfun snd-finfun Abs-finfun-inverse
finfun-apply-inverse Abs-finfun-inject finfun-apply-inject
Diag-finfun finfun-curry
lemmas [*iff*] =
const-finfun fun-upd-finfun finfun-apply map-of-finfun
lemmas [*intro*] =
finfun-left-compose fst-finfun snd-finfun

end

lemma *Abs-finfun-inject-finite*:
fixes $x\ y :: 'a \Rightarrow 'b$
assumes *fin*: *finite* (*UNIV* :: $'a$ *set*)
shows *Abs-finfun* x = *Abs-finfun* $y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *Abs-finfun-inject-finite-class*:
fixes $x\ y :: ('a :: finite) \Rightarrow 'b$
shows *Abs-finfun* x = *Abs-finfun* $y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *Abs-finfun-inj-finite*:
assumes *fin*: *finite* (*UNIV* :: $'a$ *set*)
shows *inj* (*Abs-finfun* :: ($'a \Rightarrow 'b$) $\Rightarrow$ $'a \Rightarrow_f 'b$)
⟨*proof*⟩

lemma *Abs-finfun-inverse-finite*:
fixes $x :: 'a \Rightarrow 'b$
assumes *fin*: *finite* (*UNIV* :: $'a$ *set*)
shows *finfun-apply* (*Abs-finfun* x) = x
including *finfun*
⟨*proof*⟩

lemma *Abs-finfun-inverse-finite-class*:
fixes $x :: ('a :: finite) \Rightarrow 'b$
shows *finfun-apply* (*Abs-finfun* x) = x
⟨*proof*⟩

lemma *finfun-eq-finite-UNIV*: *finite* (*UNIV* :: $'a$ *set*) $\Longrightarrow$ (*finfun* :: ($'a \Rightarrow 'b$) *set*)
= *UNIV*
⟨*proof*⟩

lemma *finfun-finite-UNIV-class*: *finfun* = (*UNIV* :: ($'a$ :: *finite* $\Rightarrow$ $'b$) *set*)
⟨*proof*⟩

lemma *map-default-in-finfun*:
assumes *fin*: *finite* (*dom* f)
shows *map-default* b $f \in$ *finfun*

⟨*proof*⟩

lemma *finfun-cases-map-default*:
obtains *b g* **where** *f* = *Abs-finfun* (*map-default b g*) *finite* (*dom g*) *b* ∉ *ran g*
⟨*proof*⟩

73.3 Kernel functions for type *'a* ⇒*f* *'b*

lift-definition *finfun-const* :: *'b* ⇒ *'a* ⇒*f* *'b* (*K*$/ - [*0*] *1*)
is λ *b x*. *b* ⟨*proof*⟩

lift-definition *finfun-update* :: *'a* ⇒*f* *'b* ⇒ *'a* ⇒ *'b* ⇒ *'a* ⇒*f* *'b* (-'(- $:= -') [*1000*,*0*,*0*] *1000*) **is** *fun-upd*
⟨*proof*⟩

lemma *finfun-update-twist*: *a* ≠ *a'* ⟹ *f*(*a* $:= *b*)(*a'* $:= *b'*) = *f*(*a'* $:= *b'*)(*a* $:= *b*)
⟨*proof*⟩

lemma *finfun-update-twice* [*simp*]:
f(*a* $:= *b*)(*a* $:= *b'*) = *f*(*a* $:= *b'*)
⟨*proof*⟩

lemma *finfun-update-const-same*: (*K*$ *b*)(*a* $:= *b*) = (*K*$ *b*)
⟨*proof*⟩

73.4 Code generator setup

definition *finfun-update-code* :: *'a* ⇒*f* *'b* ⇒ *'a* ⇒ *'b* ⇒ *'a* ⇒*f* *'b*
where [*simp*, *code del*]: *finfun-update-code* = *finfun-update*

code-datatype *finfun-const finfun-update-code*

lemma *finfun-update-const-code* [*code*]:
(*K*$ *b*)(*a* $:= *b'*) = (*if b* = *b' then* (*K*$ *b*) *else finfun-update-code* (*K*$ *b*) *a b'*)
⟨*proof*⟩

lemma *finfun-update-update-code* [*code*]:
(*finfun-update-code f a b*)(*a'* $:= *b'*) = (*if a* = *a' then f*(*a* $:= *b'*) *else finfun-update-code* (*f*(*a'* $:= *b'*)) *a b*)
⟨*proof*⟩

73.5 Setup for quickcheck

quickcheck-generator *finfun constructors*: *finfun-update-code*, *finfun-const* :: *'b* ⇒ *'a* ⇒*f* *'b*

73.6 *finfun-update* as instance of *comp-fun-commute*

interpretation *finfun-update*: *comp-fun-commute* λ*a f*. *f*(*a* :: *'a* $:= *b'*)

including *finfun*
⟨*proof*⟩

lemma *fold-finfun-update-finite-univ*:
assumes *fin*: *finite* (*UNIV* :: *'a set*)
shows *Finite-Set.fold* (λ*a f*. *f*(*a* $:= *b'*)) (*K*$ *b*) (*UNIV* :: *'a set*) = (*K*$ *b'*)
⟨*proof*⟩

73.7 Default value for FinFuns

definition *finfun-default-aux* :: (*'a* ⇒ *'b*) ⇒ *'b*
where [*code del*]: *finfun-default-aux f* = (*if finite* (*UNIV* :: *'a set*) *then undefined else THE b*. *finite* {*a*. *f a* ≠ *b*})

lemma *finfun-default-aux-infinite*:
fixes *f* :: *'a* ⇒ *'b*
assumes *infin*: ¬ *finite* (*UNIV* :: *'a set*)
and *fin*: *finite* {*a*. *f a* ≠ *b*}
shows *finfun-default-aux f* = *b*
⟨*proof*⟩

lemma *finite-finfun-default-aux*:
fixes *f* :: *'a* ⇒ *'b*
assumes *fin*: *f* ∈ *finfun*
shows *finite* {*a*. *f a* ≠ *finfun-default-aux f*}
⟨*proof*⟩

lemma *finfun-default-aux-update-const*:
fixes *f* :: *'a* ⇒ *'b*
assumes *fin*: *f* ∈ *finfun*
shows *finfun-default-aux* (*f*(*a* := *b*)) = *finfun-default-aux f*
⟨*proof*⟩

lift-definition *finfun-default* :: *'a* ⇒*f* *'b* ⇒ *'b*
is *finfun-default-aux* ⟨*proof*⟩

lemma *finite-finfun-default*: *finite* {*a*. *finfun-apply f a* ≠ *finfun-default f*}
⟨*proof*⟩

lemma *finfun-default-const*: *finfun-default* ((*K*$ *b*) :: *'a* ⇒*f* *'b*) = (*if finite* (*UNIV* :: *'a set*) *then undefined else b*)
⟨*proof*⟩

lemma *finfun-default-update-const*:
finfun-default (*f*(*a* $:= *b*)) = *finfun-default f*
⟨*proof*⟩

lemma *finfun-default-const-code* [*code*]:

finfun-default ((*K$* *c*) :: *'a* :: *card-UNIV* ⇒*f* *'b*) = (*if* *CARD*(*'a*) = *0* *then* *c* *else* *undefined*)
⟨*proof*⟩

lemma *finfun-default-update-code* [*code*]:
finfun-default (*finfun-update-code* *f* *a* *b*) = *finfun-default* *f*
⟨*proof*⟩

73.8 Recursion combinator and well-formedness conditions

definition *finfun-rec* :: (*'b* ⇒ *'c*) ⇒ (*'a* ⇒ *'b* ⇒ *'c* ⇒ *'c*) ⇒ (*'a* ⇒*f* *'b*) ⇒ *'c*
where [*code* *del*]:
finfun-rec *cnst* *upd* *f* ≡
let *b* = *finfun-default* *f*;
g = *THE* *g*. *f* = *Abs-finfun* (*map-default* *b* *g*) ∧ *finite* (*dom* *g*) ∧ *b* ∉ *ran* *g*
in *Finite-Set.fold* (λ*a*. *upd* *a* (*map-default* *b* *g* *a*)) (*cnst* *b*) (*dom* *g*)

locale *finfun-rec-wf-aux* =
fixes *cnst* :: *'b* ⇒ *'c*
and *upd* :: *'a* ⇒ *'b* ⇒ *'c* ⇒ *'c*
assumes *upd-const-same*: *upd* *a* *b* (*cnst* *b*) = *cnst* *b*
and *upd-commute*: *a* ≠ *a'* ⟹ *upd* *a* *b* (*upd* *a'* *b'* *c*) = *upd* *a'* *b'* (*upd* *a* *b* *c*)
and *upd-idemp*: *b* ≠ *b'* ⟹ *upd* *a* *b''* (*upd* *a* *b'* (*cnst* *b*)) = *upd* *a* *b''* (*cnst* *b*)
begin

lemma *upd-left-comm*: *comp-fun-commute* (λ*a*. *upd* *a* (*f* *a*))
⟨*proof*⟩

lemma *upd-upd-twice*: *upd* *a* *b''* (*upd* *a* *b'* (*cnst* *b*)) = *upd* *a* *b''* (*cnst* *b*)
⟨*proof*⟩

lemma *map-default-update-const*:
assumes *fin*: *finite* (*dom* *f*)
and *anf*: *a* ∉ *dom* *f*
and *fg*: *f* ⊆*m* *g*
shows *upd* *a* *d* (*Finite-Set.fold* (λ*a*. *upd* *a* (*map-default* *d* *g* *a*)) (*cnst* *d*) (*dom* *f*)) =
Finite-Set.fold (λ*a*. *upd* *a* (*map-default* *d* *g* *a*)) (*cnst* *d*) (*dom* *f*)
⟨*proof*⟩

lemma *map-default-update-twice*:
assumes *fin*: *finite* (*dom* *f*)
and *anf*: *a* ∉ *dom* *f*
and *fg*: *f* ⊆*m* *g*
shows *upd* *a* *d''* (*upd* *a* *d'* (*Finite-Set.fold* (λ*a*. *upd* *a* (*map-default* *d* *g* *a*)) (*cnst* *d*) (*dom* *f*))) =
upd *a* *d''* (*Finite-Set.fold* (λ*a*. *upd* *a* (*map-default* *d* *g* *a*)) (*cnst* *d*) (*dom* *f*))
⟨*proof*⟩

lemma *map-default-eq-id* [*simp*]: *map-default d* ((λ*a*. *Some* (*f a*)) |‘ {*a*. *f a* ≠ *d*}) = *f*
⟨*proof*⟩

lemma *finite-rec-cong1*:
  assumes *f*: *comp-fun-commute f* **and** *g*: *comp-fun-commute g*
  and *fin*: *finite A*
  and *eq*: ⋀*a*. *a* ∈ *A* ⟹ *f a* = *g a*
  shows *Finite-Set.fold f z A* = *Finite-Set.fold g z A*
⟨*proof*⟩

lemma *finfun-rec-upd* [*simp*]:
  finfun-rec cnst upd (*f*(*a*′ $:= *b*′)) = *upd a*′ *b*′ (*finfun-rec cnst upd f*)
  including *finfun*
⟨*proof*⟩

end

locale *finfun-rec-wf* = *finfun-rec-wf-aux* +
  assumes *const-update-all*:
  finite (*UNIV* :: ′*a set*) ⟹ *Finite-Set.fold* (λ*a*. *upd a b*′) (*cnst b*) (*UNIV* :: ′*a set*) = *cnst b*′
begin

lemma *finfun-rec-const* [*simp*]: *finfun-rec cnst upd* (*K*$ *c*) = *cnst c*
  including *finfun*
⟨*proof*⟩

end

73.9 Weak induction rule and case analysis for FinFuns

lemma *finfun-weak-induct* [*consumes 0*, *case-names const update*]:
  assumes *const*: ⋀*b*. *P* (*K*$ *b*)
  and *update*: ⋀*f a b*. *P f* ⟹ *P* (*f*(*a* $:= *b*))
  shows *P x*
  including *finfun*
⟨*proof*⟩

lemma *finfun-exhaust-disj*: (∃ *b*. *x* = *finfun-const b*) ∨ (∃ *f a b*. *x* = *finfun-update f a b*)
⟨*proof*⟩

lemma *finfun-exhaust*:
  obtains *b* **where** *x* = (*K*$ *b*)
    | *f a b* **where** *x* = *f*(*a* $:= *b*)
⟨*proof*⟩

lemma *finfun-rec-unique*:
fixes $f :: 'a \Rightarrow_f 'b \Rightarrow 'c$
assumes c: $\bigwedge c.\ f\ (K\$\ c) = cnst\ c$
and u: $\bigwedge g\ a\ b.\ f\ (g(a\ \$:=\ b)) = upd\ g\ a\ b\ (f\ g)$
and c': $\bigwedge c.\ f'\ (K\$\ c) = cnst\ c$
and u': $\bigwedge g\ a\ b.\ f'\ (g(a\ \$:=\ b)) = upd\ g\ a\ b\ (f'\ g)$
shows $f = f'$
⟨*proof*⟩

73.10 Function application

notation *finfun-apply* (**infixl** $ *999*)

interpretation *finfun-apply-aux*: *finfun-rec-wf-aux* $\lambda b.\ b\ \lambda a'\ b\ c.\ if\ (a = a')\ then\ b\ else\ c$
⟨*proof*⟩

interpretation *finfun-apply*: *finfun-rec-wf* $\lambda b.\ b\ \lambda a'\ b\ c.\ if\ (a = a')\ then\ b\ else\ c$
⟨*proof*⟩

lemma *finfun-apply-def*: $op\ \$ = (\lambda f\ a.\ finfun\text{-}rec\ (\lambda b.\ b)\ (\lambda a'\ b\ c.\ if\ (a = a')\ then\ b\ else\ c)\ f)$
⟨*proof*⟩

lemma *finfun-upd-apply*: $f(a\ \$:=\ b)\ \$\ a' = (if\ a = a'\ then\ b\ else\ f\ \$\ a')$
and *finfun-upd-apply-code* [*code*]: $(finfun\text{-}update\text{-}code\ f\ a\ b)\ \$\ a' = (if\ a = a'\ then\ b\ else\ f\ \$\ a')$
⟨*proof*⟩

lemma *finfun-const-apply* [*simp*, *code*]: $(K\$\ b)\ \$\ a = b$
⟨*proof*⟩

lemma *finfun-upd-apply-same* [*simp*]:
$f(a\ \$:=\ b)\ \$\ a = b$
⟨*proof*⟩

lemma *finfun-upd-apply-other* [*simp*]:
$a \neq a' \Longrightarrow f(a\ \$:=\ b)\ \$\ a' = f\ \$\ a'$
⟨*proof*⟩

lemma *finfun-ext*: $(\bigwedge a.\ f\ \$\ a = g\ \$\ a) \Longrightarrow f = g$
⟨*proof*⟩

lemma *expand-finfun-eq*: $(f = g) = (op\ \$\ f = op\ \$\ g)$
⟨*proof*⟩

lemma *finfun-upd-triv* [*simp*]: $f(x\ \$:=\ f\ \$\ x) = f$
⟨*proof*⟩

lemma *finfun-const-inject* [*simp*]: $(K\$\ b) = (K\$\ b') \equiv b = b'$
⟨*proof*⟩

lemma *finfun-const-eq-update*:
$((K\$\ b) = f(a\ \$:=\ b')) = (b = b' \wedge (\forall a'.\ a \neq a' \longrightarrow f\ \$\ a' = b))$
⟨*proof*⟩

73.11 Function composition

definition *finfun-comp* :: $('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow_f 'a \Rightarrow 'c \Rightarrow_f 'b$ (**infixr** $\circ\$$ *55*)
where [*code del*]: $g \circ\$ f = \textit{finfun-rec}\ (\lambda b.\ (K\$\ g\ b))\ (\lambda a\ b\ c.\ c(a\ \$:=\ g\ b))\ f$

notation (*ASCII*)
finfun-comp (**infixr** *o$* *55*)

interpretation *finfun-comp-aux*: *finfun-rec-wf-aux* $(\lambda b.\ (K\$\ g\ b))\ (\lambda a\ b\ c.\ c(a\ \$:=\ g\ b))$
⟨*proof*⟩

interpretation *finfun-comp*: *finfun-rec-wf* $(\lambda b.\ (K\$\ g\ b))\ (\lambda a\ b\ c.\ c(a\ \$:=\ g\ b))$
⟨*proof*⟩

lemma *finfun-comp-const* [*simp*, *code*]:
$g \circ\$\ (K\$\ c) = (K\$\ g\ c)$
⟨*proof*⟩

lemma *finfun-comp-update* [*simp*]: $g \circ\$\ (f(a\ \$:=\ b)) = (g \circ\$\ f)(a\ \$:=\ g\ b)$
and *finfun-comp-update-code* [*code*]:
$g \circ\$\ (\textit{finfun-update-code}\ f\ a\ b) = \textit{finfun-update-code}\ (g \circ\$\ f)\ a\ (g\ b)$
⟨*proof*⟩

lemma *finfun-comp-apply* [*simp*]:
$op\ \$\ (g \circ\$\ f) = g \circ op\ \$\ f$
⟨*proof*⟩

lemma *finfun-comp-comp-collapse* [*simp*]: $f \circ\$\ g \circ\$\ h = (f \circ g) \circ\$\ h$
⟨*proof*⟩

lemma *finfun-comp-const1* [*simp*]: $(\lambda x.\ c) \circ\$\ f = (K\$\ c)$
⟨*proof*⟩

lemma *finfun-comp-id1* [*simp*]: $(\lambda x.\ x) \circ\$\ f = f\ id \circ\$\ f = f$
⟨*proof*⟩

lemma *finfun-comp-conv-comp*: $g \circ\$\ f = \textit{Abs-finfun}\ (g \circ op\ \$\ f)$
including *finfun*
⟨*proof*⟩

definition *finfun-comp2* :: $'b \Rightarrow_f 'c \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow_f 'c$ (**infixr** $\$\circ$ *55*)

where [*code del*]: $g \$\circ f = \mathit{Abs\text{-}finfun}\ (\mathit{op}\ \$\ g \circ f)$

notation (*ASCII*)
finfun-comp2 (**infixr** $\$o$ *55*)

lemma *finfun-comp2-const* [*code*, *simp*]: *finfun-comp2* $(K\$\ c)\ f = (K\$\ c)$
including *finfun*
⟨*proof*⟩

lemma *finfun-comp2-update*:
assumes *inj*: *inj f*
shows *finfun-comp2* $(g(b\ \$:=\ c))\ f = (\mathit{if}\ b \in \mathit{range}\ f\ \mathit{then}\ (\mathit{finfun\text{-}comp2}\ g\ f)(\mathit{inv}\ f\ b\ \$:=\ c)\ \mathit{else}\ \mathit{finfun\text{-}comp2}\ g\ f)$
including *finfun*
⟨*proof*⟩

73.12 Universal quantification

definition *finfun-All-except* :: *'a list* ⇒ *'a* ⇒*f bool* ⇒ *bool*
where [*code del*]: $\mathit{finfun\text{-}All\text{-}except}\ A\ P \equiv \forall a.\ a \in \mathit{set}\ A \vee P\ \$\ a$

lemma *finfun-All-except-const*: $\mathit{finfun\text{-}All\text{-}except}\ A\ (K\$\ b) \longleftrightarrow b \vee \mathit{set}\ A = \mathit{UNIV}$
⟨*proof*⟩

lemma *finfun-All-except-const-finfun-UNIV-code* [*code*]:
$\mathit{finfun\text{-}All\text{-}except}\ A\ (K\$\ b) = (b \vee \mathit{is\text{-}list\text{-}UNIV}\ A)$
⟨*proof*⟩

lemma *finfun-All-except-update*:
$\mathit{finfun\text{-}All\text{-}except}\ A\ f(a\ \$:=\ b) = ((a \in \mathit{set}\ A \vee b) \wedge \mathit{finfun\text{-}All\text{-}except}\ (a\ \#\ A)\ f)$
⟨*proof*⟩

lemma *finfun-All-except-update-code* [*code*]:
fixes *a* :: *'a* :: *card-UNIV*
shows $\mathit{finfun\text{-}All\text{-}except}\ A\ (\mathit{finfun\text{-}update\text{-}code}\ f\ a\ b) = ((a \in \mathit{set}\ A \vee b) \wedge \mathit{finfun\text{-}All\text{-}except}\ (a\ \#\ A)\ f)$
⟨*proof*⟩

definition *finfun-All* :: *'a* ⇒*f bool* ⇒ *bool*
where $\mathit{finfun\text{-}All} = \mathit{finfun\text{-}All\text{-}except}\ []$

lemma *finfun-All-const* [*simp*]: $\mathit{finfun\text{-}All}\ (K\$\ b) = b$
⟨*proof*⟩

lemma *finfun-All-update*: $\mathit{finfun\text{-}All}\ f(a\ \$:=\ b) = (b \wedge \mathit{finfun\text{-}All\text{-}except}\ [a]\ f)$
⟨*proof*⟩

lemma *finfun-All-All*: $\mathit{finfun\text{-}All}\ P = \mathit{All}\ (\mathit{op}\ \$\ P)$

⟨*proof*⟩

definition *finfun-Ex* :: *'a* ⇒*f bool* ⇒ *bool*
where *finfun-Ex P* = *Not* (*finfun-All* (*Not* ∘$ *P*))

lemma *finfun-Ex-Ex*: *finfun-Ex P* = *Ex* (*op* $ *P*)
⟨*proof*⟩

lemma *finfun-Ex-const* [*simp*]: *finfun-Ex* (*K*$ *b*) = *b*
⟨*proof*⟩

73.13 A diagonal operator for FinFuns

definition *finfun-Diag* :: *'a* ⇒*f 'b* ⇒ *'a* ⇒*f 'c* ⇒ *'a* ⇒*f* (*'b* × *'c*) ((*1'*($-,/ -$*'*)) [*0*, *0*] *1000*)
where [*code del*]: ($*f*, *g*$) = *finfun-rec* (λ*b*. *Pair b* ∘$ *g*) (λ*a b c*. *c*(*a* $:= (*b*, *g* $ *a*))) *f*

interpretation *finfun-Diag-aux*: *finfun-rec-wf-aux* λ*b*. *Pair b* ∘$ *g* λ*a b c*. *c*(*a* $:= (*b*, *g* $ *a*))
⟨*proof*⟩

interpretation *finfun-Diag*: *finfun-rec-wf* λ*b*. *Pair b* ∘$ *g* λ*a b c*. *c*(*a* $:= (*b*, *g* $ *a*))
⟨*proof*⟩

lemma *finfun-Diag-const1*: ($*K*$ *b*, *g*$) = *Pair b* ∘$ *g*
⟨*proof*⟩

Do not use ($*K*$ *?b*, *?g*$) = *Pair ?b* ∘$ *?g* for the code generator because *Pair b* is injective, i.e. if *g* is free of redundant updates, there is no need to check for redundant updates as is done for *op* ∘$.

lemma *finfun-Diag-const-code* [*code*]:
($*K*$ *b*, *K*$ *c*$) = (*K*$ (*b*, *c*))
($*K*$ *b*, *finfun-update-code g a c*$) = *finfun-update-code* ($*K*$ *b*, *g*$) *a* (*b*, *c*)
⟨*proof*⟩

lemma *finfun-Diag-update1*: ($*f*(*a* $:= *b*), *g*$) = ($*f*, *g*$)(*a* $:= (*b*, *g* $ *a*))
and *finfun-Diag-update1-code* [*code*]: ($*finfun-update-code f a b*, *g*$) = ($*f*, *g*$)(*a* $:= (*b*, *g* $ *a*))
⟨*proof*⟩

lemma *finfun-Diag-const2*: ($*f*, *K*$ *c*$) = (λ*b*. (*b*, *c*)) ∘$ *f*
⟨*proof*⟩

lemma *finfun-Diag-update2*: ($*f*, *g*(*a* $:= *c*)$) = ($*f*, *g*$)(*a* $:= (*f* $ *a*, *c*))
⟨*proof*⟩

lemma *finfun-Diag-const-const* [*simp*]: (K *b*, *K*$ *c*$) = (*K*$ (*b*, *c*))
⟨*proof*⟩

lemma *finfun-Diag-const-update*:
(K *b*, *g*(*a* $:= *c*)$) = (K *b*, *g*$)(*a* $:= (*b*, *c*))
⟨*proof*⟩

lemma *finfun-Diag-update-const*:
($f(*a* $:= *b*), *K*$ *c*$) = ($f$, *K*$ *c*$)(*a* $:= (*b*, *c*))
⟨*proof*⟩

lemma *finfun-Diag-update-update*:
($f(*a* $:= *b*), *g*(*a*′ $:= *c*)$) = (*if a* = *a*′ *then* (f, *g*$)(*a* $:= (*b*, *c*)) *else* ($f$, *g*$)(*a* $:= (*b*, *g* $ *a*))(*a*′ $:= (*f* $ *a*′, *c*)))
⟨*proof*⟩

lemma *finfun-Diag-apply* [*simp*]: *op* $ (f, *g*$) = (λ*x*. (*f* $ *x*, *g* $ *x*))
⟨*proof*⟩

lemma *finfun-Diag-conv-Abs-finfun*:
(f, *g*$) = *Abs-finfun* ((λ*x*. (*f* $ *x*, *g* $ *x*)))
including *finfun*
⟨*proof*⟩

lemma *finfun-Diag-eq*: (f, *g*$) = ($f′$, *g*′$) ⟷ *f* = *f*′ ∧ *g* = *g*′
⟨*proof*⟩

definition *finfun-fst* :: ′*a* ⇒*f* (′*b* × ′*c*) ⇒ ′*a* ⇒*f* ′*b*
where [*code*]: *finfun-fst f* = *fst* ∘$ *f*

lemma *finfun-fst-const*: *finfun-fst* (*K*$ *bc*) = (*K*$ *fst bc*)
⟨*proof*⟩

lemma *finfun-fst-update*: *finfun-fst* (*f*(*a* $:= *bc*)) = (*finfun-fst f*)(*a* $:= *fst bc*)
and *finfun-fst-update-code*: *finfun-fst* (*finfun-update-code f a bc*) = (*finfun-fst f*)(*a* $:= *fst bc*)
⟨*proof*⟩

lemma *finfun-fst-comp-conv*: *finfun-fst* (*f* ∘$ *g*) = (*fst* ∘ *f*) ∘$ *g*
⟨*proof*⟩

lemma *finfun-fst-conv* [*simp*]: *finfun-fst* (f, *g*$) = *f*
⟨*proof*⟩

lemma *finfun-fst-conv-Abs-finfun*: *finfun-fst* = (λ*f*. *Abs-finfun* (*fst* ∘ *op* $ *f*))
⟨*proof*⟩

definition *finfun-snd* :: ′*a* ⇒*f* (′*b* × ′*c*) ⇒ ′*a* ⇒*f* ′*c*

where [*code*]: *finfun-snd f* = *snd* ∘$ *f*

lemma *finfun-snd-const*: *finfun-snd* (*K*$ *bc*) = (*K*$ *snd bc*)
⟨*proof*⟩

lemma *finfun-snd-update*: *finfun-snd* (*f*(*a* $:= *bc*)) = (*finfun-snd f*)(*a* $:= *snd bc*)
and *finfun-snd-update-code* [*code*]: *finfun-snd* (*finfun-update-code f a bc*) = (*finfun-snd f*)(*a* $:= *snd bc*)
⟨*proof*⟩

lemma *finfun-snd-comp-conv*: *finfun-snd* (*f* ∘$ *g*) = (*snd* ∘ *f*) ∘$ *g*
⟨*proof*⟩

lemma *finfun-snd-conv* [*simp*]: *finfun-snd* ($*f*, *g*$) = *g*
⟨*proof*⟩

lemma *finfun-snd-conv-Abs-finfun*: *finfun-snd* = (λ*f*. *Abs-finfun* (*snd* ∘ *op* $ *f*))
⟨*proof*⟩

lemma *finfun-Diag-collapse* [*simp*]: ($*finfun-fst f*, *finfun-snd f*$) = *f*
⟨*proof*⟩

73.14 Currying for FinFuns

definition *finfun-curry* :: ('*a* × '*b*) ⇒*f* '*c* ⇒ '*a* ⇒*f* '*b* ⇒*f* '*c*
where [*code del*]: *finfun-curry* = *finfun-rec* (*finfun-const* ∘ *finfun-const*) (λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*)))

interpretation *finfun-curry-aux*: *finfun-rec-wf-aux finfun-const* ∘ *finfun-const* λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*))
⟨*proof*⟩

interpretation *finfun-curry*: *finfun-rec-wf finfun-const* ∘ *finfun-const* λ(*a*, *b*) *c f*. *f*(*a* $:= (*f* $ *a*)(*b* $:= *c*))
⟨*proof*⟩

lemma *finfun-curry-const* [*simp*, *code*]: *finfun-curry* (*K*$ *c*) = (*K*$ *K*$ *c*)
⟨*proof*⟩

lemma *finfun-curry-update* [*simp*]:
finfun-curry (*f*((*a*, *b*) $:= *c*)) = (*finfun-curry f*)(*a* $:= (*finfun-curry f* $ *a*)(*b* $:= *c*))
and *finfun-curry-update-code* [*code*]:
finfun-curry (*finfun-update-code f* (*a*, *b*) *c*) = (*finfun-curry f*)(*a* $:= (*finfun-curry f* $ *a*)(*b* $:= *c*))
⟨*proof*⟩

lemma *finfun-Abs-finfun-curry*: **assumes** *fin*: *f* ∈ *finfun*
shows (λ*a*. *Abs-finfun* (*curry f a*)) ∈ *finfun*

including *finfun*
⟨*proof*⟩

lemma *finfun-curry-conv-curry*:
fixes $f :: ('a \times 'b) \Rightarrow f\ 'c$
shows *finfun-curry f* = *Abs-finfun* (λ*a*. *Abs-finfun* (*curry* (*finfun-apply f*) *a*))
including *finfun*
⟨*proof*⟩

73.15 Executable equality for FinFuns

lemma *eq-finfun-All-ext*: $(f = g) \longleftrightarrow$ *finfun-All* ((λ(*x*, *y*). *x* = *y*) ∘$ ($*f*, *g*$))
⟨*proof*⟩

instantiation *finfun* :: ({*card-UNIV*,*equal*},*equal*) *equal* **begin**
definition *eq-finfun-def* [*code*]: *HOL.equal f g* $\longleftrightarrow$ *finfun-All* ((λ(*x*, *y*). *x* = *y*) ∘$ ($*f*, *g*$))
instance ⟨*proof*⟩
end

lemma [*code nbe*]:
HOL.equal (f :: - ⇒*f* -) $f \longleftrightarrow$ *True*
⟨*proof*⟩

73.16 An operator that explicitly removes all redundant updates in the generated representations

definition *finfun-clearjunk* :: $'a \Rightarrow f\ 'b \Rightarrow 'a \Rightarrow f\ 'b$
where [*simp*, *code del*]: *finfun-clearjunk* = *id*

lemma *finfun-clearjunk-const* [*code*]: *finfun-clearjunk* (*K*$ *b*) = (*K*$ *b*)
⟨*proof*⟩

lemma *finfun-clearjunk-update* [*code*]:
finfun-clearjunk (*finfun-update-code f a b*) = *f*(*a* $:= *b*)
⟨*proof*⟩

73.17 The domain of a FinFun as a FinFun

definition *finfun-dom* :: $('a \Rightarrow f\ 'b) \Rightarrow ('a \Rightarrow f\ bool)$
where [*code del*]: *finfun-dom f* = *Abs-finfun* (λ*a*. *f* $ *a* ≠ *finfun-default f*)

lemma *finfun-dom-const*:
finfun-dom ((*K*$ *c*) :: $'a \Rightarrow f\ 'b$) = (*K*$ *finite* (*UNIV* :: $'a$ *set*) ∧ *c* ≠ *undefined*)
⟨*proof*⟩

finfun-dom raises an exception when called on a FinFun whose domain is a finite type. For such FinFuns, the default value (and as such the domain) is undefined.

lemma *finfun-dom-const-code* [*code*]:
finfun-dom ((*K$ c*) :: (*'a* :: *card-UNIV*) ⇒*f* *'b*) =
(*if CARD*('*a*) = *0 then* (*K$ False*) *else Code.abort* (*STR ''finfun-dom called on finite type''*) (λ-. *finfun-dom* (*K$ c*)))
⟨*proof*⟩

lemma *finfun-dom-finfunI*: (λ*a*. *f* $ *a* ≠ *finfun-default f*) ∈ *finfun*
⟨*proof*⟩

lemma *finfun-dom-update* [*simp*]:
finfun-dom (*f*(*a* $:= *b*)) = (*finfun-dom f*)(*a* $:= (*b* ≠ *finfun-default f*))
including *finfun* ⟨*proof*⟩

lemma *finfun-dom-update-code* [*code*]:
finfun-dom (*finfun-update-code f a b*) = *finfun-update-code* (*finfun-dom f*) *a* (*b* ≠ *finfun-default f*)
⟨*proof*⟩

lemma *finite-finfun-dom*: *finite* {*x*. *finfun-dom f* $ *x*}
⟨*proof*⟩

73.18 The domain of a FinFun as a sorted list

definition *finfun-to-list* :: (*'a* :: *linorder*) ⇒*f* *'b* ⇒ *'a list*
where
finfun-to-list f = (*THE xs*. *set xs* = {*x*. *finfun-dom f* $ *x*} ∧ *sorted xs* ∧ *distinct xs*)

lemma *set-finfun-to-list* [*simp*]: *set* (*finfun-to-list f*) = {*x*. *finfun-dom f* $ *x*} (**is** *?thesis1*)
and *sorted-finfun-to-list*: *sorted* (*finfun-to-list f*) (**is** *?thesis2*)
and *distinct-finfun-to-list*: *distinct* (*finfun-to-list f*) (**is** *?thesis3*)
⟨*proof*⟩

lemma *finfun-const-False-conv-bot*: *op* $ (*K$ False*) = *bot*
⟨*proof*⟩

lemma *finfun-const-True-conv-top*: *op* $ (*K$ True*) = *top*
⟨*proof*⟩

lemma *finfun-to-list-const*:
finfun-to-list ((*K$ c*) :: (*'a* :: {*linorder*} ⇒*f* *'b*)) =
(*if* ¬ *finite* (*UNIV* :: *'a set*) ∨ *c* = *undefined then* [] *else THE xs*. *set xs* = *UNIV* ∧ *sorted xs* ∧ *distinct xs*)
⟨*proof*⟩

lemma *finfun-to-list-const-code* [*code*]:
finfun-to-list ((*K$ c*) :: (*'a* :: {*linorder*, *card-UNIV*} ⇒*f* *'b*)) =
(*if CARD*('*a*) = *0 then* [] *else Code.abort* (*STR ''finfun-to-list called on finite*

type'') (λ-. *finfun-to-list* ((*K*$ *c*) :: ('*a* ⇒*f* '*b*))))
⟨*proof*⟩

lemma *remove1-insort-insert-same*:
x ∉ *set xs* ⟹ *remove1 x* (*insort-insert x xs*) = *xs*
⟨*proof*⟩

lemma *finfun-dom-conv*:
finfun-dom f $ *x* ⟷ *f* $ *x* ≠ *finfun-default f*
⟨*proof*⟩

lemma *finfun-to-list-update*:
finfun-to-list (*f*(*a* $:= *b*)) =
(*if b* = *finfun-default f then List.remove1 a* (*finfun-to-list f*) *else List.insort-insert a* (*finfun-to-list f*))
⟨*proof*⟩

lemma *finfun-to-list-update-code* [*code*]:
finfun-to-list (*finfun-update-code f a b*) =
(*if b* = *finfun-default f then List.remove1 a* (*finfun-to-list f*) *else List.insort-insert a* (*finfun-to-list f*))
⟨*proof*⟩

More type class instantiations

lemma *card-eq-1-iff*: *card A* = *1* ⟷ *A* ≠ {} ∧ (∀ *x*∈*A*. ∀ *y*∈*A*. *x* = *y*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *card-UNIV-finfun*:
defines *F* == *finfun* :: ('*a* ⇒ '*b*) *set*
shows *CARD*('*a* ⇒*f* '*b*) = (*if CARD*('*a*) ≠ *0* ∧ *CARD*('*b*) ≠ *0* ∨ *CARD*('*b*) = *1 then CARD*('*b*) ^ *CARD*('*a*) *else 0*)
⟨*proof*⟩

lemma *finite-UNIV-finfun*:
finite (*UNIV* :: ('*a* ⇒*f* '*b*) *set*) ⟷
(*finite* (*UNIV* :: '*a set*) ∧ *finite* (*UNIV* :: '*b set*) ∨ *CARD*('*b*) = *1*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

instantiation *finfun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* ⇒*f* '*b*)
(*let cb* = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in cb = *1* ∨ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧ *cb* ≠ *0*)
instance
⟨*proof*⟩
end

instantiation *finfun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*(*'a* ⇒*f* *'b*)
(*let ca* = *of-phantom* (*card-UNIV* :: *'a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: *'b card-UNIV*)
in if ca ≠ *0* ∧ *cb* ≠ *0* ∨ *cb* = *1 then cb* ^ *ca else 0*)
instance ⟨*proof*⟩
end

73.18.1 Bundles for concrete syntax

bundle *finfun-syntax*
begin

type-notation *finfun* ((- ⇒*f* /-) [*22*, *21*] *21*)

notation
finfun-const (*K*$/ - [*0*] *1*) **and**
finfun-update (-'(- $:= -') [*1000*, *0*, *0*] *1000*) **and**
finfun-apply (**infixl** $ *999*) **and**
finfun-comp (**infixr** ∘$ *55*) **and**
finfun-comp2 (**infixr** $∘ *55*) **and**
finfun-Diag ((*1'*($-,/ -$')) [*0*, *0*] *1000*)

notation (*ASCII*)
finfun-comp (**infixr** *o*$ *55*) **and**
finfun-comp2 (**infixr** $*o* *55*)

end

bundle *no-finfun-syntax*
begin

no-type-notation
finfun ((- ⇒*f* /-) [*22*, *21*] *21*)

no-notation
finfun-const (*K*$/ - [*0*] *1*) **and**
finfun-update (-'(- $:= -') [*1000*, *0*, *0*] *1000*) **and**
finfun-apply (**infixl** $ *999*) **and**
finfun-comp (**infixr** ∘$ *55*) **and**
finfun-comp2 (**infixr** $∘ *55*) **and**
finfun-Diag ((*1'*($-,/ -$')) [*0*, *0*] *1000*)

no-notation (*ASCII*)
finfun-comp (**infixr** *o*$ *55*) **and**
finfun-comp2 (**infixr** $*o* *55*)

end

unbundle *no-finfun-syntax*

end

74 Predicates modelled as FinFuns

theory *FinFunPred*
imports ~~/*src/HOL/Library/FinFun*
begin

unbundle *finfun-syntax*

Instantiate FinFun predicates just like predicates

type-synonym *'a pred*$_f$ = *'a* ⇒*f bool*

instantiation *finfun* :: (*type*, *ord*) *ord*
begin

definition *le-finfun-def* [*code del*]: $f \leq g \longleftrightarrow (\forall x.\ f\ \$\ x \leq g\ \$\ x)$

definition [*code del*]: $(f::'a \Rightarrow_f 'b) < g \longleftrightarrow f \leq g \wedge \neg\ g \leq f$

instance ⟨*proof*⟩

lemma *le-finfun-code* [*code*]:
$f \leq g \longleftrightarrow$ *finfun-All* $((\lambda(x,\ y).\ x \leq y) \circ\$\ (\$f,\ g\$))$
⟨*proof*⟩

end

instance *finfun* :: (*type*, *preorder*) *preorder*
⟨*proof*⟩

instance *finfun* :: (*type*, *order*) *order*
⟨*proof*⟩

instantiation *finfun* :: (*type*, *order-bot*) *order-bot* **begin**
definition *bot* = *finfun-const bot*
instance ⟨*proof*⟩
end

lemma *bot-finfun-apply* [*simp*]: *op* $ *bot* = (λ-. *bot*)
⟨*proof*⟩

instantiation *finfun* :: (*type*, *order-top*) *order-top* **begin**
definition *top* = *finfun-const top*
instance ⟨*proof*⟩
end

lemma *top-finfun-apply* [*simp*]: *op* $ *top* = (λ-. *top*)
⟨*proof*⟩

instantiation *finfun* :: (*type*, *inf*) *inf* **begin**
definition [*code*]: *inf f g* = (λ(*x*, *y*). *inf x y*) ∘$ ($*f*, *g*$)
instance ⟨*proof*⟩
end

lemma *inf-finfun-apply* [*simp*]: *op* $ (*inf f g*) = *inf* (*op* $ *f*) (*op* $ *g*)
⟨*proof*⟩

instantiation *finfun* :: (*type*, *sup*) *sup* **begin**
definition [*code*]: *sup f g* = (λ(*x*, *y*). *sup x y*) ∘$ ($*f*, *g*$)
instance ⟨*proof*⟩
end

lemma *sup-finfun-apply* [*simp*]: *op* $ (*sup f g*) = *sup* (*op* $ *f*) (*op* $ *g*)
⟨*proof*⟩

instance *finfun* :: (*type*, *semilattice-inf*) *semilattice-inf*
⟨*proof*⟩

instance *finfun* :: (*type*, *semilattice-sup*) *semilattice-sup*
⟨*proof*⟩

instance *finfun* :: (*type*, *lattice*) *lattice* ⟨*proof*⟩

instance *finfun* :: (*type*, *bounded-lattice*) *bounded-lattice*
⟨*proof*⟩

instance *finfun* :: (*type*, *distrib-lattice*) *distrib-lattice*
⟨*proof*⟩

instantiation *finfun* :: (*type*, *minus*) *minus* **begin**
definition *f* − *g* = *case-prod* (*op* −) ∘$ ($*f*, *g*$)
instance ⟨*proof*⟩
end

lemma *minus-finfun-apply* [*simp*]: *op* $ (*f* − *g*) = *op* $ *f* − *op* $ *g*
⟨*proof*⟩

instantiation *finfun* :: (*type*, *uminus*) *uminus* **begin**
definition − *A* = *uminus* ∘$ *A*
instance ⟨*proof*⟩
end

lemma *uminus-finfun-apply* [*simp*]: *op* $ (− *g*) = − *op* $ *g*
⟨*proof*⟩

instance *finfun* :: (*type*, *boolean-algebra*) *boolean-algebra*
⟨*proof*⟩

Replicate predicate operations for FinFuns

abbreviation *finfun-empty* :: *'a pred*$_f$ ({}$_f$)
where {}$_f$ ≡ *bot*

abbreviation *finfun-UNIV* :: *'a pred*$_f$
where *finfun-UNIV* ≡ *top*

definition *finfun-single* :: *'a* ⇒ *'a pred*$_f$
where [*code*]: *finfun-single x* = *finfun-empty*(*x* $:= *True*)

lemma *finfun-single-apply* [*simp*]:
finfun-single x $ *y* ⟷ *x* = *y*
⟨*proof*⟩

lemma [*iff*]:
shows *finfun-single-neq-bot*: *finfun-single x* ≠ *bot*
and *bot-neq-finfun-single*: *bot* ≠ *finfun-single x*
⟨*proof*⟩

lemma *finfun-leI* [*intro*!]: (!!*x*. *A* $ *x* ⟹ *B* $ *x*) ⟹ *A* ≤ *B*
⟨*proof*⟩

lemma *finfun-leD* [*elim*]: ⟦ *A* ≤ *B*; *A* $ *x* ⟧ ⟹ *B* $ *x*
⟨*proof*⟩

Bounded quantification. Warning: *finfun-Ball* and *finfun-Ex* may raise an exception, they should not be used for quickcheck

definition *finfun-Ball-except* :: *'a list* ⇒ *'a pred*$_f$ ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where [*code del*]: *finfun-Ball-except xs A P* = (∀ *a*. *A* $ *a* ⟶ *a* ∈ *set xs* ∨ *P a*)

lemma *finfun-Ball-except-const*:
finfun-Ball-except xs (*K*$ *b*) *P* ⟷ ¬ *b* ∨ *set xs* = *UNIV* ∨ *Code.abort* (*STR* *''finfun-ball-except''*) (λ*u*. *finfun-Ball-except xs* (*K*$ *b*) *P*)
⟨*proof*⟩

lemma *finfun-Ball-except-const-finfun-UNIV-code* [*code*]:
finfun-Ball-except xs (*K*$ *b*) *P* ⟷ ¬ *b* ∨ *is-list-UNIV xs* ∨ *Code.abort* (*STR* *''finfun-ball-except''*) (λ*u*. *finfun-Ball-except xs* (*K*$ *b*) *P*)
⟨*proof*⟩

lemma *finfun-Ball-except-update*:
finfun-Ball-except xs (*A*(*a* $:= *b*)) *P* = ((*a* ∈ *set xs* ∨ (*b* ⟶ *P a*)) ∧ *finfun-Ball-except* (*a* # *xs*) *A P*)
⟨*proof*⟩

lemma *finfun-Ball-except-update-code* [*code*]:

fixes $a :: 'a :: card\text{-}UNIV$
shows $finfun\text{-}Ball\text{-}except\ xs\ (finfun\text{-}update\text{-}code\ f\ a\ b)\ P = ((a \in set\ xs \vee (b \longrightarrow P\ a)) \wedge finfun\text{-}Ball\text{-}except\ (a\ \#\ xs)\ f\ P)$
⟨*proof*⟩

definition $finfun\text{-}Ball :: 'a\ pred_f \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where [*code del*]: $finfun\text{-}Ball\ A\ P = Ball\ \{x.\ A\ \$\ x\}\ P$

lemma *finfun-Ball-code* [*code*]: $finfun\text{-}Ball = finfun\text{-}Ball\text{-}except\ []$
⟨*proof*⟩

definition $finfun\text{-}Bex\text{-}except :: 'a\ list \Rightarrow 'a\ pred_f \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where [*code del*]: $finfun\text{-}Bex\text{-}except\ xs\ A\ P = (\exists a.\ A\ \$\ a \wedge a \notin set\ xs \wedge P\ a)$

lemma *finfun-Bex-except-const*:
$finfun\text{-}Bex\text{-}except\ xs\ (K\$\ b)\ P \longleftrightarrow b \wedge set\ xs \neq UNIV \wedge Code.abort\ (STR\ ''finfun\text{-}Bex\text{-}except'')\ (\lambda u.\ finfun\text{-}Bex\text{-}except\ xs\ (K\$\ b)\ P)$
⟨*proof*⟩

lemma *finfun-Bex-except-const-finfun-UNIV-code* [*code*]:
$finfun\text{-}Bex\text{-}except\ xs\ (K\$\ b)\ P \longleftrightarrow b \wedge \neg\ is\text{-}list\text{-}UNIV\ xs \wedge Code.abort\ (STR\ ''finfun\text{-}Bex\text{-}except'')\ (\lambda u.\ finfun\text{-}Bex\text{-}except\ xs\ (K\$\ b)\ P)$
⟨*proof*⟩

lemma *finfun-Bex-except-update*:
$finfun\text{-}Bex\text{-}except\ xs\ (A(a\ \$:=\ b))\ P \longleftrightarrow (a \notin set\ xs \wedge b \wedge P\ a) \vee finfun\text{-}Bex\text{-}except\ (a\ \#\ xs)\ A\ P$
⟨*proof*⟩

lemma *finfun-Bex-except-update-code* [*code*]:
fixes $a :: 'a :: card\text{-}UNIV$
shows $finfun\text{-}Bex\text{-}except\ xs\ (finfun\text{-}update\text{-}code\ f\ a\ b)\ P \longleftrightarrow ((a \notin set\ xs \wedge b \wedge P\ a) \vee finfun\text{-}Bex\text{-}except\ (a\ \#\ xs)\ f\ P)$
⟨*proof*⟩

definition $finfun\text{-}Bex :: 'a\ pred_f \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where [*code del*]: $finfun\text{-}Bex\ A\ P = Bex\ \{x.\ A\ \$\ x\}\ P$

lemma *finfun-Bex-code* [*code*]: $finfun\text{-}Bex = finfun\text{-}Bex\text{-}except\ []$
⟨*proof*⟩

Automatically replace predicate operations by finfun predicate operations where possible

lemma *iso-finfun-le* [*code-unfold*]:
$op\ \$\ A \leq op\ \$\ B \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *iso-finfun-less* [*code-unfold*]:

$op\ \$\ A < op\ \$\ B \longleftrightarrow A < B$
⟨*proof*⟩

lemma *iso-finfun-eq* [*code-unfold*]:
$op\ \$\ A = op\ \$\ B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *iso-finfun-sup* [*code-unfold*]:
$sup\ (op\ \$\ A)\ (op\ \$\ B) = op\ \$\ (sup\ A\ B)$
⟨*proof*⟩

lemma *iso-finfun-disj* [*code-unfold*]:
$A\ \$\ x \vee B\ \$\ x \longleftrightarrow sup\ A\ B\ \$\ x$
⟨*proof*⟩

lemma *iso-finfun-inf* [*code-unfold*]:
$inf\ (op\ \$\ A)\ (op\ \$\ B) = op\ \$\ (inf\ A\ B)$
⟨*proof*⟩

lemma *iso-finfun-conj* [*code-unfold*]:
$A\ \$\ x \wedge B\ \$\ x \longleftrightarrow inf\ A\ B\ \$\ x$
⟨*proof*⟩

lemma *iso-finfun-empty-conv* [*code-unfold*]:
$(\lambda\text{-}.\ False) = op\ \$\ \{\}_f$
⟨*proof*⟩

lemma *iso-finfun-UNIV-conv* [*code-unfold*]:
$(\lambda\text{-}.\ True) = op\ \$\ finfun\text{-}UNIV$
⟨*proof*⟩

lemma *iso-finfun-upd* [*code-unfold*]:
fixes $A :: 'a\ pred_f$
shows $(op\ \$\ A)(x := b) = op\ \$\ (A(x\ \$:= b))$
⟨*proof*⟩

lemma *iso-finfun-uminus* [*code-unfold*]:
fixes $A :: 'a\ pred_f$
shows $-\ op\ \$\ A = op\ \$\ (-\ A)$
⟨*proof*⟩

lemma *iso-finfun-minus* [*code-unfold*]:
fixes $A :: 'a\ pred_f$
shows $op\ \$\ A - op\ \$\ B = op\ \$\ (A - B)$
⟨*proof*⟩

Do not declare the following two theorems as [*code-unfold*], because this causes quickcheck to fail frequently when bounded quantification is used which raises an exception. For code generation, the same problems occur,

but then, no randomly generated FinFun is usually around.

lemma *iso-finfun-Ball-Ball*:
$(\forall x.\ A\ \$\ x \longrightarrow P\ x) \longleftrightarrow$ *finfun-Ball A P*
⟨*proof*⟩

lemma *iso-finfun-Bex-Bex*:
$(\exists x.\ A\ \$\ x \wedge P\ x) \longleftrightarrow$ *finfun-Bex A P*
⟨*proof*⟩

Test code setup

notepad begin
⟨*proof*⟩
end

declare *iso-finfun-Ball-Ball*[*code-unfold*]
notepad begin
⟨*proof*⟩
end
declare *iso-finfun-Ball-Ball*[*code-unfold del*]

end

75 Examples for the set comprehension to pointfree simproc

theory *Set-Comprehension-Pointfree-Examples*
imports *Main*
begin

declare [[*simproc add*: *finite-Collect*]]

lemma
finite (*UNIV*::'*a set*) ==> *finite* {*p*. *EX x*::'*a*. *p* = (*x*, *x*)}
⟨*proof*⟩

lemma
finite A ==> *finite B* ==> *finite* {*f a b*| *a b*. *a* : *A* ∧ *b* : *B*}
⟨*proof*⟩

lemma
finite B ==> *finite A'* ==> *finite* {*f a b*| *a b*. *a* : *A* ∧ *a* : *A'* ∧ *b* : *B*}
⟨*proof*⟩

lemma
finite A ==> *finite B* ==> *finite* {*f a b*| *a b*. *a* : *A* ∧ *b* : *B* ∧ *b* : *B'*}
⟨*proof*⟩

lemma

finite A ==> finite B ==> finite C ==> finite {f a b c| a b c. a : A ∧ b : B ∧ c : C}
⟨*proof*⟩

lemma
finite A ==> finite B ==> finite C ==> finite D ==>
finite {f a b c d| a b c d. a : A ∧ b : B ∧ c : C ∧ d : D}
⟨*proof*⟩

lemma
finite A ==> finite B ==> finite C ==> finite D ==> finite E ==>
finite {f a b c d e | a b c d e. a : A ∧ b : B ∧ c : C ∧ d : D ∧ e : E}
⟨*proof*⟩

lemma
finite A ==> finite B ==> finite C ==> finite D ==> finite E ⟹
finite {f a d c b e | e b c d a. b : B ∧ a : A ∧ e : E' ∧ c : C ∧ d : D ∧ e : E ∧ b : B'}
⟨*proof*⟩

lemma
⟦ *finite A* ; *finite B* ; *finite C* ; *finite D* ⟧
⟹ *finite* ({*f a b c d| a b c d. a : A ∧ b : B ∧ c : C ∧ d : D*})
⟨*proof*⟩

lemma
finite ((λ(*a*,*b*,*c*,*d*). *f a b c d*) ‘ (*A* × *B* × *C* × *D*))
⟹ *finite* ({*f a b c d| a b c d. a : A ∧ b : B ∧ c : C ∧ d : D*})
⟨*proof*⟩

lemma
finite S ==> finite {s'. EX s:S. s' = f a e s}
⟨*proof*⟩

lemma
finite A ==> finite B ==> finite {f a b| a b. a : A ∧ b : B ∧ a ∉ Z}
⟨*proof*⟩

lemma
finite A ==> finite B ==> finite R ==> finite {f a b x y| a b x y. a : A ∧ b : B ∧ (x,y) ∈ R}
⟨*proof*⟩

lemma
finite A ==> finite B ==> finite R ==> finite {f a b x y| a b x y. a : A ∧ (x,y) ∈ R ∧ b : B}
⟨*proof*⟩

lemma

finite A ==> finite B ==> finite R ==> finite {f a (x, b) y| y b x a. a : A ∧ (x,y) ∈ R ∧ b : B}
⟨*proof*⟩

lemma
finite A ==> finite AA ==> finite B ==> finite {f a b| a b. (a : A ∨ a : AA) ∧ b : B ∧ a ∉ Z}
⟨*proof*⟩

lemma
finite A1 ==> finite A2 ==> finite A3 ==> finite A4 ==> finite A5 ==> finite B ==>
finite {f a b c | a b c. ((a : A1 ∧ a : A2) ∨ (a : A3 ∧ (a : A4 ∨ a : A5))) ∧ b : B ∧ a ∉ Z}
⟨*proof*⟩

lemma *finite B ==> finite {c. EX x. x : B & c = a * x}*
⟨*proof*⟩

lemma
*finite A ==> finite B ==> finite {f a * g b |a b. a : A & b : B}*
⟨*proof*⟩

lemma
finite S ==> inj (%(x, y). g x y) ==> finite {f x y| x y. g x y : S}
⟨*proof*⟩

lemma
finite A ==> finite S ==> inj (%(x, y). g x y) ==> finite {f x y z | x y z. g x y : S & z : A}
⟨*proof*⟩

lemma
finite S ==> finite A ==> inj (%(x, y). g x y) ==> inj (%(x, y). h x y)
==> finite {f a b c d | a b c d. g a c : S & h b d : A}
⟨*proof*⟩

lemma
assumes *finite S* **shows** *finite {(a,b,c,d). ([a, b], [c, d]) : S}*
⟨*proof*⟩

schematic-goal
finite {x :: ?'A ⇒ ?'B ⇒ bool. ∃ a b. x = Pair-Rep a b}
= finite ((λ(b :: ?'B, a:: ?'A). Pair-Rep a b) ‘ (UNIV × UNIV))
⟨*proof*⟩

declare [[*simproc del*: *finite-Collect*]]

76 Testing simproc in code generation

definition *union* :: *nat set* => *nat set* => *nat set*
where
union A B = {*x*. *x* : *A* ∨ *x* : *B*}

definition *common-subsets* :: *nat set* => *nat set* => *nat set set*
where
common-subsets S1 S2 = {*S*. *S* ⊆ *S1* ∧ *S* ⊆ *S2*}

definition *products* :: *nat set* => *nat set* => *nat set*
where
products A B = {*c*. *EX a b*. *a* : *A* & *b* : *B* & *c* = *a* ∗ *b*}

export-code *products* **in** *Haskell*

export-code *union common-subsets products* **in** *Haskell*

end

77 Futures and parallel lists for code generated towards Isabelle/ML

theory *Parallel*
imports *Main*
begin

77.1 Futures

datatype *'a future* = *fork unit* ⇒ *'a*

primrec *join* :: *'a future* ⇒ *'a* **where**
join (*fork f*) = *f* ()

lemma *future-eqI* [*intro*!]:
assumes *join f* = *join g*
shows *f* = *g*
⟨*proof*⟩

code-printing
type-constructor *future* ⇀ (*Eval*) - *future*
| **constant** *fork* ⇀ (*Eval*) *Future.fork*
| **constant** *join* ⇀ (*Eval*) *Future.join*

code-reserved *Eval Future future*

77.2 Parallel lists

definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list* **where**

[*simp*]: *map* = *List.map*

definition *forall* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool* **where**
forall = *list-all*

lemma *forall-all* [*simp*]:
forall P xs ⟷ (∀ *x*∈*set xs*. *P x*)
⟨*proof*⟩

definition *exists* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool* **where**
exists = *list-ex*

lemma *exists-ex* [*simp*]:
exists P xs ⟷ (∃ *x*∈*set xs*. *P x*)
⟨*proof*⟩

code-printing
constant *map* ⇀ (*Eval*) *Par'-List.map*
| **constant** *forall* ⇀ (*Eval*) *Par'-List.forall*
| **constant** *exists* ⇀ (*Eval*) *Par'-List.exists*

code-reserved *Eval Par-List*

hide-const (**open**) *fork join map exists forall*

end

78 Debugging facilities for code generated towards Isabelle/ML

theory *Debug*
imports *Main*
begin

context
begin

qualified definition *trace* :: *String.literal* ⇒ *unit* **where**
[*simp*]: *trace s* = ()

qualified definition *tracing* :: *String.literal* ⇒ '*a* ⇒ '*a* **where**
[*simp*]: *tracing s* = *id*

lemma [*code*]:
tracing s = (*let u* = *trace s in id*)
⟨*proof*⟩ **definition** *flush* :: '*a* ⇒ *unit* **where**
[*simp*]: *flush x* = ()

qualified definition *flushing* :: *'a* ⇒ *'b* ⇒ *'b* **where**
  [*simp*]: *flushing x* = *id*

lemma [*code*, *code-unfold*]:
  flushing x = (*let u* = *flush x in id*)
  ⟨*proof*⟩ **definition** *timing* :: *String.literal* ⇒ (*'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b* **where**
  [*simp*]: *timing s f x* = *f x*

end

code-printing
  constant *Debug.trace* ⇀ (*Eval*) *Output.tracing*
| **constant** *Debug.flush* ⇀ (*Eval*) *Output.tracing*/ (@{*make'-string*} -) — note indirection via antiquotation
| **constant** *Debug.timing* ⇀ (*Eval*) *Timing.timeap'-msg*

code-reserved *Eval Output Timing*

end

79 A simple example demonstrating parallelism for code generated towards Isabelle/ML

theory *Parallel-Example*
imports *Complex-Main* ~~/*src/HOL/Library/Parallel* ~~/*src/HOL/Library/Debug*
begin

79.1 Compute-intensive examples.

79.1.1 Fragments of the harmonic series

definition *harmonic* :: *nat* ⇒ *rat* **where**
  harmonic n = *sum-list* (*map* (λ*n*. *1* / *of-nat n*) [*1*..<*n*])

79.1.2 The sieve of Erathostenes

The attentive reader may relate this ad-hoc implementation to the arithmetic notion of prime numbers as a little exercise.

primrec *mark* :: *nat* ⇒ *nat* ⇒ *bool list* ⇒ *bool list* **where**
  mark - - [] = []
| *mark m n* (*p* # *ps*) = (*case n of 0* ⇒ *False* # *mark m m ps*
    | *Suc n* ⇒ *p* # *mark m n ps*)

lemma *length-mark* [*simp*]:
  length (*mark m n ps*) = *length ps*
  ⟨*proof*⟩

function *sieve* :: *nat* ⇒ *bool list* ⇒ *bool list* **where**
sieve m ps = (*case dropWhile Not ps*
of [] ⇒ *ps*
| *p*#*ps*′ ⇒ *let n* = *m* − *length ps*′ *in takeWhile Not ps* @ *p* # *sieve m* (*mark n n ps*′))
⟨*proof*⟩

termination — tuning of this proof is left as an exercise to the reader
⟨*proof*⟩

primrec *natify* :: *nat* ⇒ *bool list* ⇒ *nat list* **where**
natify - [] = []
| *natify n* (*p*#*ps*) = (*if p then n* # *natify* (*Suc n*) *ps else natify* (*Suc n*) *ps*)

primrec *list-primes* **where**
list-primes (*Suc n*) = *natify 1* (*sieve n* (*False* # *replicate n True*))

79.1.3 Naive factorisation

function *factorise-from* :: *nat* ⇒ *nat* ⇒ *nat list* **where**
factorise-from k n = (*if* 1 < *k* ∧ *k* ≤ *n*
then
let (*q*, *r*) = *Divides.divmod-nat n k*
in if r = 0 *then k* # *factorise-from k q*
else factorise-from (*Suc k*) *n*
else [])
⟨*proof*⟩

termination *factorise-from* — tuning of this proof is left as an exercise to the reader
term *measure*
⟨*proof*⟩

definition *factorise* :: *nat* ⇒ *nat list* **where**
factorise n = *factorise-from* 2 *n*

79.2 Concurrent computation via futures

definition *computation-harmonic* :: *unit* ⇒ *rat* **where**
computation-harmonic - = *Debug.timing* (*STR* ''*harmonic example*'') *harmonic* 300

definition *computation-primes* :: *unit* ⇒ *nat list* **where**
computation-primes - = *Debug.timing* (*STR* ''*primes example*'') *list-primes* 4000

definition *computation-future* :: *unit* ⇒ *nat list* × *rat* **where**
computation-future = *Debug.timing* (*STR* ''*overall computation*'')
(λ() ⇒ *let c* = *Parallel.fork computation-harmonic*
in (*computation-primes* (), *Parallel.join c*))

value *computation-future* ()

definition *computation-factorise* :: *nat* ⇒ *nat list* **where**
computation-factorise = *Debug.timing* (*STR* ''*factorise*'') *factorise*

definition *computation-parallel* :: *unit* ⇒ *nat list list* **where**
computation-parallel - = *Debug.timing* (*STR* ''*overall computation*'')
(*Parallel.map computation-factorise*) [*20000*..<*20100*]

value *computation-parallel* ()

end

80 Immutable Arrays with Code Generation

theory *IArray*
imports *Main*
begin

Immutable arrays are lists wrapped up in an additional constructor. There are no update operations. Hence code generation can safely implement this type by efficient target language arrays. Currently only SML is provided. Should be extended to other target languages and more operations.

Note that arrays cannot be printed directly but only by turning them into lists first. Arrays could be converted back into lists for printing if they were wrapped up in an additional constructor.

context
begin

datatype 'a *iarray* = *IArray* 'a *list*

qualified primrec *list-of* :: 'a *iarray* ⇒ 'a *list* **where**
list-of (*IArray xs*) = *xs*

qualified definition *of-fun* :: (*nat* ⇒ 'a) ⇒ *nat* ⇒ 'a *iarray* **where**
[*simp*]: *of-fun f n* = *IArray* (*map f* [*0*..<*n*])

qualified definition *sub* :: 'a *iarray* ⇒ *nat* ⇒ 'a (**infixl** !! *100*) **where**
[*simp*]: *as* !! *n* = *IArray.list-of as* ! *n*

qualified definition *length* :: 'a *iarray* ⇒ *nat* **where**
[*simp*]: *length as* = *List.length* (*IArray.list-of as*)

qualified fun *all* :: ('a ⇒ *bool*) ⇒ 'a *iarray* ⇒ *bool* **where**
all p (*IArray as*) = (*ALL a* : *set as*. *p a*)

qualified fun *exists* :: ('a ⇒ *bool*) ⇒ 'a *iarray* ⇒ *bool* **where**
exists p (*IArray as*) = (*EX a* : *set as*. *p a*)

lemma *list-of-code* [*code*]:
IArray.list-of as = *map* (λ*n*. *as* !! *n*) [*0* ..< *IArray.length as*]
⟨*proof*⟩

end

80.1 Code Generation

code-reserved *SML Vector*

code-printing
  type-constructor *iarray* ⇀ (*SML*) - *Vector.vector*
| **constant** *IArray* ⇀ (*SML*) *Vector.fromList*
| **constant** *IArray.all* ⇀ (*SML*) *Vector.all*
| **constant** *IArray.exists* ⇀ (*SML*) *Vector.exists*

lemma [*code*]:
  size (*as* :: *'a iarray*) = *Suc* (*length* (*IArray.list-of as*))
  ⟨*proof*⟩

lemma [*code*]:
  size-iarray f as = *Suc* (*size-list f* (*IArray.list-of as*))
  ⟨*proof*⟩

lemma [*code*]:
  rec-iarray f as = *f* (*IArray.list-of as*)
  ⟨*proof*⟩

lemma [*code*]:
  case-iarray f as = *f* (*IArray.list-of as*)
  ⟨*proof*⟩

lemma [*code*]:
  set-iarray as = *set* (*IArray.list-of as*)
  ⟨*proof*⟩

lemma [*code*]:
  map-iarray f as = *IArray* (*map f* (*IArray.list-of as*))
  ⟨*proof*⟩

lemma [*code*]:
  rel-iarray r as bs = *list-all2 r* (*IArray.list-of as*) (*IArray.list-of bs*)
  ⟨*proof*⟩

lemma [*code*]:
  HOL.equal as bs ⟷ *HOL.equal* (*IArray.list-of as*) (*IArray.list-of bs*)
  ⟨*proof*⟩

context
begin

qualified primrec *tabulate* :: *integer* × (*integer* ⇒ *'a*) ⇒ *'a iarray* **where**
tabulate (*n*, *f*) = *IArray* (*map* (*f* ∘ *integer-of-nat*) [*0*..<*nat-of-integer n*])

end

lemma [*code*]:
IArray.of-fun f n = *IArray.tabulate* (*integer-of-nat n*, *f* ∘ *nat-of-integer*)
⟨*proof*⟩

code-printing
constant *IArray.tabulate* ⇀ (*SML*) *Vector.tabulate*

context
begin

qualified primrec *sub'* :: *'a iarray* × *integer* ⇒ *'a* **where**
[*code del*]: *sub'* (*as*, *n*) = *IArray.list-of as* ! *nat-of-integer n*

end

lemma [*code*]:
IArray.sub' (*IArray as*, *n*) = *as* ! *nat-of-integer n*
⟨*proof*⟩

lemma [*code*]:
as !! *n* = *IArray.sub'* (*as*, *integer-of-nat n*)
⟨*proof*⟩

code-printing
constant *IArray.sub'* ⇀ (*SML*) *Vector.sub*

context
begin

qualified definition *length'* :: *'a iarray* ⇒ *integer* **where**
[*code del*, *simp*]: *length' as* = *integer-of-nat* (*List.length* (*IArray.list-of as*))

end

lemma [*code*]:
IArray.length' (*IArray as*) = *integer-of-nat* (*List.length as*)
⟨*proof*⟩

lemma [*code*]:
IArray.length as = *nat-of-integer* (*IArray.length' as*)
⟨*proof*⟩

context *term-syntax*
begin

lemma [*code*]:
Code-Evaluation.term-of (*as* :: *'a::typerep iarray*) =
Code-Evaluation.Const (*STR ''IArray.iarray.IArray''*) (*TYPEREP*(*'a list* ⇒ *'a iarray*)) <·> (*Code-Evaluation.term-of* (*IArray.list-of as*))
⟨*proof*⟩

end

code-printing
constant *IArray.length'* ⇀ (*SML*) *Vector.length*

end

81 Implementation of integer numbers by target-language integers

theory *Code-Target-Int*
imports *../GCD*
begin

code-datatype *int-of-integer*

declare [[*code drop*: *integer-of-int*]]

context
includes *integer.lifting*
begin

lemma [*code*]:
integer-of-int (*int-of-integer k*) = *k*
⟨*proof*⟩

lemma [*code*]:
Int.Pos = *int-of-integer* ∘ *integer-of-num*
⟨*proof*⟩

lemma [*code*]:
Int.Neg = *int-of-integer* ∘ *uminus* ∘ *integer-of-num*
⟨*proof*⟩

lemma [*code-abbrev*]:
int-of-integer (*numeral k*) = *Int.Pos k*
⟨*proof*⟩

lemma [*code-abbrev*]:
int-of-integer (− *numeral k*) = *Int.Neg k*
⟨*proof*⟩

lemma [*code*, *symmetric*, *code-post*]:
0 = *int-of-integer 0*
⟨*proof*⟩

lemma [*code*, *symmetric*, *code-post*]:
1 = *int-of-integer 1*
⟨*proof*⟩

lemma [*code-post*]:
int-of-integer (− *1*) = − *1*
⟨*proof*⟩

lemma [*code*]:
k + *l* = *int-of-integer* (*of-int k* + *of-int l*)
⟨*proof*⟩

lemma [*code*]:
− *k* = *int-of-integer* (− *of-int k*)
⟨*proof*⟩

lemma [*code*]:
k − *l* = *int-of-integer* (*of-int k* − *of-int l*)
⟨*proof*⟩

lemma [*code*]:
Int.dup k = *int-of-integer* (*Code-Numeral.dup* (*of-int k*))
⟨*proof*⟩

declare [[*code drop*: *Int.sub*]]

lemma [*code*]:
k ∗ *l* = *int-of-integer* (*of-int k* ∗ *of-int l*)
⟨*proof*⟩

lemma [*code*]:
k div l = *int-of-integer* (*of-int k div of-int l*)
⟨*proof*⟩

lemma [*code*]:
k mod l = *int-of-integer* (*of-int k mod of-int l*)
⟨*proof*⟩

lemma [*code*]:
divmod m n = *map-prod int-of-integer int-of-integer* (*divmod m n*)
⟨*proof*⟩

lemma [*code*]:
  HOL.equal k l = *HOL.equal* (*of-int k* :: *integer*) (*of-int l*)
  ⟨*proof*⟩

lemma [*code*]:
  k ≤ *l* ⟷ (*of-int k* :: *integer*) ≤ *of-int l*
  ⟨*proof*⟩

lemma [*code*]:
  k < *l* ⟷ (*of-int k* :: *integer*) < *of-int l*
  ⟨*proof*⟩

declare [[*code drop*: *gcd* :: *int* ⇒ - *lcm* :: *int* ⇒ -]]

lemma *gcd-int-of-integer* [*code*]:
  gcd (*int-of-integer x*) (*int-of-integer y*) = *int-of-integer* (*gcd x y*)
⟨*proof*⟩

lemma *lcm-int-of-integer* [*code*]:
  lcm (*int-of-integer x*) (*int-of-integer y*) = *int-of-integer* (*lcm x y*)
⟨*proof*⟩

end

lemma (**in** *ring-1*) *of-int-code-if*:
  of-int k = (*if k* = *0 then 0*
    else if k < *0 then* − *of-int* (− *k*)
    else let
      l = *2* ∗ *of-int* (*k div 2*);
      j = *k mod 2*
    in if j = *0 then l else l* + *1*)
⟨*proof*⟩

declare *of-int-code-if* [*code*]

lemma [*code*]:
  nat = *nat-of-integer* ∘ *of-int*
  including *integer.lifting* ⟨*proof*⟩

code-identifier
  code-module *Code-Target-Int* ⇀
    (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

82 Implementation of natural numbers by target-language integers

theory *Code-Target-Nat*
imports *Code-Abstract-Nat*
begin

82.1 Implementation for *nat*

context
includes *natural.lifting integer.lifting*
begin

lift-definition *Nat* :: *integer* ⇒ *nat*
  is *nat*
  ⟨*proof*⟩

lemma [*code-post*]:
  Nat 0 = 0
  Nat 1 = 1
  Nat (*numeral k*) = *numeral k*
  ⟨*proof*⟩

lemma [*code-abbrev*]:
  integer-of-nat = *of-nat*
  ⟨*proof*⟩

lemma [*code-unfold*]:
  Int.nat (*int-of-integer k*) = *nat-of-integer k*
  ⟨*proof*⟩

lemma [*code abstype*]:
  Code-Target-Nat.Nat (*integer-of-nat n*) = *n*
  ⟨*proof*⟩

lemma [*code abstract*]:
  integer-of-nat (*nat-of-integer k*) = *max 0 k*
  ⟨*proof*⟩

lemma [*code-abbrev*]:
  nat-of-integer (*numeral k*) = *nat-of-num k*
  ⟨*proof*⟩

lemma [*code abstract*]:
  integer-of-nat (*nat-of-num n*) = *integer-of-num n*
  ⟨*proof*⟩

lemma [*code abstract*]:
  integer-of-nat 0 = 0

⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat 1 = 1
⟨*proof*⟩

lemma [*code*]:
Suc n = n + 1
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (m + n) = of-nat m + of-nat n
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (m − n) = max 0 (of-nat m − of-nat n)
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (m ∗ n) = of-nat m ∗ of-nat n
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (m div n) = of-nat m div of-nat n
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (m mod n) = of-nat m mod of-nat n
⟨*proof*⟩

lemma [*code*]:
Divides.divmod-nat m n = (m div n, m mod n)
⟨*proof*⟩

lemma [*code*]:
divmod m n = map-prod nat-of-integer nat-of-integer (divmod m n)
⟨*proof*⟩

lemma [*code*]:
HOL.equal m n = HOL.equal (of-nat m :: integer) (of-nat n)
⟨*proof*⟩

lemma [*code*]:
m ≤ n ⟷ (of-nat m :: integer) ≤ of-nat n
⟨*proof*⟩

lemma [*code*]:
m < n ⟷ (of-nat m :: integer) < of-nat n
⟨*proof*⟩

lemma *num-of-nat-code* [*code*]:
num-of-nat = *num-of-integer* ∘ *of-nat*
⟨*proof*⟩

end

lemma (**in** *semiring-1*) *of-nat-code-if*:
of-nat n = (*if n* = *0 then 0*
else let
(*m*, *q*) = *Divides.divmod-nat n 2*;
m′ = *2* ∗ *of-nat m*
in if q = *0 then m′ else m′* + *1*)
⟨*proof*⟩

declare *of-nat-code-if* [*code*]

definition *int-of-nat* :: *nat* ⇒ *int* **where**
[*code-abbrev*]: *int-of-nat* = *of-nat*

lemma [*code*]:
int-of-nat n = *int-of-integer* (*of-nat n*)
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*nat k*) = *max 0* (*integer-of-int k*)
including *integer.lifting* ⟨*proof*⟩

lemma *term-of-nat-code* [*code*]:
— Use *nat-of-integer* in term reconstruction instead of *Code-Target-Nat.Nat* such that reconstructed terms can be fed back to the code generator
term-of-class.term-of n =
Code-Evaluation.App
(*Code-Evaluation.Const* (*STR ''Code-Numeral.nat-of-integer''*)
(*typerep.Typerep* (*STR ''fun''*)
[*typerep.Typerep* (*STR ''Code-Numeral.integer''*) [],
typerep.Typerep (*STR ''Nat.nat''*) []]))
(*term-of-class.term-of* (*integer-of-nat n*))
⟨*proof*⟩

lemma *nat-of-integer-code-post* [*code-post*]:
nat-of-integer 0 = *0*
nat-of-integer 1 = *1*
nat-of-integer (*numeral k*) = *numeral k*
including *integer.lifting* ⟨*proof*⟩

code-identifier
code-module *Code-Target-Nat* ⇀
(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

83 Implementation of natural and integer numbers by target-language integers

theory *Code-Target-Numeral*
imports *Code-Target-Int Code-Target-Nat*
begin

end
theory *IArray-Examples*
imports ~~/*src*/*HOL*/*Library*/*IArray* ~~/*src*/*HOL*/*Library*/*Code-Target-Numeral*
begin

lemma *IArray* [*True*,*False*] !! *1* = *False*
⟨*proof*⟩

lemma *IArray.length* (*IArray* [[],[]]) = *2*
⟨*proof*⟩

lemma *IArray.list-of* (*IArray* [*1*,*3*::*int*]) = [*1*,*3*]
⟨*proof*⟩

lemma *IArray.list-of* (*IArray.of-fun* (%*n*. *n*∗*n*) *5*) = [*0*,*1*,*4*,*9*,*16*]
⟨*proof*⟩

lemma ¬ *IArray.all* (λ*x*. *x* > *2*) (*IArray* [*1*,*3*::*int*])
⟨*proof*⟩

lemma *IArray.exists* (λ*x*. *x* > *2*) (*IArray* [*1*,*3*::*int*])
⟨*proof*⟩

fun *sum2* :: *'a*::*monoid-add iarray* ⇒ *nat* ⇒ *'a* ⇒ *'a* **where**
sum2 A n s = (*if n*=*0 then s else sum2 A* (*n* − *1*) (*s* + *A*!!(*n* − *1*)))

definition *sum* :: *'a*::*monoid-add iarray* ⇒ *'a* **where**
sum A = *sum2 A* (*IArray.length A*) *0*

lemma *sum* (*IArray* [*1*,*2*,*3*,*4*,*5*,*6*,*7*,*8*,*9*::*int*]) = *45*
⟨*proof*⟩

end

theory *Simps-Case-Conv*
imports *Main*
keywords

simps-of-case case-of-simps :: *thy-decl*
abbrevs
simps-of-case =
case-of-simps =
begin

⟨*ML*⟩

end
theory *Simps-Case-Conv-Examples* **imports**
~~/*src*/*HOL*/*Library*/*Simps-Case-Conv*
begin

84 Tests for the Simps¡-¿Case conversion tools

fun *foo* **where**
foo (*x* # *xs*) *Nil* = *0* |
foo (*x* # *xs*) (*y* # *ys*) = *foo* [] [] |
foo Nil (*y* # *ys*) = *1* |
foo Nil Nil = *3*

fun *bar* **where**
bar x 0 y = *0* + *x* |
bar x (*Suc n*) *y* = *n* + *x*

definition
split-rule-test :: ((*nat* => *'a*) + (*'b* * ((*'b* => *'a*) *option*))) => (*'a* => *nat*) => *nat*
where
split-rule-test x f = *f* (*case x of Inl af* ⇒ *af 1*
| *Inr* (*b*, *None*) => *inv f 0*
| *Inr* (*b*, *Some g*) => *g b*)

definition *test* **where** *test x y* = (*case x of None* => (*case y of* [] => *1* | - # - => *2*) | *Some x* => *x*)

definition *nosplit* **where** *nosplit x* = *x* @ (*case x of* [] ⇒ [*1*] | *xs* ⇒ *xs*)

Function with complete, non-overlapping patterns

case-of-simps *foo-cases1*: *foo.simps*
lemma
fixes *xs* :: *'a list* **and** *ys* :: *'b list*
shows *foo xs ys* = (*case* (*xs*, *ys*) *of*
([], []) ⇒ *3*
| ([], *y* # *ys*) ⇒ *1*
| (*x* # *xs*, []) ⇒ *0*
| (*x* # *xs*, *y* # *ys*) ⇒ *foo* ([] :: *'a list*) ([] :: *'b list*))
⟨*proof*⟩

Redundant equations are ignored

case-of-simps *foo-cases2*: *foo.simps foo.simps*
lemma
  fixes *xs* :: *'a list* **and** *ys* :: *'b list*
  shows *foo xs ys* = (*case* (*xs*, *ys*) *of*
    ([], []) ⇒ *3*
    | ([], *y* # *ys*) ⇒ *1*
    | (*x* # *xs*, []) ⇒ *0*
    | (*x* # *xs*, *y* # *ys*) ⇒ *foo* ([] :: *'a list*) ([] :: *'b list*))
  ⟨*proof*⟩

Variable patterns

case-of-simps *bar-cases*: *bar.simps*
print-theorems

Case expression not at top level

simps-of-case *split-rule-test-simps*: *split-rule-test-def*
lemma
  split-rule-test (*Inl x*) *f* = *f* (*x 1*)
  split-rule-test (*Inr* (*x*, *None*)) *f* = *f* (*inv f 0*)
  split-rule-test (*Inr* (*x*, *Some y*)) *f* = *f* (*y x*)
  ⟨*proof*⟩

Argument occurs both as case parameter and seperately

simps-of-case *nosplit-simps1*: *nosplit-def*
lemma
  nosplit [] = [] @ [*1*]
  nosplit (*x* # *xs*) = (*x* # *xs*) @ *x* # *xs*
  ⟨*proof*⟩

Nested case expressions

simps-of-case *test-simps1*: *test-def*
lemma
  test None [] = *1*
  test None (*x* # *xs*) = *2*
  test (*Some x*) *y* = *x*
  ⟨*proof*⟩

Single-constructor patterns

case-of-simps *fst-conv-simps*: *fst-conv*
lemma *fst x* = (*case x of* (*a*,*b*) ⇒ *a*)
  ⟨*proof*⟩

Partial split of case

simps-of-case *nosplit-simps2*: *nosplit-def* (*splits*: *list.split*)
lemma
  nosplit [] = [] @ [*1*]

nosplit (*x* # *xs*) = (*x* # *xs*) @ *x* # *xs*
⟨*proof*⟩

simps-of-case *test-simps2*: *test-def* (*splits*: *option.split*)
lemma
test None y = (*case y of* [] ⇒ *1* | *x* # *xs* ⇒ *2*)
test (*Some x*) *y* = *x*
⟨*proof*⟩

Reversal

case-of-simps *test-def1*: *test-simps1*
lemma
test x y = (*case* (*x*,*y*) *of*
(*None*, []) ⇒ *1*
| (*None*, -#-) ⇒ *2*
| (*Some x*, -) ⇒ *x*)
⟨*proof*⟩

Case expressions on RHS

case-of-simps *test-def2*: *test-simps2*
lemma *test xs y* = (*case* (*xs*, *y*) *of* (*None*, []) ⇒ *1* | (*None*, *x* # *xa*) ⇒ *2* | (*Some x*, *y*) ⇒ *x*)
⟨*proof*⟩

Partial split of simps

case-of-simps *foo-cons-def*: *foo.simps*(*1*,*2*)
lemma
fixes *xs* :: *'a list* **and** *ys* :: *'b list*
shows *foo* (*x* # *xs*) *ys* = (*case* (*x*,*xs*,*ys*) *of*
(-,-,[]) ⇒ *0*
| (-,-,- # -) ⇒ *foo* ([] :: *'a list*) ([] :: *'b list*))
⟨*proof*⟩

end

85 Isabelle/ML basics

theory *ML*
imports *Main*
begin

86 ML expressions

The Isabelle command **ML** allows to embed Isabelle/ML source into the formal text. It is type-checked, compiled, and run within that environment.

Note that side-effects should be avoided, unless the intention is to change global parameters of the run-time environment (rare).

ML top-level bindings are managed within the theory context.

⟨*ML*⟩

87 Antiquotations

There are some language extensions (via antiquotations), as explained in the "Isabelle/Isar implementation manual", chapter 0.

⟨*ML*⟩

Formal entities from the surrounding context may be referenced as follows:

term *1 + 1* — term within theory source

⟨*ML*⟩

88 Recursive ML evaluation

⟨*ML*⟩

89 IDE support

ML embedded into the Isabelle environment is connected to the Prover IDE. Poly/ML provides:

- precise positions for warnings / errors
- markup for defining positions of identifiers
- markup for inferred types of sub-expressions
- pretty-printing of ML values with markup
- completion of ML names
- source-level debugger

⟨*ML*⟩

90 Example: factorial and ackermann function in Isabelle/ML

⟨*ML*⟩

See http://mathworld.wolfram.com/AckermannFunction.html.

⟨*ML*⟩

91 Parallel Isabelle/ML

Future.fork/join/cancel manage parallel evaluation.

Note that within Isabelle theory documents, the top-level command boundary may not be transgressed without special precautions. This is normally managed by the system when performing parallel proof checking.

⟨*ML*⟩

The `Par_List` module provides high-level combinators for parallel list operations.

⟨*ML*⟩

92 Function specifications in Isabelle/HOL

fun *factorial* :: *nat* ⇒ *nat*
where
  factorial 0 = *1*
| *factorial* (*Suc n*) = *Suc n* ∗ *factorial n*

term *factorial 4* — symbolic term
value *factorial 4* — evaluation via ML code generation in the background

declare [[*ML-source-trace*]]
⟨*ML*⟩

fun *ackermann* :: *nat* ⇒ *nat* ⇒ *nat*
where
  ackermann 0 n = *n* + *1*
| *ackermann* (*Suc m*) *0* = *ackermann m 1*
| *ackermann* (*Suc m*) (*Suc n*) = *ackermann m* (*ackermann* (*Suc m*) *n*)

value *ackermann 3 5*

end

theory *Rewrite*
imports *Main*
begin

consts *rewrite-HOLE* :: *'a*::{} (¤)

lemma *eta-expand*:
  fixes *f* :: *'a*::{} ⇒ *'b*::{}
  shows *f* ≡ λ*x*. *f x* ⟨*proof*⟩

lemma *rewr-imp*:

assumes $PROP\ A \equiv PROP\ B$
shows $(PROP\ A \Longrightarrow PROP\ C) \equiv (PROP\ B \Longrightarrow PROP\ C)$
⟨*proof*⟩

lemma *imp-cong-eq*:
$(PROP\ A \Longrightarrow (PROP\ B \Longrightarrow PROP\ C) \equiv (PROP\ B' \Longrightarrow PROP\ C')) \equiv$
$((PROP\ B \Longrightarrow PROP\ A \Longrightarrow PROP\ C) \equiv (PROP\ B' \Longrightarrow PROP\ A \Longrightarrow PROP\ C'))$
⟨*proof*⟩

⟨*ML*⟩

end
theory *Rewrite-Examples*
imports *Main* ~~/*src*/*HOL*/*Library*/*Rewrite*
begin

93 The rewrite Proof Method by Example

lemma
fixes a::int **and** b::int **and** c::int
assumes $P\ (b + a)$
shows $P\ (a + b)$
⟨*proof*⟩

lemma
fixes $a\ b\ c :: int$
assumes $f\ (a - a + (a - a)) + f\ (\ \ 0\ \ + c) = f\ 0 + f\ c$
shows $f\ (a - a + (a - a)) + f\ ((a - a) + c) = f\ 0 + f\ c$
⟨*proof*⟩

lemma
fixes $a\ b\ c :: int$
assumes $f\ (a - a +\ \ 0\ \ ) + f\ ((a - a) + c) = f\ 0 + f\ c$
shows $f\ (a - a + (a - a)) + f\ ((a - a) + c) = f\ 0 + f\ c$
⟨*proof*⟩

lemma
fixes $a\ b\ c :: int$
assumes $f\ (\ \ 0\ \ + (a - a)) + f\ ((a - a) + c) = f\ 0 + f\ c$
shows $f\ (a - a + (a - a)) + f\ ((a - a) + c) = f\ 0 + f\ c$
⟨*proof*⟩

lemma
fixes $a\ b\ c :: int$
assumes $f\ (a - a +\ \ 0\ \ ) + f\ ((a - a) + c) = f\ 0 + f\ c$
shows $f\ (a - a + (a - a)) + f\ ((a - a) + c) = f\ 0 + f\ c$
⟨*proof*⟩

lemma
fixes $x\ y :: nat$
shows$x + y > c \Longrightarrow y + x > c$
⟨*proof*⟩

lemma
fixes $x\ y :: nat$
assumes $y + x > c \Longrightarrow y + x > c$
shows $x + y > c \Longrightarrow y + x > c$
⟨*proof*⟩

lemma
fixes $x\ y :: nat$
assumes $y + x > c \Longrightarrow y + x > c$
shows $x + y > c \Longrightarrow y + x > c$
⟨*proof*⟩

lemma
fixes $x\ y :: nat$
assumes $y + x > c \Longrightarrow y + x > c$
shows $x + y > c \Longrightarrow y + x > c$
⟨*proof*⟩

lemma
assumes $P\ \{x{::}int.\ y + 1 = 1 + x\}$
shows $P\ \{x{::}int.\ y + 1 = x + 1\}$
⟨*proof*⟩

lemma
assumes $P\ \{x{::}int.\ y + 1 = 1 + x\}$
shows $P\ \{x{::}int.\ y + 1 = x + 1\}$
⟨*proof*⟩

lemma
assumes $P\ \{(x{::}nat,\ y{::}nat,\ z).\ x + z * 3 = Q\ (\lambda s\ t.\ s * t + y - 3)\}$
shows $P\ \{(x{::}nat,\ y{::}nat,\ z).\ x + z * 3 = Q\ (\lambda s\ t.\ y + s * t - 3)\}$
⟨*proof*⟩

lemma
assumes $PROP\ P \equiv PROP\ Q$
shows $PROP\ R \Longrightarrow PROP\ P \Longrightarrow PROP\ Q$
⟨*proof*⟩

lemma
assumes $PROP\ P \equiv PROP\ Q$
shows $PROP\ R \Longrightarrow PROP\ R \Longrightarrow PROP\ P \Longrightarrow PROP\ Q$

⟨*proof*⟩

lemma
assumes (*PROP P* ⟹ *PROP Q*) ≡ (*PROP S* ⟹ *PROP R*)
shows *PROP S* ⟹ (*PROP P* ⟹ *PROP Q*) ⟹ *PROP R*
⟨*proof*⟩

lemma *test-theorem*:
fixes *x* :: *nat*
shows *x* ≤ *y* ⟹ *x* ≥ *y* ⟹ *x* = *y*
⟨*proof*⟩

lemma
fixes *f* :: *nat* ⇒ *nat*
shows *f x* ≤ *0* ⟹ *f x* ≥ *0* ⟹ *f x* = *0*
⟨*proof*⟩

lemma
assumes *rewr*: *PROP P* ⟹ *PROP Q* ⟹ *PROP R* ≡ *PROP R′*
assumes *A1*: *PROP S* ⟹ *PROP T* ⟹ *PROP U* ⟹ *PROP P*
assumes *A2*: *PROP S* ⟹ *PROP T* ⟹ *PROP U* ⟹ *PROP Q*
assumes *C*: *PROP S* ⟹ *PROP R′* ⟹ *PROP T* ⟹ *PROP U* ⟹ *PROP V*
shows *PROP S* ⟹ *PROP R* ⟹ *PROP T* ⟹ *PROP U* ⟹ *PROP V*
⟨*proof*⟩

fun *f* :: *nat* ⇒ *nat* **where** *f n* = *n*
definition *f-inv* (*I* :: *nat* ⇒ *bool*) *n* ≡ *f n*

lemma *annotate-f*: *f* = *f-inv I*
⟨*proof*⟩

lemma
assumes *P* (λ*n*. *f-inv* (λ-. *True*) *n* + *1*) = *x*
shows *P* (λ*n*. *f n* + *1*) = *x*
⟨*proof*⟩

lemma

assumes $P\ (\lambda n.\ f\text{-}inv\ (\lambda x.\ n < x + 1)\ n + 1) = x$
shows $P\ (\lambda n.\ f\ n + 1) = x$
⟨*proof*⟩

lemma
assumes $P\ (\lambda n.\ f\text{-}inv\ (\lambda x.\ n < x + 1)\ n + 1) = x$
shows $P\ (\lambda n.\ f\ n + 1) = x$
⟨*proof*⟩

lemma
assumes $P\ (2 + 1)$
shows $\bigwedge x\ y.\ P\ (1 + 2 :: nat)$
⟨*proof*⟩

lemma
assumes $\bigwedge x\ y.\ P\ (y + x)$
shows $\bigwedge x\ y.\ P\ (x + y :: nat)$
⟨*proof*⟩

lemma
assumes $\bigwedge x\ y\ z.\ y + x + z = z + y + (x::int)$
shows $\bigwedge x\ y\ z.\ x + y + z = z + y + (x::int)$
⟨*proof*⟩

lemma
assumes $\bigwedge x\ y\ z.\ z + (x + y) = z + y + (x::int)$
shows $\bigwedge x\ y\ z.\ x + y + z = z + y + (x::int)$
⟨*proof*⟩

lemma
assumes $\bigwedge x\ y\ z.\ x + y + z = y + z + (x::int)$
shows $\bigwedge x\ y\ z.\ x + y + z = z + y + (x::int)$
⟨*proof*⟩

lemma
assumes *eq*: $\bigwedge x.\ P\ x \Longrightarrow g\ x = x$
assumes *f1*: $\bigwedge x.\ Q\ x \Longrightarrow P\ x$
assumes *f2*: $\bigwedge x.\ Q\ x \Longrightarrow x$
shows $\bigwedge x.\ Q\ x \Longrightarrow g\ x$
⟨*proof*⟩

lemma
assumes $(\bigwedge (x::int).\ x < 1 + x)$
and $(x::int) + 1 > x$
shows $(\bigwedge (x::int).\ x + 1 > x) \Longrightarrow (x::int) + 1 > x$
⟨*proof*⟩

lemma
assumes $\bigwedge a\ b.\ P\ ((a\ +\ 1)\ *\ (1\ +\ b))$
shows $\bigwedge a\ b :: nat.\ P\ ((a\ +\ 1)\ *\ (b\ +\ 1))$
⟨*proof*⟩

lemma
assumes $Q\ (\lambda b :: int.\ P\ (\lambda a.\ a\ +\ b)\ (\lambda a.\ a\ +\ b))$
shows $Q\ (\lambda b :: int.\ P\ (\lambda a.\ a\ +\ b)\ (\lambda a.\ b\ +\ a))$
⟨*proof*⟩

⟨*ML*⟩

94 Regression tests

⟨*ML*⟩

lemma
assumes *eq*: $PROP\ A \Longrightarrow PROP\ B \equiv PROP\ C$
assumes *f1*: $PROP\ D \Longrightarrow PROP\ A$
assumes *f2*: $PROP\ D \Longrightarrow PROP\ C$
shows $\bigwedge x.\ PROP\ D \Longrightarrow PROP\ B$
⟨*proof*⟩

end

95 Examples for proof methods "sat" and "satx"

theory *SAT-Examples*
imports *Main*
begin

lemma *True*
⟨*proof*⟩

lemma $a\ |\ {\sim}a$
⟨*proof*⟩

lemma $(a\ |\ b)\ \&\ {\sim}a \Longrightarrow b$
⟨*proof*⟩

lemma $(a\ \&\ b)\ |\ (c\ \&\ d) \Longrightarrow (a\ \&\ b)\ |\ (c\ \&\ d)$

⟨*proof*⟩

lemma $(a \;\&\; b) \mid (c \;\&\; d) \Longrightarrow (a \;\&\; b) \mid (c \;\&\; d)$

⟨*proof*⟩

lemma $(a \;\&\; b \mid c \;\&\; d) \;\&\; (e \;\&\; f \mid g \;\&\; h) \mid (i \;\&\; j \mid k \;\&\; l) \;\&\; (m \;\&\; n \mid p \;\&\; q)$
$\Longrightarrow (a \;\&\; b \mid c \;\&\; d) \;\&\; (e \;\&\; f \mid g \;\&\; h) \mid (i \;\&\; j \mid k \;\&\; l) \;\&\; (m \;\&\; n \mid p \;\&\; q)$

⟨*proof*⟩

lemma $(a \;\&\; b \mid c \;\&\; d) \;\&\; (e \;\&\; f \mid g \;\&\; h) \mid (i \;\&\; j \mid k \;\&\; l) \;\&\; (m \;\&\; n \mid p \;\&\; q)$
$\Longrightarrow (a \;\&\; b \mid c \;\&\; d) \;\&\; (e \;\&\; f \mid g \;\&\; h) \mid (i \;\&\; j \mid k \;\&\; l) \;\&\; (m \;\&\; n \mid p \;\&\; q)$

⟨*proof*⟩

lemma $P{=}P{=}P{=}P{=}P{=}P{=}P{=}P{=}P$
⟨*proof*⟩

lemma $P{=}P{=}P{=}P{=}P{=}P{=}P{=}P{=}P$
⟨*proof*⟩

lemma $!!\; a\; b\; c.\; [|\; a \mid b \mid c \mid d\; ;$
$e \mid f \mid (a \;\&\; d)\; ;$
$\sim(a \mid (c \;\&\; \sim c)) \mid b\; ;$
$\sim(b \;\&\; (x \mid \sim x)) \mid c\; ;$
$\sim(d \mid \mathit{False}) \mid c\; ;$
$\sim(c \mid (\sim p \;\&\; (p \mid (q \;\&\; \sim q)))) \;|]] \Longrightarrow \mathit{False}$
⟨*proof*⟩

lemma $!!\; a\; b\; c.\; [|\; a \mid b \mid c \mid d\; ;$
$e \mid f \mid (a \;\&\; d)\; ;$
$\sim(a \mid (c \;\&\; \sim c)) \mid b\; ;$
$\sim(b \;\&\; (x \mid \sim x)) \mid c\; ;$
$\sim(d \mid \mathit{False}) \mid c\; ;$
$\sim(c \mid (\sim p \;\&\; (p \mid (q \;\&\; \sim q)))) \;|]] \Longrightarrow \mathit{False}$
⟨*proof*⟩

eta-Equivalence

lemma $(\mathit{ALL}\; x.\; P\; x) \mid \sim \mathit{All}\; P$
⟨*proof*⟩

declare [[*sat-trace* = *false*]]
declare [[*quick-and-dirty* = *false*]]

⟨*ML*⟩

lemma assumes *1*: $\sim x0$

and *2*: ~*x30*
and *3*: ~*x29*
and *4*: ~*x59*
and *5*: *x1* | *x31* | *x0*
and *6*: *x2* | *x32* | *x1*
and *7*: *x3* | *x33* | *x2*
and *8*: *x4* | *x34* | *x3*
and *9*: *x35* | *x4*
and *10*: *x5* | *x36* | *x30*
and *11*: *x6* | *x37* | *x5* | *x31*
and *12*: *x7* | *x38* | *x6* | *x32*
and *13*: *x8* | *x39* | *x7* | *x33*
and *14*: *x9* | *x40* | *x8* | *x34*
and *15*: *x41* | *x9* | *x35*
and *16*: *x10* | *x42* | *x36*
and *17*: *x11* | *x43* | *x10* | *x37*
and *18*: *x12* | *x44* | *x11* | *x38*
and *19*: *x13* | *x45* | *x12* | *x39*
and *20*: *x14* | *x46* | *x13* | *x40*
and *21*: *x47* | *x14* | *x41*
and *22*: *x15* | *x48* | *x42*
and *23*: *x16* | *x49* | *x15* | *x43*
and *24*: *x17* | *x50* | *x16* | *x44*
and *25*: *x18* | *x51* | *x17* | *x45*
and *26*: *x19* | *x52* | *x18* | *x46*
and *27*: *x53* | *x19* | *x47*
and *28*: *x20* | *x54* | *x48*
and *29*: *x21* | *x55* | *x20* | *x49*
and *30*: *x22* | *x56* | *x21* | *x50*
and *31*: *x23* | *x57* | *x22* | *x51*
and *32*: *x24* | *x58* | *x23* | *x52*
and *33*: *x59* | *x24* | *x53*
and *34*: *x25* | *x54*
and *35*: *x26* | *x25* | *x55*
and *36*: *x27* | *x26* | *x56*
and *37*: *x28* | *x27* | *x57*
and *38*: *x29* | *x28* | *x58*
and *39*: ~*x1* | ~*x31*
and *40*: ~*x1* | ~*x0*
and *41*: ~*x31* | ~*x0*
and *42*: ~*x2* | ~*x32*
and *43*: ~*x2* | ~*x1*
and *44*: ~*x32* | ~*x1*
and *45*: ~*x3* | ~*x33*
and *46*: ~*x3* | ~*x2*
and *47*: ~*x33* | ~*x2*
and *48*: ~*x4* | ~*x34*
and *49*: ~*x4* | ~*x3*
and *50*: ~*x34* | ~*x3*

and *51*: ~*x35* | ~*x4*
and *52*: ~*x5* | ~*x36*
and *53*: ~*x5* | ~*x30*
and *54*: ~*x36* | ~*x30*
and *55*: ~*x6* | ~*x37*
and *56*: ~*x6* | ~*x5*
and *57*: ~*x6* | ~*x31*
and *58*: ~*x37* | ~*x5*
and *59*: ~*x37* | ~*x31*
and *60*: ~*x5* | ~*x31*
and *61*: ~*x7* | ~*x38*
and *62*: ~*x7* | ~*x6*
and *63*: ~*x7* | ~*x32*
and *64*: ~*x38* | ~*x6*
and *65*: ~*x38* | ~*x32*
and *66*: ~*x6* | ~*x32*
and *67*: ~*x8* | ~*x39*
and *68*: ~*x8* | ~*x7*
and *69*: ~*x8* | ~*x33*
and *70*: ~*x39* | ~*x7*
and *71*: ~*x39* | ~*x33*
and *72*: ~*x7* | ~*x33*
and *73*: ~*x9* | ~*x40*
and *74*: ~*x9* | ~*x8*
and *75*: ~*x9* | ~*x34*
and *76*: ~*x40* | ~*x8*
and *77*: ~*x40* | ~*x34*
and *78*: ~*x8* | ~*x34*
and *79*: ~*x41* | ~*x9*
and *80*: ~*x41* | ~*x35*
and *81*: ~*x9* | ~*x35*
and *82*: ~*x10* | ~*x42*
and *83*: ~*x10* | ~*x36*
and *84*: ~*x42* | ~*x36*
and *85*: ~*x11* | ~*x43*
and *86*: ~*x11* | ~*x10*
and *87*: ~*x11* | ~*x37*
and *88*: ~*x43* | ~*x10*
and *89*: ~*x43* | ~*x37*
and *90*: ~*x10* | ~*x37*
and *91*: ~*x12* | ~*x44*
and *92*: ~*x12* | ~*x11*
and *93*: ~*x12* | ~*x38*
and *94*: ~*x44* | ~*x11*
and *95*: ~*x44* | ~*x38*
and *96*: ~*x11* | ~*x38*
and *97*: ~*x13* | ~*x45*
and *98*: ~*x13* | ~*x12*
and *99*: ~*x13* | ~*x39*

and *100*: ~*x45* | ~*x12*
and *101*: ~*x45* | ~*x39*
and *102*: ~*x12* | ~*x39*
and *103*: ~*x14* | ~*x46*
and *104*: ~*x14* | ~*x13*
and *105*: ~*x14* | ~*x40*
and *106*: ~*x46* | ~*x13*
and *107*: ~*x46* | ~*x40*
and *108*: ~*x13* | ~*x40*
and *109*: ~*x47* | ~*x14*
and *110*: ~*x47* | ~*x41*
and *111*: ~*x14* | ~*x41*
and *112*: ~*x15* | ~*x48*
and *113*: ~*x15* | ~*x42*
and *114*: ~*x48* | ~*x42*
and *115*: ~*x16* | ~*x49*
and *116*: ~*x16* | ~*x15*
and *117*: ~*x16* | ~*x43*
and *118*: ~*x49* | ~*x15*
and *119*: ~*x49* | ~*x43*
and *120*: ~*x15* | ~*x43*
and *121*: ~*x17* | ~*x50*
and *122*: ~*x17* | ~*x16*
and *123*: ~*x17* | ~*x44*
and *124*: ~*x50* | ~*x16*
and *125*: ~*x50* | ~*x44*
and *126*: ~*x16* | ~*x44*
and *127*: ~*x18* | ~*x51*
and *128*: ~*x18* | ~*x17*
and *129*: ~*x18* | ~*x45*
and *130*: ~*x51* | ~*x17*
and *131*: ~*x51* | ~*x45*
and *132*: ~*x17* | ~*x45*
and *133*: ~*x19* | ~*x52*
and *134*: ~*x19* | ~*x18*
and *135*: ~*x19* | ~*x46*
and *136*: ~*x52* | ~*x18*
and *137*: ~*x52* | ~*x46*
and *138*: ~*x18* | ~*x46*
and *139*: ~*x53* | ~*x19*
and *140*: ~*x53* | ~*x47*
and *141*: ~*x19* | ~*x47*
and *142*: ~*x20* | ~*x54*
and *143*: ~*x20* | ~*x48*
and *144*: ~*x54* | ~*x48*
and *145*: ~*x21* | ~*x55*
and *146*: ~*x21* | ~*x20*
and *147*: ~*x21* | ~*x49*
and *148*: ~*x55* | ~*x20*

and *149*: ~*x55* | ~*x49*
and *150*: ~*x20* | ~*x49*
and *151*: ~*x22* | ~*x56*
and *152*: ~*x22* | ~*x21*
and *153*: ~*x22* | ~*x50*
and *154*: ~*x56* | ~*x21*
and *155*: ~*x56* | ~*x50*
and *156*: ~*x21* | ~*x50*
and *157*: ~*x23* | ~*x57*
and *158*: ~*x23* | ~*x22*
and *159*: ~*x23* | ~*x51*
and *160*: ~*x57* | ~*x22*
and *161*: ~*x57* | ~*x51*
and *162*: ~*x22* | ~*x51*
and *163*: ~*x24* | ~*x58*
and *164*: ~*x24* | ~*x23*
and *165*: ~*x24* | ~*x52*
and *166*: ~*x58* | ~*x23*
and *167*: ~*x58* | ~*x52*
and *168*: ~*x23* | ~*x52*
and *169*: ~*x59* | ~*x24*
and *170*: ~*x59* | ~*x53*
and *171*: ~*x24* | ~*x53*
and *172*: ~*x25* | ~*x54*
and *173*: ~*x26* | ~*x25*
and *174*: ~*x26* | ~*x55*
and *175*: ~*x25* | ~*x55*
and *176*: ~*x27* | ~*x26*
and *177*: ~*x27* | ~*x56*
and *178*: ~*x26* | ~*x56*
and *179*: ~*x28* | ~*x27*
and *180*: ~*x28* | ~*x57*
and *181*: ~*x27* | ~*x57*
and *182*: ~*x29* | ~*x28*
and *183*: ~*x29* | ~*x58*
and *184*: ~*x28* | ~*x58*
shows *False*
⟨*proof*⟩

lemma assumes *1*: *x0* | *x1* | *x2* | *x3* | *x4* | *x5* | *x6*
and *2*: *x7* | *x8* | *x9* | *x10* | *x11* | *x12* | *x13*
and *3*: *x14* | *x15* | *x16* | *x17* | *x18* | *x19* | *x20*
and *4*: *x21* | *x22* | *x23* | *x24* | *x25* | *x26* | *x27*
and *5*: *x28* | *x29* | *x30* | *x31* | *x32* | *x33* | *x34*
and *6*: *x35* | *x36* | *x37* | *x38* | *x39* | *x40* | *x41*
and *7*: *x42* | *x43* | *x44* | *x45* | *x46* | *x47* | *x48*
and *8*: *x49* | *x50* | *x51* | *x52* | *x53* | *x54* | *x55*

and *9*: ~*x0* | ~*x7*
and *10*: ~*x0* | ~*x14*
and *11*: ~*x0* | ~*x21*
and *12*: ~*x0* | ~*x28*
and *13*: ~*x0* | ~*x35*
and *14*: ~*x0* | ~*x42*
and *15*: ~*x0* | ~*x49*
and *16*: ~*x7* | ~*x14*
and *17*: ~*x7* | ~*x21*
and *18*: ~*x7* | ~*x28*
and *19*: ~*x7* | ~*x35*
and *20*: ~*x7* | ~*x42*
and *21*: ~*x7* | ~*x49*
and *22*: ~*x14* | ~*x21*
and *23*: ~*x14* | ~*x28*
and *24*: ~*x14* | ~*x35*
and *25*: ~*x14* | ~*x42*
and *26*: ~*x14* | ~*x49*
and *27*: ~*x21* | ~*x28*
and *28*: ~*x21* | ~*x35*
and *29*: ~*x21* | ~*x42*
and *30*: ~*x21* | ~*x49*
and *31*: ~*x28* | ~*x35*
and *32*: ~*x28* | ~*x42*
and *33*: ~*x28* | ~*x49*
and *34*: ~*x35* | ~*x42*
and *35*: ~*x35* | ~*x49*
and *36*: ~*x42* | ~*x49*
and *37*: ~*x1* | ~*x8*
and *38*: ~*x1* | ~*x15*
and *39*: ~*x1* | ~*x22*
and *40*: ~*x1* | ~*x29*
and *41*: ~*x1* | ~*x36*
and *42*: ~*x1* | ~*x43*
and *43*: ~*x1* | ~*x50*
and *44*: ~*x8* | ~*x15*
and *45*: ~*x8* | ~*x22*
and *46*: ~*x8* | ~*x29*
and *47*: ~*x8* | ~*x36*
and *48*: ~*x8* | ~*x43*
and *49*: ~*x8* | ~*x50*
and *50*: ~*x15* | ~*x22*
and *51*: ~*x15* | ~*x29*
and *52*: ~*x15* | ~*x36*
and *53*: ~*x15* | ~*x43*
and *54*: ~*x15* | ~*x50*
and *55*: ~*x22* | ~*x29*
and *56*: ~*x22* | ~*x36*
and *57*: ~*x22* | ~*x43*

and *58*: ~*x22* | ~*x50*
and *59*: ~*x29* | ~*x36*
and *60*: ~*x29* | ~*x43*
and *61*: ~*x29* | ~*x50*
and *62*: ~*x36* | ~*x43*
and *63*: ~*x36* | ~*x50*
and *64*: ~*x43* | ~*x50*
and *65*: ~*x2* | ~*x9*
and *66*: ~*x2* | ~*x16*
and *67*: ~*x2* | ~*x23*
and *68*: ~*x2* | ~*x30*
and *69*: ~*x2* | ~*x37*
and *70*: ~*x2* | ~*x44*
and *71*: ~*x2* | ~*x51*
and *72*: ~*x9* | ~*x16*
and *73*: ~*x9* | ~*x23*
and *74*: ~*x9* | ~*x30*
and *75*: ~*x9* | ~*x37*
and *76*: ~*x9* | ~*x44*
and *77*: ~*x9* | ~*x51*
and *78*: ~*x16* | ~*x23*
and *79*: ~*x16* | ~*x30*
and *80*: ~*x16* | ~*x37*
and *81*: ~*x16* | ~*x44*
and *82*: ~*x16* | ~*x51*
and *83*: ~*x23* | ~*x30*
and *84*: ~*x23* | ~*x37*
and *85*: ~*x23* | ~*x44*
and *86*: ~*x23* | ~*x51*
and *87*: ~*x30* | ~*x37*
and *88*: ~*x30* | ~*x44*
and *89*: ~*x30* | ~*x51*
and *90*: ~*x37* | ~*x44*
and *91*: ~*x37* | ~*x51*
and *92*: ~*x44* | ~*x51*
and *93*: ~*x3* | ~*x10*
and *94*: ~*x3* | ~*x17*
and *95*: ~*x3* | ~*x24*
and *96*: ~*x3* | ~*x31*
and *97*: ~*x3* | ~*x38*
and *98*: ~*x3* | ~*x45*
and *99*: ~*x3* | ~*x52*
and *100*: ~*x10* | ~*x17*
and *101*: ~*x10* | ~*x24*
and *102*: ~*x10* | ~*x31*
and *103*: ~*x10* | ~*x38*
and *104*: ~*x10* | ~*x45*
and *105*: ~*x10* | ~*x52*
and *106*: ~*x17* | ~*x24*

and *107*: ~*x17* | ~*x31*
and *108*: ~*x17* | ~*x38*
and *109*: ~*x17* | ~*x45*
and *110*: ~*x17* | ~*x52*
and *111*: ~*x24* | ~*x31*
and *112*: ~*x24* | ~*x38*
and *113*: ~*x24* | ~*x45*
and *114*: ~*x24* | ~*x52*
and *115*: ~*x31* | ~*x38*
and *116*: ~*x31* | ~*x45*
and *117*: ~*x31* | ~*x52*
and *118*: ~*x38* | ~*x45*
and *119*: ~*x38* | ~*x52*
and *120*: ~*x45* | ~*x52*
and *121*: ~*x4* | ~*x11*
and *122*: ~*x4* | ~*x18*
and *123*: ~*x4* | ~*x25*
and *124*: ~*x4* | ~*x32*
and *125*: ~*x4* | ~*x39*
and *126*: ~*x4* | ~*x46*
and *127*: ~*x4* | ~*x53*
and *128*: ~*x11* | ~*x18*
and *129*: ~*x11* | ~*x25*
and *130*: ~*x11* | ~*x32*
and *131*: ~*x11* | ~*x39*
and *132*: ~*x11* | ~*x46*
and *133*: ~*x11* | ~*x53*
and *134*: ~*x18* | ~*x25*
and *135*: ~*x18* | ~*x32*
and *136*: ~*x18* | ~*x39*
and *137*: ~*x18* | ~*x46*
and *138*: ~*x18* | ~*x53*
and *139*: ~*x25* | ~*x32*
and *140*: ~*x25* | ~*x39*
and *141*: ~*x25* | ~*x46*
and *142*: ~*x25* | ~*x53*
and *143*: ~*x32* | ~*x39*
and *144*: ~*x32* | ~*x46*
and *145*: ~*x32* | ~*x53*
and *146*: ~*x39* | ~*x46*
and *147*: ~*x39* | ~*x53*
and *148*: ~*x46* | ~*x53*
and *149*: ~*x5* | ~*x12*
and *150*: ~*x5* | ~*x19*
and *151*: ~*x5* | ~*x26*
and *152*: ~*x5* | ~*x33*
and *153*: ~*x5* | ~*x40*
and *154*: ~*x5* | ~*x47*
and *155*: ~*x5* | ~*x54*

and *156*: ~*x12* | ~*x19*
and *157*: ~*x12* | ~*x26*
and *158*: ~*x12* | ~*x33*
and *159*: ~*x12* | ~*x40*
and *160*: ~*x12* | ~*x47*
and *161*: ~*x12* | ~*x54*
and *162*: ~*x19* | ~*x26*
and *163*: ~*x19* | ~*x33*
and *164*: ~*x19* | ~*x40*
and *165*: ~*x19* | ~*x47*
and *166*: ~*x19* | ~*x54*
and *167*: ~*x26* | ~*x33*
and *168*: ~*x26* | ~*x40*
and *169*: ~*x26* | ~*x47*
and *170*: ~*x26* | ~*x54*
and *171*: ~*x33* | ~*x40*
and *172*: ~*x33* | ~*x47*
and *173*: ~*x33* | ~*x54*
and *174*: ~*x40* | ~*x47*
and *175*: ~*x40* | ~*x54*
and *176*: ~*x47* | ~*x54*
and *177*: ~*x6* | ~*x13*
and *178*: ~*x6* | ~*x20*
and *179*: ~*x6* | ~*x27*
and *180*: ~*x6* | ~*x34*
and *181*: ~*x6* | ~*x41*
and *182*: ~*x6* | ~*x48*
and *183*: ~*x6* | ~*x55*
and *184*: ~*x13* | ~*x20*
and *185*: ~*x13* | ~*x27*
and *186*: ~*x13* | ~*x34*
and *187*: ~*x13* | ~*x41*
and *188*: ~*x13* | ~*x48*
and *189*: ~*x13* | ~*x55*
and *190*: ~*x20* | ~*x27*
and *191*: ~*x20* | ~*x34*
and *192*: ~*x20* | ~*x41*
and *193*: ~*x20* | ~*x48*
and *194*: ~*x20* | ~*x55*
and *195*: ~*x27* | ~*x34*
and *196*: ~*x27* | ~*x41*
and *197*: ~*x27* | ~*x48*
and *198*: ~*x27* | ~*x55*
and *199*: ~*x34* | ~*x41*
and *200*: ~*x34* | ~*x48*
and *201*: ~*x34* | ~*x55*
and *202*: ~*x41* | ~*x48*
and *203*: ~*x41* | ~*x55*
and *204*: ~*x48* | ~*x55*

shows *False*
⟨*proof*⟩

Function `benchmark` takes the name of an existing DIMACS CNF file, parses this file, passes the problem to a SAT solver, and checks the proof of unsatisfiability found by the solver. The function measures the time spent on proof reconstruction (at least real time also includes time spent in the SAT solver), and additionally returns the number of resolution steps in the proof.

⟨*ML*⟩

end

96 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming

theory *Sum-of-Squares*
imports *Complex-Main*
begin

⟨*ML*⟩

end

theory *SOS*
imports ~~/*src/HOL/Library/Sum-of-Squares*
begin

lemma (*3*::*real*) ∗ *x* + *7* ∗ *a* < *4* & *3* < *2* ∗ *x* ⟹ *a* < *0*
⟨*proof*⟩

lemma *a1* ≥ *0* ∧ *a2* ≥ *0* ∧ (*a1* ∗ *a1* + *a2* ∗ *a2* = *b1* ∗ *b1* + *b2* ∗ *b2* + *2*) ∧ (*a1* ∗ *b1* + *a2* ∗ *b2* = *0*) ⟶
a1 ∗ *a2* − *b1* ∗ *b2* ≥ (*0*::*real*)
⟨*proof*⟩

lemma (*3*::*real*) ∗ *x* + *7* ∗ *a* < *4* & *3* < *2* ∗ *x* ⟶ *a* < *0*
⟨*proof*⟩

lemma (*0*::*real*) ≤ *x* ∧ *x* ≤ *1* ∧ *0* ≤ *y* ∧ *y* ≤ *1* ⟶
*x*² + *y*² < *1* ∨ (*x* − *1*)² + *y*² < *1* ∨ *x*² + (*y* − *1*)² < *1* ∨ (*x* − *1*)² + (*y* − *1*)² < *1*
⟨*proof*⟩

lemma $(0::real) \leq x \wedge 0 \leq y \wedge 0 \leq z \wedge x + y + z \leq 3 \longrightarrow x * y + x * z + y * z \geq 3 * x * y * z$
⟨*proof*⟩

lemma $(x::real)^2 + y^2 + z^2 = 1 \longrightarrow (x + y + z)^2 \leq 3$
⟨*proof*⟩

lemma $w^2 + x^2 + y^2 + z^2 = 1 \longrightarrow (w + x + y + z)^2 \leq (4::real)$
⟨*proof*⟩

lemma $(x::real) \geq 1 \wedge y \geq 1 \longrightarrow x * y \geq x + y - 1$
⟨*proof*⟩

lemma $(x::real) > 1 \wedge y > 1 \longrightarrow x * y > x + y - 1$
⟨*proof*⟩

lemma $|x| \leq 1 \longrightarrow |64 * x\hat{}7 - 112 * x\hat{}5 + 56 * x\hat{}3 - 7 * x| \leq (1::real)$
⟨*proof*⟩

One component of denominator in dodecahedral example.

lemma $2 \leq x \wedge x \leq 125841 / 50000 \wedge 2 \leq y \wedge y \leq 125841 / 50000 \wedge 2 \leq z \wedge z \leq 125841 / 50000 \longrightarrow$
$2 * (x * z + x * y + y * z) - (x * x + y * y + z * z) \geq (0::real)$
⟨*proof*⟩

Over a larger but simpler interval.

lemma $(2::real) \leq x \wedge x \leq 4 \wedge 2 \leq y \wedge y \leq 4 \wedge 2 \leq z \wedge z \leq 4 \longrightarrow$
$0 \leq 2 * (x * z + x * y + y * z) - (x * x + y * y + z * z)$
⟨*proof*⟩

We can do 12. I think 12 is a sharp bound; see PP's certificate.

lemma $2 \leq (x::real) \wedge x \leq 4 \wedge 2 \leq y \wedge y \leq 4 \wedge 2 \leq z \wedge z \leq 4 \longrightarrow$
$12 \leq 2 * (x * z + x * y + y * z) - (x * x + y * y + z * z)$
⟨*proof*⟩

Inequality from sci.math (see "Leon-Sotelo, por favor").

lemma $0 \leq (x::real) \wedge 0 \leq y \wedge x * y = 1 \longrightarrow x + y \leq x^2 + y^2$
⟨*proof*⟩

lemma $0 \leq (x::real) \wedge 0 \leq y \wedge x * y = 1 \longrightarrow x * y * (x + y) \leq x^2 + y^2$
⟨*proof*⟩

lemma $0 \leq (x::real) \wedge 0 \leq y \longrightarrow x * y * (x + y)^2 \leq (x^2 + y^2)^2$
⟨*proof*⟩

lemma $(0::real) \leq a \wedge 0 \leq b \wedge 0 \leq c \wedge c * (2 * a + b)\hat{}3 / 27 \leq x \longrightarrow c * a^2 * b \leq x$
⟨*proof*⟩

lemma $(0::real) < x \longrightarrow 0 < 1 + x + x^2$
⟨*proof*⟩

lemma $(0::real) \leq x \longrightarrow 0 < 1 + x + x^2$
⟨*proof*⟩

lemma $(0::real) < 1 + x^2$
⟨*proof*⟩

lemma $(0::real) \leq 1 + 2 * x + x^2$
⟨*proof*⟩

lemma $(0::real) < 1 + |x|$
⟨*proof*⟩

lemma $(0::real) < 1 + (1 + x)^2 * |x|$
⟨*proof*⟩

lemma $|(1::real) + x^2| = (1::real) + x^2$
⟨*proof*⟩

lemma $(3::real) * x + 7 * a < 4 \wedge 3 < 2 * x \longrightarrow a < 0$
⟨*proof*⟩

lemma $(0::real) < x \longrightarrow 1 < y \longrightarrow y * x \leq z \longrightarrow x < z$
⟨*proof*⟩

lemma $(1::real) < x \longrightarrow x^2 < y \longrightarrow 1 < y$
⟨*proof*⟩

lemma $(b::real)^2 < 4 * a * c \longrightarrow a * x^2 + b * x + c \neq 0$
⟨*proof*⟩

lemma $(b::real)^2 < 4 * a * c \longrightarrow a * x^2 + b * x + c \neq 0$
⟨*proof*⟩

lemma $(a::real) * x^2 + b * x + c = 0 \longrightarrow b^2 \geq 4 * a * c$
⟨*proof*⟩

lemma $(0::real) \leq b \wedge 0 \leq c \wedge 0 \leq x \wedge 0 \leq y \wedge x^2 = c \wedge y^2 = a^2 * c + b \longrightarrow a * c \leq y * x$
⟨*proof*⟩

lemma $|x - z| \leq e \wedge |y - z| \leq e \wedge 0 \leq u \wedge 0 \leq v \wedge u + v = 1 \text{ --> } |(u * x + v * y) - z| \leq (e::real)$
⟨*proof*⟩

lemma $(x::real) - y - 2 * x\hat{}4 = 0 \wedge 0 \leq x \wedge x \leq 2 \wedge 0 \leq y \wedge y \leq 3 \longrightarrow y^2 - 7 * y - 12 * x + 17 \geq 0$
⟨*proof*⟩

lemma $(0::real) \leq x \longrightarrow (1 + x + x^2) / (1 + x^2) \leq 1 + x$
⟨*proof*⟩

lemma $(0::real) \leq x \longrightarrow 1 - x \leq 1 / (1 + x + x^2)$
⟨*proof*⟩

lemma $(x::real) \leq 1 / 2 \longrightarrow - x - 2 * x^2 \leq - x / (1 - x)$
⟨*proof*⟩

lemma $4 * r^2 = p^2 - 4 * q \wedge r \geq (0::real) \wedge x^2 + p * x + q = 0 \longrightarrow$
$2 * (x::real) = - p + 2 * r \vee 2 * x = - p - 2 * r$
⟨*proof*⟩

end

theory *SOS-Cert*
imports ~~/*src/HOL/Library/Sum-of-Squares*
begin

lemma $(3::real) * x + 7 * a < 4 \wedge 3 < 2 * x \Longrightarrow a < 0$
⟨*proof*⟩

lemma $a1 \geq 0 \wedge a2 \geq 0 \wedge (a1 * a1 + a2 * a2 = b1 * b1 + b2 * b2 + 2) \wedge (a1 * b1 + a2 * b2 = 0) \longrightarrow$
$a1 * a2 - b1 * b2 \geq (0::real)$
⟨*proof*⟩

lemma $(3::real) * x + 7 * a < 4 \wedge 3 < 2 * x \longrightarrow a < 0$
⟨*proof*⟩

lemma $(0::real) \leq x \wedge x \leq 1 \wedge 0 \leq y \wedge y \leq 1 \longrightarrow$
$x^2 + y^2 < 1 \vee (x - 1)^2 + y^2 < 1 \vee x^2 + (y - 1)^2 < 1 \vee (x - 1)^2 + (y - 1)^2 < 1$
⟨*proof*⟩

lemma $(0::real) \leq x \wedge 0 \leq y \wedge 0 \leq z \wedge x + y + z \leq 3 \longrightarrow x * y + x * z + y * z \geq 3 * x * y * z$
⟨*proof*⟩

lemma $(x::real)^2 + y^2 + z^2 = 1 \longrightarrow (x + y + z)^2 \leq 3$
⟨*proof*⟩

lemma $w^2 + x^2 + y^2 + z^2 = 1 \longrightarrow (w + x + y + z)^2 \leq (4::real)$
⟨*proof*⟩

lemma $(x::real) \geq 1 \wedge y \geq 1 \longrightarrow x * y \geq x + y - 1$
⟨*proof*⟩

lemma $(x::real) > 1 \wedge y > 1 \longrightarrow x * y > x + y - 1$
⟨*proof*⟩

lemma $|x| \leq 1 \longrightarrow |64 * x\hat{}7 - 112 * x\hat{}5 + 56 * x\hat{}3 - 7 * x| \leq (1::real)$
⟨*proof*⟩

One component of denominator in dodecahedral example.

lemma $2 \leq x \wedge x \leq 125841 / 50000 \wedge 2 \leq y \wedge y \leq 125841 / 50000 \wedge 2 \leq z \wedge z \leq 125841 / 50000 \longrightarrow$
$2 * (x * z + x * y + y * z) - (x * x + y * y + z * z) \geq (0::real)$
⟨*proof*⟩

Over a larger but simpler interval.

lemma $(2::real) \leq x \wedge x \leq 4 \wedge 2 \leq y \wedge y \leq 4 \wedge 2 \leq z \wedge z \leq 4 \longrightarrow$
$0 \leq 2 * (x * z + x * y + y * z) - (x * x + y * y + z * z)$
⟨*proof*⟩

We can do 12. I think 12 is a sharp bound; see PP's certificate.

lemma $2 \leq (x::real) \wedge x \leq 4 \wedge 2 \leq y \wedge y \leq 4 \wedge 2 \leq z \wedge z \leq 4 \longrightarrow$
$12 \leq 2 * (x * z + x * y + y * z) - (x * x + y * y + z * z)$
⟨*proof*⟩

Inequality from sci.math (see "Leon-Sotelo, por favor").

lemma $0 \leq (x::real) \wedge 0 \leq y \wedge x * y = 1 \longrightarrow x + y \leq x^2 + y^2$
⟨*proof*⟩

lemma $0 \leq (x::real) \wedge 0 \leq y \wedge x * y = 1 \longrightarrow x * y * (x + y) \leq x^2 + y^2$
⟨*proof*⟩

lemma $0 \leq (x::real) \wedge 0 \leq y \longrightarrow x * y * (x + y)^2 \leq (x^2 + y^2)^2$
⟨*proof*⟩

lemma $(0::real) \leq a \wedge 0 \leq b \wedge 0 \leq c \wedge c * (2 * a + b)\hat{}3 / 27 \leq x \longrightarrow c * a^2 * b \leq x$
⟨*proof*⟩

lemma $(0::real) < x \longrightarrow 0 < 1 + x + x^2$
⟨*proof*⟩

lemma $(0::real) \leq x \longrightarrow 0 < 1 + x + x^2$
⟨*proof*⟩

lemma $(0::real) < 1 + x^2$
⟨*proof*⟩

lemma $(0::real) \leq 1 + 2 * x + x^2$
⟨*proof*⟩

lemma $(0::real) < 1 + |x|$
⟨*proof*⟩

lemma $(0::real) < 1 + (1 + x)^2 * |x|$
⟨*proof*⟩

lemma $|(1::real) + x^2| = (1::real) + x^2$
⟨*proof*⟩

lemma $(3::real) * x + 7 * a < 4 \wedge 3 < 2 * x \longrightarrow a < 0$
⟨*proof*⟩

lemma $(0::real) < x \longrightarrow 1 < y \longrightarrow y * x \leq z \longrightarrow x < z$
⟨*proof*⟩

lemma $(1::real) < x \longrightarrow x^2 < y \longrightarrow 1 < y$
⟨*proof*⟩

lemma $(b::real)^2 < 4 * a * c \longrightarrow a * x^2 + b * x + c \neq 0$
⟨*proof*⟩

lemma $(b::real)^2 < 4 * a * c \longrightarrow a * x\hat{}2 + b * x + c \neq 0$
⟨*proof*⟩

lemma $(a::real) * x^2 + b * x + c = 0 \longrightarrow b^2 \geq 4 * a * c$
⟨*proof*⟩

lemma $(0::real) \leq b \wedge 0 \leq c \wedge 0 \leq x \wedge 0 \leq y \wedge x^2 = c \wedge y^2 = a^2 * c + b \longrightarrow a * c \leq y * x$
⟨*proof*⟩

lemma $|x - z| \leq e \wedge |y - z| \leq e \wedge 0 \leq u \wedge 0 \leq v \wedge u + v = 1 \longrightarrow |(u * x + v * y) - z| \leq (e::real)$
⟨*proof*⟩

lemma $(x::real) - y - 2 * x\hat{}4 = 0 \wedge 0 \leq x \wedge x \leq 2 \wedge 0 \leq y \wedge y \leq 3 \longrightarrow y^2 - 7 * y - 12 * x + 17 \geq 0$
⟨*proof*⟩

lemma $(0::real) \leq x \longrightarrow (1 + x + x^2) \,/\, (1 + x^2) \leq 1 + x$
⟨*proof*⟩

lemma $(0::real) \leq x \longrightarrow 1 - x \leq 1 \,/\, (1 + x + x^2)$

⟨*proof*⟩

lemma (*x*::*real*) ≤ *1* / *2* ⟶ − *x* − *2* ∗ *x*^{2} ≤ − *x* / (*1* − *x*)
⟨*proof*⟩

lemma *4* ∗ *r*^{2} = *p*^{2} − *4* ∗ *q* ∧ *r* ≥ (*0*::*real*) ∧ *x*^{2} + *p* ∗ *x* + *q* = *0* ⟶ *2* ∗ (*x*::*real*) = − *p* + *2* ∗ *r* ∨ *2* ∗ *x* = − *p* − *2* ∗ *r*
⟨*proof*⟩

end

97 Bertrand's Ballot Theorem

theory *Ballot*
imports
Complex-Main
~~/*src*/*HOL*/*Library*/*FuncSet*
begin

97.1 Preliminaries

lemma *card-bij'*:
assumes *f* ∈ *A* → *B* ⋀*x*. *x* ∈ *A* ⟹ *g* (*f x*) = *x*
and *g* ∈ *B* → *A* ⋀*x*. *x* ∈ *B* ⟹ *f* (*g x*) = *x*
shows *card A* = *card B*
⟨*proof*⟩

97.2 Formalization of Problem Statement

97.2.1 Basic Definitions

datatype *vote* = *A* | *B*

definition
all-countings a b = *card* {*f* ∈ {*1* .. *a* + *b*} →$_E$ {*A*, *B*}.
card {*x* ∈ {*1* .. *a* + *b*}. *f x* = *A*} = *a* ∧ *card* {*x* ∈ {*1* .. *a* + *b*}. *f x* = *B*} = *b*}

definition
valid-countings a b =
card {*f*∈{*1*..*a*+*b*} →$_E$ {*A*, *B*}.
card {*x*∈{*1*..*a*+*b*}. *f x* = *A*} = *a* ∧ *card* {*x*∈{*1*..*a*+*b*}. *f x* = *B*} = *b* ∧
(∀ *m*∈{*1*..*a*+*b*}. *card* {*x*∈{*1*..*m*}. *f x* = *A*} > *card* {*x*∈{*1*..*m*}. *f x* = *B*})}

97.2.2 Equivalence with Set Cardinality

lemma *Collect-on-transfer*:
assumes *rel-set R X Y*
shows *rel-fun* (*rel-fun R op* =) (*rel-set R*) (λ*P*. {*x*∈*X*. *P x*}) (λ*P*. {*y*∈*Y*. *P y*})
⟨*proof*⟩

lemma *rel-fun-trans*:
rel-fun P Q g g' $\Longrightarrow$ *rel-fun R P f f'* $\Longrightarrow$ *rel-fun R Q* $(\lambda x.\ g\ (f\ x))\ (\lambda y.\ g'\ (f'\ y))$
⟨*proof*⟩

lemma *rel-fun-trans2*:
rel-fun P1 (*rel-fun P2 Q*) *g g'* $\Longrightarrow$ *rel-fun R P1 f1 f1'* $\Longrightarrow$ *rel-fun R P2 f2 f2'* $\Longrightarrow$
rel-fun R Q $(\lambda x.\ g\ (f1\ x)\ (f2\ x))\ (\lambda y.\ g'\ (f1'\ y)\ (f2'\ y))$
⟨*proof*⟩

lemma *rel-fun-trans2'*:
rel-fun R (*op* =) *f1 f1'* $\Longrightarrow$ *rel-fun R* (*op* =) *f2 f2'* $\Longrightarrow$
rel-fun R (*op* =) $(\lambda x.\ g\ (f1\ x)\ (f2\ x))\ (\lambda y.\ g\ (f1'\ y)\ (f2'\ y))$
⟨*proof*⟩

lemma *rel-fun-const*: *rel-fun R* (*op* =) $(\lambda x.\ a)\ (\lambda y.\ a)$
⟨*proof*⟩

lemma *rel-fun-conj*:
rel-fun R (*op* =) *f f'* $\Longrightarrow$ *rel-fun R* (*op* =) *g g'* $\Longrightarrow$ *rel-fun R* (*op* =) $(\lambda x.\ f\ x \wedge g\ x)\ (\lambda y.\ f'\ y \wedge g'\ y)$
⟨*proof*⟩

lemma *rel-fun-ball*:
$(\bigwedge i.\ i \in I \Longrightarrow$ *rel-fun R* (*op* =) (*f i*) (*f' i*)) $\Longrightarrow$ *rel-fun R* (*op* =) $(\lambda x.\ \forall\, i{\in}I.\ f\ i\ x)\ (\lambda y.\ \forall\, i{\in}I.\ f'\ i\ y)$
⟨*proof*⟩

lemma
shows *all-countings-set*: *all-countings a b* = *card* $\{V{\in}Pow\ \{0..{<}a{+}b\}.\ card\ V = a\}$
(**is** - = *card ?A*)
and *valid-countings-set*: *valid-countings a b* =
card $\{V{\in}Pow\ \{0..{<}a{+}b\}.\ card\ V = a \wedge (\forall\, m{\in}\{1..a{+}b\}.\ card\ (\{0..{<}m\} \cap V) > m - card\ (\{0..{<}m\} \cap V))\}$
(**is** - = *card ?V*)
⟨*proof*⟩

lemma *all-countings*: *all-countings a b* = (*a* + *b*) *choose a*
⟨*proof*⟩

97.3 Facts About *valid-countings*

97.3.1 Non-Recursive Cases

lemma *card-V-eq-a*: $V \subseteq \{0..{<}a\} \Longrightarrow$ *card V* = *a* $\longleftrightarrow$ $V = \{0..{<}a\}$
⟨*proof*⟩

lemma *valid-countings-a-0*: *valid-countings a 0* = *1*

⟨*proof*⟩

lemma *valid-countings-eq-zero*:
$a \leq b \Longrightarrow 0 < b \Longrightarrow$ *valid-countings* $a\ b = 0$
⟨*proof*⟩

lemma *Ico-subset-finite*: $i \subseteq \{a\ ..<\ b::nat\} \Longrightarrow$ *finite* i
⟨*proof*⟩

lemma *Icc-Suc2*: $a \leq b \Longrightarrow \{a..Suc\ b\} = insert\ (Suc\ b)\ \{a..b\}$
⟨*proof*⟩

lemma *Ico-Suc2*: $a \leq b \Longrightarrow \{a..<Suc\ b\} = insert\ b\ \{a..<b\}$
⟨*proof*⟩

lemma *valid-countings-Suc-Suc*:
assumes $b < a$
shows *valid-countings* $(Suc\ a)\ (Suc\ b) =$ *valid-countings* $a\ (Suc\ b) +$ *valid-countings* $(Suc\ a)\ b$
⟨*proof*⟩

lemma *valid-countings*:
$(a + b) *$ *valid-countings* $a\ b = (a - b) *$ $((a + b)$ *choose* $a)$
⟨*proof*⟩

lemma *valid-countings-eq*[*code*]:
valid-countings $a\ b =$ (*if* $a + b = 0$ *then* 1 *else* $((a - b) *$ $((a + b)$ *choose* $a))$ *div* $(a + b))$
⟨*proof*⟩

97.4 Relation Between *valid-countings* and *all-countings*

lemma *main-nat*: $(a + b) *$ *valid-countings* $a\ b = (a - b) *$ *all-countings* $a\ b$
⟨*proof*⟩

lemma *main-real*:
assumes $b < a$
shows *valid-countings* $a\ b = (a - b)\ /\ (a + b) *$ *all-countings* $a\ b$
⟨*proof*⟩

lemma
valid-countings $a\ b =$ (*if* $a \leq b$ *then* (*if* $b = 0$ *then* 1 *else* 0) *else* $(a - b)\ /\ (a + b) *$ *all-countings* $a\ b$)
⟨*proof*⟩

97.4.1 Executable Definition

declare *all-countings-def* [*code del*]
declare *all-countings*[*code*]

value *all-countings 1 0*
value *all-countings 0 1*
value *all-countings 1 1*
value *all-countings 2 1*
value *all-countings 1 2*
value *all-countings 2 4*
value *all-countings 4 2*

97.4.2 Executable Definition

declare *valid-countings-def* [*code del*]

value *valid-countings 1 0*
value *valid-countings 0 1*
value *valid-countings 1 1*
value *valid-countings 2 1*
value *valid-countings 1 2*
value *valid-countings 2 4*
value *valid-countings 4 2*

end

98 The Erdoes-Szekeres Theorem

theory *Erdoes-Szekeres*
imports *Main*
begin

98.1 Addition to *Lattices-Big* Theory

lemma *Max-gr*:
  assumes *finite A*
  assumes $a \in A$ $a > x$
  shows $x < Max\ A$
⟨*proof*⟩

98.2 Additions to *Finite-Set* Theory

lemma *obtain-subset-with-card-n*:
  assumes $n \leq card\ S$
  obtains T **where** $T \subseteq S$ *card* $T = n$
⟨*proof*⟩

lemma *exists-set-with-max-card*:
  assumes *finite S* $S \neq \{\}$
  shows $\exists s \in S.\ card\ s = Max\ (card\ `\ S)$
⟨*proof*⟩

98.3 Definition of Monotonicity over a Carrier Set

definition
mono-on f R S = $(\forall i{\in}S.\ \forall j{\in}S.\ i \leq j \longrightarrow R\ (f\ i)\ (f\ j))$

lemma *mono-on-empty* [*simp*]: *mono-on f R* {}
⟨*proof*⟩

lemma *mono-on-singleton* [*simp*]: *reflp R* ⟹ *mono-on f R* {*x*}
⟨*proof*⟩

lemma *mono-on-subset*: *T* ⊆ *S* ⟹ *mono-on f R S* ⟹ *mono-on f R T*
⟨*proof*⟩

lemma *not-mono-on-subset*: *T* ⊆ *S* ⟹ ¬ *mono-on f R T* ⟹ ¬ *mono-on f R S*
⟨*proof*⟩

lemma [*simp*]:
reflp (*op* ≤ :: *'a*::*order* ⇒ - ⇒ *bool*)
reflp (*op* ≥ :: *'a*::*order* ⇒ - ⇒ *bool*)
transp (*op* ≤ :: *'a*::*order* ⇒ - ⇒ *bool*)
transp (*op* ≥ :: *'a*::*order* ⇒ - ⇒ *bool*)
⟨*proof*⟩

98.4 The Erdoes-Szekeres Theorem following Seidenberg's (1959) argument

lemma *Erdoes-Szekeres*:
fixes *f* :: - ⇒ *'a*::*linorder*
shows (∃ *S*. *S* ⊆ {*0*..*m* ∗ *n*} ∧ *card S* = *m* + *1* ∧ *mono-on f* (*op* ≤) *S*) ∨
(∃ *S*. *S* ⊆ {*0*..*m* ∗ *n*} ∧ *card S* = *n* + *1* ∧ *mono-on f* (*op* ≥) *S*)
⟨*proof*⟩

end

99 Sum of Powers

theory *Sum-of-Powers*
imports *Complex-Main*
begin

99.1 Additions to *Binomial* Theory

lemma (**in** *field-char-0*) *one-plus-of-nat-neq-zero* [*simp*]:
1 + *of-nat n* ≠ *0*
⟨*proof*⟩

lemma *of-nat-binomial-eq-mult-binomial-Suc*:
assumes *k* ≤ *n*

shows (*of-nat* :: (*nat* ⇒ ('*a* :: *field-char-0*))) (*n choose k*) = *of-nat* (*n* + *1* − *k*) / *of-nat* (*n* + *1*) ∗ *of-nat* (*Suc n choose k*)
⟨*proof*⟩

lemma *real-binomial-eq-mult-binomial-Suc*:
assumes *k* ≤ *n*
shows (*n choose k*) = (*n* + *1* − *k*) / (*n* + *1*) ∗ (*Suc n choose k*)
⟨*proof*⟩

99.2 Preliminaries

lemma *integrals-eq*:
assumes *f 0* = *g 0*
assumes ⋀ *x*. ((λ*x*. *f x* − *g x*) *has-real-derivative 0*) (*at x*)
shows *f x* = *g x*
⟨*proof*⟩

lemma *sum-diff*: ((∑ *i*≤*n*::*nat*. *f* (*i* + *1*) − *f i*)::'*a*::*field*) = *f* (*n* + *1*) − *f 0*
⟨*proof*⟩

declare *One-nat-def* [*simp del*]

99.3 Bernoulli Numbers and Bernoulli Polynomials

declare *sum.cong* [*fundef-cong*]

fun *bernoulli* :: *nat* ⇒ *real*
where
bernoulli 0 = (*1*::*real*)
| *bernoulli* (*Suc n*) = (−*1* / (*n* + *2*)) ∗ (∑ *k* ≤ *n*. ((*n* + *2 choose k*) ∗ *bernoulli k*))

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

definition
bernpoly n = (λ*x*. ∑ *k* ≤ *n*. (*n choose k*) ∗ *bernoulli k* ∗ *x* ^ (*n* − *k*))

99.4 Basic Observations on Bernoulli Polynomials

lemma *bernpoly-0*: *bernpoly n 0* = *bernoulli n*
⟨*proof*⟩

lemma *sum-binomial-times-bernoulli*:
(∑ *k*≤*n*. ((*Suc n*) *choose k*) ∗ *bernoulli k*) = (*if n* = *0 then 1 else 0*)
⟨*proof*⟩

99.5 Sum of Powers with Bernoulli Polynomials

lemma *bernpoly-derivative* [*derivative-intros*]:
(*bernpoly* (*Suc n*) *has-real-derivative* ((*n* + *1*) ∗ *bernpoly n x*)) (*at x*)

⟨*proof*⟩

lemma *diff-bernpoly*:
*bernpoly n (x + 1) − bernpoly n x = n * x ^ (n − 1)*
⟨*proof*⟩

lemma *sum-of-powers*: *(∑k≤n::nat. (real k) ^ m) = (bernpoly (Suc m) (n + 1) − bernpoly (Suc m) 0) / (m + 1)*
⟨*proof*⟩

99.6 Instances for Square And Cubic Numbers

lemma *binomial-unroll*:
n > 0 ⟹ (n choose k) = (if k = 0 then 1 else (n − 1) choose (k − 1) + ((n − 1) choose k))
⟨*proof*⟩

lemma *sum-unroll*:
(∑k≤n::nat. f k) = (if n = 0 then f 0 else f n + (∑k≤n − 1. f k))
⟨*proof*⟩

lemma *bernoulli-unroll*:
*n > 0 ⟹ bernoulli n = − 1 / (real n + 1) * (∑k≤n − 1. real (n + 1 choose k) * bernoulli k)*
⟨*proof*⟩

lemmas *unroll = binomial-unroll*
bernoulli.simps(1) bernoulli-unroll sum-unroll bernpoly-def

lemma *sum-of-squares*: *(∑k≤n::nat. k ^ 2) = (2 * n ^ 3 + 3 * n ^ 2 + n) / 6*
⟨*proof*⟩

lemma *sum-of-squares-nat*: *(∑k≤n::nat. k ^ 2) = (2 * n ^ 3 + 3 * n ^ 2 + n) div 6*
⟨*proof*⟩

lemma *sum-of-cubes*: *(∑k≤n::nat. k ^ 3) = (n ^ 2 + n) ^ 2 / 4*
⟨*proof*⟩

lemma *sum-of-cubes-nat*: *(∑k≤n::nat. k ^ 3) = (n ^ 2 + n) ^ 2 div 4*
⟨*proof*⟩

end

100 A SAT-based Sudoku Solver

theory *Sudoku*
imports *Main*
begin

See the paper “A SAT-based Sudoku Solver” (Tjark Weber, published at LPAR’05) for further explanations. (The paper describes an older version of this theory that used the model finder *refute* to find Sudoku solutions. The *refute* tool has since been superseded by *nitpick*, which is used below.)

no-notation *Groups.one-class.one* (*1*)

datatype *digit* = *A* (*1*) | *B* (*2*) | *C* (*3*) | *D* (*4*) | *E* (*5*) | *F* (*6*) | *G* (*7*) | *H* (*8*) | *I* (*9*)

definition *valid* :: *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *bool* **where**

valid x1 x2 x3 x4 x5 x6 x7 x8 x9 ==
$(x1 \neq x2) \wedge (x1 \neq x3) \wedge (x1 \neq x4) \wedge (x1 \neq x5) \wedge (x1 \neq x6) \wedge (x1 \neq x7) \wedge (x1 \neq x8) \wedge (x1 \neq x9)$
$\wedge (x2 \neq x3) \wedge (x2 \neq x4) \wedge (x2 \neq x5) \wedge (x2 \neq x6) \wedge (x2 \neq x7) \wedge (x2 \neq x8) \wedge (x2 \neq x9)$
$\wedge (x3 \neq x4) \wedge (x3 \neq x5) \wedge (x3 \neq x6) \wedge (x3 \neq x7) \wedge (x3 \neq x8) \wedge (x3 \neq x9)$
$\wedge (x4 \neq x5) \wedge (x4 \neq x6) \wedge (x4 \neq x7) \wedge (x4 \neq x8) \wedge (x4 \neq x9)$
$\wedge (x5 \neq x6) \wedge (x5 \neq x7) \wedge (x5 \neq x8) \wedge (x5 \neq x9)$
$\wedge (x6 \neq x7) \wedge (x6 \neq x8) \wedge (x6 \neq x9)$
$\wedge (x7 \neq x8) \wedge (x7 \neq x9)$
$\wedge (x8 \neq x9)$

definition *sudoku* :: *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* =>
digit => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* =>
digit => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* =>
digit => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* =>
digit => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* =>
digit => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* =>
digit => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* =>
digit => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* =>
digit => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *digit* => *bool* **where**

sudoku x11 x12 x13 x14 x15 x16 x17 x18 x19
x21 x22 x23 x24 x25 x26 x27 x28 x29
x31 x32 x33 x34 x35 x36 x37 x38 x39
x41 x42 x43 x44 x45 x46 x47 x48 x49
x51 x52 x53 x54 x55 x56 x57 x58 x59
x61 x62 x63 x64 x65 x66 x67 x68 x69

x71 x72 x73 x74 x75 x76 x77 x78 x79
x81 x82 x83 x84 x85 x86 x87 x88 x89
x91 x92 x93 x94 x95 x96 x97 x98 x99 ==

valid x11 x12 x13 x14 x15 x16 x17 x18 x19
∧ *valid x21 x22 x23 x24 x25 x26 x27 x28 x29*
∧ *valid x31 x32 x33 x34 x35 x36 x37 x38 x39*
∧ *valid x41 x42 x43 x44 x45 x46 x47 x48 x49*
∧ *valid x51 x52 x53 x54 x55 x56 x57 x58 x59*
∧ *valid x61 x62 x63 x64 x65 x66 x67 x68 x69*
∧ *valid x71 x72 x73 x74 x75 x76 x77 x78 x79*
∧ *valid x81 x82 x83 x84 x85 x86 x87 x88 x89*
∧ *valid x91 x92 x93 x94 x95 x96 x97 x98 x99*

∧ *valid x11 x21 x31 x41 x51 x61 x71 x81 x91*
∧ *valid x12 x22 x32 x42 x52 x62 x72 x82 x92*
∧ *valid x13 x23 x33 x43 x53 x63 x73 x83 x93*
∧ *valid x14 x24 x34 x44 x54 x64 x74 x84 x94*
∧ *valid x15 x25 x35 x45 x55 x65 x75 x85 x95*
∧ *valid x16 x26 x36 x46 x56 x66 x76 x86 x96*
∧ *valid x17 x27 x37 x47 x57 x67 x77 x87 x97*
∧ *valid x18 x28 x38 x48 x58 x68 x78 x88 x98*
∧ *valid x19 x29 x39 x49 x59 x69 x79 x89 x99*

∧ *valid x11 x12 x13 x21 x22 x23 x31 x32 x33*
∧ *valid x14 x15 x16 x24 x25 x26 x34 x35 x36*
∧ *valid x17 x18 x19 x27 x28 x29 x37 x38 x39*
∧ *valid x41 x42 x43 x51 x52 x53 x61 x62 x63*
∧ *valid x44 x45 x46 x54 x55 x56 x64 x65 x66*
∧ *valid x47 x48 x49 x57 x58 x59 x67 x68 x69*
∧ *valid x71 x72 x73 x81 x82 x83 x91 x92 x93*
∧ *valid x74 x75 x76 x84 x85 x86 x94 x95 x96*
∧ *valid x77 x78 x79 x87 x88 x89 x97 x98 x99*

Just an arbitrary Sudoku grid:

theorem ¬ *sudoku*
x11 x12 x13 x14 x15 x16 x17 x18 x19
x21 x22 x23 x24 x25 x26 x27 x28 x29
x31 x32 x33 x34 x35 x36 x37 x38 x39
x41 x42 x43 x44 x45 x46 x47 x48 x49
x51 x52 x53 x54 x55 x56 x57 x58 x59
x61 x62 x63 x64 x65 x66 x67 x68 x69
x71 x72 x73 x74 x75 x76 x77 x78 x79
x81 x82 x83 x84 x85 x86 x87 x88 x89
x91 x92 x93 x94 x95 x96 x97 x98 x99
nitpick [*expect*=*genuine*]
⟨*proof*⟩

An “easy” Sudoku:

theorem ¬ *sudoku*
5 3 x13 x14 7 x16 x17 x18 x19
6 x22 x23 1 9 5 x27 x28 x29
x31 9 8 x34 x35 x36 x37 6 x39
8 x42 x43 x44 6 x46 x47 x48 3
4 x52 x53 8 x55 3 x57 x58 1
7 x62 x63 x64 2 x66 x67 x68 6
x71 6 x73 x74 x75 x76 2 8 x79
x81 x82 x83 4 1 9 x87 x88 5
x91 x92 x93 x94 8 x96 x97 7 9
nitpick [*expect*=*genuine*]
⟨*proof*⟩

A "hard" Sudoku:

theorem ¬ *sudoku*
x11 2 x13 x14 x15 x16 x17 x18 x19
x21 x22 x23 6 x25 x26 x27 x28 3
x31 7 4 x34 8 x36 x37 x38 x39
x41 x42 x43 x44 x45 3 x47 x48 2
x51 8 x53 x54 4 x56 x57 1 x59
6 x62 x63 5 x65 x66 x67 x68 x69
x71 x72 x73 x74 1 x76 7 8 x79
5 x82 x83 x84 x85 9 x87 x88 x89
x91 x92 x93 x94 x95 x96 x97 4 x99
nitpick [*expect*=*genuine*]
⟨*proof*⟩

Some "exceptionally difficult" Sudokus, taken from http://en.wikipedia.org/w/index.php?title=Algorithmics_of_sudoku&oldid=254685903 (accessed December 2, 2008).

```
Rating Program: gsf's sudoku q1 (rating)
Rating: 99408
Poster: JPF
Label: Easter Monster
1.......2.9.4...5...6...7...5.9.3.......7.......85..4.7.....6...3...9.8...2.....1
1 . . | . . . | . . 2
. 9 . | 4 . . | . 5 .
. . 6 | . . . | 7 . .
------+-------+------
. 5 . | 9 . 3 | . . .
. . . | . 7 . | . . .
. . . | 8 5 . | . 4 .
------+-------+------
7 . . | . . . | 6 . .
. 3 . | . . 9 | . 8 .
. . 2 | . . . | . . 1
```

theorem ¬ *sudoku*
1 x12 x13 x14 x15 x16 x17 x18 2
x21 9 x23 4 x25 x26 x27 5 x29
x31 x32 6 x34 x35 x36 7 x38 x39
x41 5 x43 9 x45 3 x47 x48 x49
x51 x52 x53 x54 7 x56 x57 x58 x59
x61 x62 x63 8 5 x66 x67 4 x69
7 x72 x73 x74 x75 x76 6 x78 x79
x81 3 x83 x84 x85 9 x87 8 x89
x91 x92 2 x94 x95 x96 x97 x98 1
nitpick [*expect*=*genuine*]
⟨*proof*⟩

```
Rating Program: gsf's sudoku q1 (Processing time)
Rating: 4m19s@2 GHz
Poster: tarek
Label: tarek071223170000-052
..1..4.......6.3.5...9.....8.....7.3.......285...7.6..3...8...6..92......4...1...
. . 1 | . . 4 | . . .
. . . | . 6 . | 3 . 5
. . . | 9 . . | . . .
------+-------+------
8 . . | . . . | 7 . 3
. . . | . . . | . 2 8
5 . . | . 7 . | 6 . .
------+-------+------
3 . . | . 8 . | . . 6
. . 9 | 2 . . | . . .
. 4 . | . . 1 | . . .
```

theorem ¬ *sudoku*
x11 x12 1 x14 x15 4 x17 x18 x19
x21 x22 x23 x24 6 x26 3 x28 5
x31 x32 x33 9 x35 x36 x37 x38 x39
8 x42 x43 x44 x45 x46 7 x48 3
x51 x52 x53 x54 x55 x56 x57 2 8
5 x62 x63 x64 7 x66 6 x68 x69
3 x72 x73 x74 8 x76 x77 x78 6
x81 x82 9 2 x85 x86 x87 x88 x89
x91 4 x93 x94 x95 1 x97 x98 x99
nitpick [*expect*=*genuine*]
⟨*proof*⟩

```
Rating Program: Nicolas Juillerat's Sudoku explainer 1.2.1
Rating: 11.9
```

Poster: tarek
Label: golden nugget
.....39.....1..5..3.5.8....8.9...6.7...2...1..4.....9.8..5..2....6..4..7.....

```
. . . | . . . | . 3 9
. . . | . . 1 | . . 5
. . 3 | . 5 . | 8 . .
------+-------+------
. . 8 | . 9 . | . . 6
. 7 . | . . 2 | . . .
1 . . | 4 . . | . . .
------+-------+------
. . 9 | . 8 . | . 5 .
. 2 . | . . . | 6 . .
4 . . | 7 . . | . . .
```

theorem ¬ *sudoku*
x11 x12 x13 x14 x15 x16 x17 3 9
x21 x22 x23 x24 x25 1 x27 x28 5
x31 x32 3 x34 5 x36 8 x38 x39
x41 x42 8 x44 9 x46 x47 x48 6
x51 7 x53 x54 x55 2 x57 x58 x59
1 x62 x63 4 x65 x66 x67 x68 x69
x71 x72 9 x74 8 x76 x77 5 x79
x81 2 x83 x84 x85 x86 6 x88 x89
4 x92 x93 7 x95 x96 x97 x98 x99
nitpick [*expect*=*genuine*]
⟨*proof*⟩

Rating Program: dukuso's suexrat9
Rating: 4483
Poster: coloin
Label: col-02-08-071
.2.4.37.....32.....4.4.2...7.8...5.....1...5.....9...3.9....7..1..86..

```
. 2 . | 4 . 3 | 7 . .
. . . | . . . | . 3 2
. . . | . . . | . . 4
------+-------+------
. 4 . | 2 . . | . 7 .
8 . . | . 5 . | . . .
. . . | . . 1 | . . .
------+-------+------
5 . . | . . . | 9 . .
. 3 . | 9 . . | . . 7
. . 1 | . . 8 | 6 . .
```

theorem ¬ *sudoku*
x11 2 x13 4 x15 3 7 x18 x19
x21 x22 x23 x24 x25 x26 x27 3 2
x31 x32 x33 x34 x35 x36 x37 x38 4
x41 4 x43 2 x45 x46 x47 7 x49
8 x52 x53 x54 5 x56 x57 x58 x59
x61 x62 x63 x64 x65 1 x67 x68 x69
5 x72 x73 x74 x75 x76 9 x78 x79
x81 3 x83 9 x85 x86 x87 x88 7
x91 x92 1 x94 x95 8 6 x98 x99
nitpick [*expect*=*genuine*]
⟨*proof*⟩

```
Rating Program: dukuso's suexratt (10000 2 option)
Rating: 2141
Poster: tarek
Label: golden nugget
.......39.....1..5..3.5.8....8.9...6.7...2...1..4.......9.8..5..2....6..4..7.....
. . . | . . . | . 3 9
. . . | . . 1 | . . 5
. . 3 | . 5 . | 8 . .
------+-------+------
. . 8 | . 9 . | . . 6
. 7 . | . . 2 | . . .
1 . . | 4 . . | . . .
------+-------+------
. . 9 | . 8 . | . 5 .
. 2 . | . . . | 6 . .
4 . . | 7 . . | . . .
```

theorem ¬ *sudoku*
x11 x12 x13 x14 x15 x16 x17 3 9
x21 x22 x23 x24 x25 1 x27 x28 5
x31 x32 3 x34 5 x36 8 x38 x39
x41 x42 8 x44 9 x46 x47 x48 6
x51 7 x53 x54 x55 2 x57 x58 x59
1 x62 x63 4 x65 x66 x67 x68 x69
x71 x72 9 x74 8 x76 x77 5 x79
x81 2 x83 x84 x85 x86 6 x88 x89
4 x92 x93 7 x95 x96 x97 x98 x99
nitpick [*expect*=*genuine*]
⟨*proof*⟩

end

101 The sieve of Eratosthenes

theory *Eratosthenes*
imports *Main Primes*
begin

101.1 Preliminary: strict divisibility

context *dvd*
begin

abbreviation *dvd-strict* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** *dvd'-strict 50*)
where
b *dvd-strict* $a \equiv b$ *dvd* $a \wedge \neg\ a$ *dvd* b

end

101.2 Main corpus

The sieve is modelled as a list of booleans, where *False* means *marked out*.

type-synonym *marks* = *bool list*

definition *numbers-of-marks* :: *nat* ⇒ *marks* ⇒ *nat set*
where
numbers-of-marks n bs = *fst* ‘ {$x \in$ *set* (*enumerate n bs*). *snd x*}

lemma *numbers-of-marks-simps* [*simp*, *code*]:
numbers-of-marks n [] = {}
numbers-of-marks n (*True* # *bs*) = *insert n* (*numbers-of-marks* (*Suc n*) *bs*)
numbers-of-marks n (*False* # *bs*) = *numbers-of-marks* (*Suc n*) *bs*
⟨*proof*⟩

lemma *numbers-of-marks-Suc*:
numbers-of-marks (*Suc n*) *bs* = *Suc* ‘ *numbers-of-marks n bs*
⟨*proof*⟩

lemma *numbers-of-marks-replicate-False* [*simp*]:
numbers-of-marks n (*replicate m False*) = {}
⟨*proof*⟩

lemma *numbers-of-marks-replicate-True* [*simp*]:
numbers-of-marks n (*replicate m True*) = {*n*..<*n*+*m*}
⟨*proof*⟩

lemma *in-numbers-of-marks-eq*:
$m \in$ *numbers-of-marks n bs* $\longleftrightarrow$ $m \in$ {*n*..<*n* + *length bs*} $\wedge$ *bs* ! (m − *n*)
⟨*proof*⟩

lemma *sorted-list-of-set-numbers-of-marks*:

sorted-list-of-set (*numbers-of-marks n bs*) = *map fst* (*filter snd* (*enumerate n bs*))
⟨*proof*⟩

Marking out multiples in a sieve

definition *mark-out* :: *nat* ⇒ *marks* ⇒ *marks*
where
mark-out n bs = *map* (λ(*q*, *b*). *b* ∧ ¬ *Suc n dvd Suc* (*Suc q*)) (*enumerate n bs*)

lemma *mark-out-Nil* [*simp*]: *mark-out n* [] = []
⟨*proof*⟩

lemma *length-mark-out* [*simp*]: *length* (*mark-out n bs*) = *length bs*
⟨*proof*⟩

lemma *numbers-of-marks-mark-out*:
numbers-of-marks n (*mark-out m bs*) = {*q* ∈ *numbers-of-marks n bs*. ¬ *Suc m dvd Suc q* − *n*}
⟨*proof*⟩

Auxiliary operation for efficient implementation

definition *mark-out-aux* :: *nat* ⇒ *nat* ⇒ *marks* ⇒ *marks*
where
mark-out-aux n m bs =
map (λ(*q*, *b*). *b* ∧ (*q* < *m* + *n* ∨ ¬ *Suc n dvd Suc* (*Suc q*) + (*n* − *m mod Suc n*))) (*enumerate n bs*)

lemma *mark-out-code* [*code*]: *mark-out n bs* = *mark-out-aux n n bs*
⟨*proof*⟩

lemma *mark-out-aux-simps* [*simp*, *code*]:
mark-out-aux n m [] = []
mark-out-aux n 0 (*b* # *bs*) = *False* # *mark-out-aux n n bs*
mark-out-aux n (*Suc m*) (*b* # *bs*) = *b* # *mark-out-aux n m bs*
⟨*proof*⟩

Main entry point to sieve

fun *sieve* :: *nat* ⇒ *marks* ⇒ *marks*
where
sieve n [] = []
| *sieve n* (*False* # *bs*) = *False* # *sieve* (*Suc n*) *bs*
| *sieve n* (*True* # *bs*) = *True* # *sieve* (*Suc n*) (*mark-out n bs*)

There are the following possible optimisations here:

- *sieve* can abort as soon as *n* is too big to let *mark-out* have any effect.
- Search for further primes can be given up as soon as the search position exceeds the square root of the maximum candidate.

This is left as an constructive exercise to the reader.

lemma *numbers-of-marks-sieve*:
numbers-of-marks (*Suc n*) (*sieve n bs*) =
{*q* ∈ *numbers-of-marks* (*Suc n*) *bs*. ∀ *m* ∈ *numbers-of-marks* (*Suc n*) *bs*. ¬ *m dvd-strict q*}
⟨*proof*⟩

Relation of the sieve algorithm to actual primes

definition *primes-upto* :: *nat* ⇒ *nat list*
where
primes-upto n = *sorted-list-of-set* {*m*. *m* ≤ *n* ∧ *prime m*}

lemma *set-primes-upto*: *set* (*primes-upto n*) = {*m*. *m* ≤ *n* ∧ *prime m*}
⟨*proof*⟩

lemma *sorted-primes-upto* [*iff*]: *sorted* (*primes-upto n*)
⟨*proof*⟩

lemma *distinct-primes-upto* [*iff*]: *distinct* (*primes-upto n*)
⟨*proof*⟩

lemma *set-primes-upto-sieve*:
set (*primes-upto n*) = *numbers-of-marks 2* (*sieve 1* (*replicate* (*n* − *1*) *True*))
⟨*proof*⟩

lemma *primes-upto-sieve* [*code*]:
primes-upto n = *map fst* (*filter snd* (*enumerate 2* (*sieve 1* (*replicate* (*n* − *1*) *True*))))
⟨*proof*⟩

lemma *prime-in-primes-upto*: *prime n* ⟷ *n* ∈ *set* (*primes-upto n*)
⟨*proof*⟩

101.3 Application: smallest prime beyond a certain number

definition *smallest-prime-beyond* :: *nat* ⇒ *nat*
where
smallest-prime-beyond n = (*LEAST p*. *prime p* ∧ *p* ≥ *n*)

lemma *prime-smallest-prime-beyond* [*iff*]: *prime* (*smallest-prime-beyond n*) (**is** *?P*)
and *smallest-prime-beyond-le* [*iff*]: *smallest-prime-beyond n* ≥ *n* (**is** *?Q*)
⟨*proof*⟩

lemma *smallest-prime-beyond-smallest*: *prime p* ⟹ *p* ≥ *n* ⟹ *smallest-prime-beyond n* ≤ *p*
⟨*proof*⟩

lemma *smallest-prime-beyond-eq*:

prime p $\Longrightarrow$ *p* $\geq$ *n* $\Longrightarrow$ ($\bigwedge$*q*. *prime q* $\Longrightarrow$ *q* $\geq$ *n* $\Longrightarrow$ *q* $\geq$ *p*) $\Longrightarrow$ *smallest-prime-beyond n* = *p*
⟨*proof*⟩

definition *smallest-prime-between* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat option*
where
smallest-prime-between m n =
(*if* ($\exists$ *p*. *prime p* $\wedge$ *m* $\leq$ *p* $\wedge$ *p* $\leq$ *n*) *then Some* (*smallest-prime-beyond m*) *else None*)

lemma *smallest-prime-between-None*:
smallest-prime-between m n = *None* $\longleftrightarrow$ ($\forall$ *q*. *m* $\leq$ *q* $\wedge$ *q* $\leq$ *n* $\longrightarrow$ $\neg$ *prime q*)
⟨*proof*⟩

lemma *smallest-prime-betwen-Some*:
smallest-prime-between m n = *Some p* $\longleftrightarrow$ *smallest-prime-beyond m* = *p* $\wedge$ *p* $\leq$ *n*
⟨*proof*⟩

lemma [*code*]: *smallest-prime-between m n* = *List.find* (λ*p*. *p* $\geq$ *m*) (*primes-upto n*)
⟨*proof*⟩

definition *smallest-prime-beyond-aux* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat*
where
smallest-prime-beyond-aux k n = *smallest-prime-beyond n*

lemma [*code*]:
smallest-prime-beyond-aux k n =
(*case smallest-prime-between n* (*k* $*$ *n*) *of*
Some p $\Rightarrow$ *p*
| *None* $\Rightarrow$ *smallest-prime-beyond-aux* (*Suc k*) *n*)
⟨*proof*⟩

lemma [*code*]: *smallest-prime-beyond n* = *smallest-prime-beyond-aux 2 n*
⟨*proof*⟩

end

102 Examples for code generation timing measures

theory *Code-Timing*
imports ~~/*src*/*HOL*/*Number-Theory*/*Eratosthenes*
begin

declare [[*code-timing*]]

definition *primes-upto* :: *nat* $\Rightarrow$ *int list*
where

primes-upto = *map int* ∘ *Eratosthenes.primes-upto*

definition *required-symbols* - = (*primes-upto*, *0* :: *nat*, *Suc*, *1* :: *nat*,
numeral :: *num* ⇒ *nat*, *Num.One*, *Num.Bit0*, *Num.Bit1*,
Code-Evaluation.TERM-OF-EQUAL :: *int list itself*)

⟨*ML*⟩

end

103 Permutations as abstract type

theory *Perm*
imports *Main*
begin

This theory introduces basics about permutations, i.e. almost everywhere fix bijections. But it is by no means complete. Grieviously missing are cycles since these would require more elaboration, e.g. the concept of distinct lists equivalent under rotation, which maybe would also deserve its own theory. But see theory *src/HOL/ex/Perm-Fragments.thy* for fragments on that.

103.1 Abstract type of permutations

typedef *'a perm* = {*f* :: *'a* ⇒ *'a*. *bij f* ∧ *finite* {*a*. *f a* ≠ *a*}}
morphisms *apply Perm*
⟨*proof*⟩

setup-lifting *type-definition-perm*

notation *apply* (**infixl** ⟨$⟩ *999*)
no-notation *apply* (*op* ⟨$⟩)

lemma *bij-apply* [*simp*]:
bij (*apply f*)
⟨*proof*⟩

lemma *perm-eqI*:
assumes ⋀*a*. *f* ⟨$⟩ *a* = *g* ⟨$⟩ *a*
shows *f* = *g*
⟨*proof*⟩

lemma *perm-eq-iff*:
f = *g* ⟷ (∀ *a*. *f* ⟨$⟩ *a* = *g* ⟨$⟩ *a*)
⟨*proof*⟩

lemma *apply-inj*:
f ⟨$⟩ *a* = *f* ⟨$⟩ *b* ⟷ *a* = *b*

⟨*proof*⟩

lift-definition *affected* :: *'a perm* ⇒ *'a set*
is λ*f*. {*a*. *f a* ≠ *a*} ⟨*proof*⟩

lemma *in-affected*:
a ∈ *affected f* ⟷ *f* ⟨$⟩ *a* ≠ *a*
⟨*proof*⟩

lemma *finite-affected* [*simp*]:
finite (*affected f*)
⟨*proof*⟩

lemma *apply-affected* [*simp*]:
f ⟨$⟩ *a* ∈ *affected f* ⟷ *a* ∈ *affected f*
⟨*proof*⟩

lemma *card-affected-not-one*:
card (*affected f*) ≠ *1*
⟨*proof*⟩

103.2 Identity, composition and inversion

instantiation *Perm.perm* :: (*type*) {*monoid-mult*, *inverse*}
begin

lift-definition *one-perm* :: *'a perm*
is *id*
⟨*proof*⟩

lemma *apply-one* [*simp*]:
apply 1 = *id*
⟨*proof*⟩

lemma *affected-one* [*simp*]:
affected 1 = {}
⟨*proof*⟩

lemma *affected-empty-iff* [*simp*]:
affected f = {} ⟷ *f* = *1*
⟨*proof*⟩

lift-definition *times-perm* :: *'a perm* ⇒ *'a perm* ⇒ *'a perm*
is *comp*
⟨*proof*⟩

lemma *apply-times*:
apply (*f* ∗ *g*) = *apply f* ∘ *apply g*
⟨*proof*⟩

lemma *apply-sequence*:
  $f \langle\$\rangle (g \langle\$\rangle a) = apply (f * g) a$
  ⟨*proof*⟩

lemma *affected-times* [*simp*]:
  $affected (f * g) \subseteq affected f \cup affected g$
  ⟨*proof*⟩

lift-definition *inverse-perm* :: $'a\ perm \Rightarrow 'a\ perm$
  is *inv*
⟨*proof*⟩

instance
  ⟨*proof*⟩

end

lemma *apply-inverse*:
  $apply (inverse f) = inv (apply f)$
  ⟨*proof*⟩

lemma *affected-inverse* [*simp*]:
  $affected (inverse f) = affected f$
⟨*proof*⟩

global-interpretation *perm*: *group times 1*::$'a\ perm\ inverse$
⟨*proof*⟩

declare *perm.inverse-distrib-swap* [*simp*]

lemma *perm-mult-commute*:
  assumes $affected f \cap affected g = \{\}$
  shows $g * f = f * g$
⟨*proof*⟩

lemma *apply-power*:
  $apply (f \hat{}\ n) = apply f \hat{}\hat{}\ n$
  ⟨*proof*⟩

lemma *perm-power-inverse*:
  $inverse f \hat{}\ n = inverse ((f :: 'a\ perm) \hat{}\ n)$
⟨*proof*⟩

103.3 Orbit and order of elements

definition *orbit* :: $'a\ perm \Rightarrow 'a \Rightarrow 'a\ set$
where
  $orbit f a = range (\lambda n.\ (f \hat{}\ n) \langle\$\rangle a)$

lemma *in-orbitI*:
assumes $(f \hat{} n)\ \langle\$\rangle\ a = b$
shows $b \in \mathit{orbit}\ f\ a$
⟨*proof*⟩

lemma *apply-power-self-in-orbit* [*simp*]:
$(f \hat{} n)\ \langle\$\rangle\ a \in \mathit{orbit}\ f\ a$
⟨*proof*⟩

lemma *in-orbit-self* [*simp*]:
$a \in \mathit{orbit}\ f\ a$
⟨*proof*⟩

lemma *apply-self-in-orbit* [*simp*]:
$f\ \langle\$\rangle\ a \in \mathit{orbit}\ f\ a$
⟨*proof*⟩

lemma *orbit-not-empty* [*simp*]:
$\mathit{orbit}\ f\ a \neq \{\}$
⟨*proof*⟩

lemma *not-in-affected-iff-orbit-eq-singleton*:
$a \notin \mathit{affected}\ f \longleftrightarrow \mathit{orbit}\ f\ a = \{a\}$ (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

definition *order* :: $'a\ \mathit{perm} \Rightarrow 'a \Rightarrow \mathit{nat}$
where
$\mathit{order}\ f = \mathit{card} \circ \mathit{orbit}\ f$

lemma *orbit-subset-eq-affected*:
assumes $a \in \mathit{affected}\ f$
shows $\mathit{orbit}\ f\ a \subseteq \mathit{affected}\ f$
⟨*proof*⟩

lemma *finite-orbit* [*simp*]:
finite $(\mathit{orbit}\ f\ a)$
⟨*proof*⟩

lemma *orbit-1* [*simp*]:
$\mathit{orbit}\ 1\ a = \{a\}$
⟨*proof*⟩

lemma *order-1* [*simp*]:
$\mathit{order}\ 1\ a = 1$
⟨*proof*⟩

lemma *card-orbit-eq* [*simp*]:
$\mathit{card}\ (\mathit{orbit}\ f\ a) = \mathit{order}\ f\ a$

⟨*proof*⟩

lemma *order-greater-zero* [*simp*]:
  order f a > *0*
  ⟨*proof*⟩

lemma *order-eq-one-iff*:
  order f a = *Suc 0* ⟷ *a* ∉ *affected f* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *order-greater-eq-two-iff*:
  order f a ≥ *2* ⟷ *a* ∈ *affected f*
  ⟨*proof*⟩

lemma *order-less-eq-affected*:
  assumes *f* ≠ *1*
  shows *order f a* ≤ *card* (*affected f*)
⟨*proof*⟩

lemma *affected-order-greater-eq-two*:
  assumes *a* ∈ *affected f*
  shows *order f a* ≥ *2*
⟨*proof*⟩

lemma *order-witness-unfold*:
  assumes *n* > *0* **and** (*f* ^ *n*) ⟨$⟩ *a* = *a*
  shows *order f a* = *card* ((λ*m*. (*f* ^ *m*) ⟨$⟩ *a*) ‘ {*0*..<*n*})
⟨*proof*⟩

lemma *inj-on-apply-range*:
  inj-on (λ*m*. (*f* ^ *m*) ⟨$⟩ *a*) {..<*order f a*}
⟨*proof*⟩

lemma *orbit-unfold-image*:
  orbit f a = (λ*n*. (*f* ^ *n*) ⟨$⟩ *a*) ‘ {..<*order f a*} (**is** - = *?A*)
⟨*proof*⟩

lemma *in-orbitE*:
  assumes *b* ∈ *orbit f a*
  obtains *n* **where** *b* = (*f* ^ *n*) ⟨$⟩ *a* **and** *n* < *order f a*
  ⟨*proof*⟩

lemma *apply-power-order* [*simp*]:
  (*f* ^ *order f a*) ⟨$⟩ *a* = *a*
⟨*proof*⟩

lemma *apply-power-left-mult-order* [*simp*]:
  (*f* ^ (*n* ∗ *order f a*)) ⟨$⟩ *a* = *a*
  ⟨*proof*⟩

lemma *apply-power-right-mult-order* [*simp*]:
(*f* ^ (*order f a* ∗ *n*)) ⟨$⟩ *a* = *a*
⟨*proof*⟩

lemma *apply-power-mod-order-eq* [*simp*]:
(*f* ^ (*n mod order f a*)) ⟨$⟩ *a* = (*f* ^ *n*) ⟨$⟩ *a*
⟨*proof*⟩

lemma *apply-power-eq-iff*:
(*f* ^ *m*) ⟨$⟩ *a* = (*f* ^ *n*) ⟨$⟩ *a* ⟷ *m mod order f a* = *n mod order f a* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *apply-inverse-eq-apply-power-order-minus-one*:
(*inverse f*) ⟨$⟩ *a* = (*f* ^ (*order f a* − *1*)) ⟨$⟩ *a*
⟨*proof*⟩

lemma *apply-inverse-self-in-orbit* [*simp*]:
(*inverse f*) ⟨$⟩ *a* ∈ *orbit f a*
⟨*proof*⟩

lemma *apply-inverse-power-eq*:
(*inverse* (*f* ^ *n*)) ⟨$⟩ *a* = (*f* ^ (*order f a* − *n mod order f a*)) ⟨$⟩ *a*
⟨*proof*⟩

lemma *apply-power-eq-self-iff*:
(*f* ^ *n*) ⟨$⟩ *a* = *a* ⟷ *order f a dvd n*
⟨*proof*⟩

lemma *orbit-equiv*:
assumes *b* ∈ *orbit f a*
shows *orbit f b* = *orbit f a* (**is** *?B* = *?A*)
⟨*proof*⟩

lemma *orbit-apply* [*simp*]:
orbit f (*f* ⟨$⟩ *a*) = *orbit f a*
⟨*proof*⟩

lemma *order-apply* [*simp*]:
order f (*f* ⟨$⟩ *a*) = *order f a*
⟨*proof*⟩

lemma *orbit-apply-inverse* [*simp*]:
orbit f (*inverse f* ⟨$⟩ *a*) = *orbit f a*
⟨*proof*⟩

lemma *order-apply-inverse* [*simp*]:
order f (*inverse f* ⟨$⟩ *a*) = *order f a*

⟨*proof*⟩

lemma *orbit-apply-power* [*simp*]:
orbit f ((*f* ^ *n*) ⟨$⟩ *a*) = *orbit f a*
⟨*proof*⟩

lemma *order-apply-power* [*simp*]:
order f ((*f* ^ *n*) ⟨$⟩ *a*) = *order f a*
⟨*proof*⟩

lemma *orbit-inverse* [*simp*]:
orbit (*inverse f*) = *orbit f*
⟨*proof*⟩

lemma *order-inverse* [*simp*]:
order (*inverse f*) = *order f*
⟨*proof*⟩

lemma *orbit-disjoint*:
assumes *orbit f a* ≠ *orbit f b*
shows *orbit f a* ∩ *orbit f b* = {}
⟨*proof*⟩

103.4 Swaps

lift-definition *swap* :: *'a* ⇒ *'a* ⇒ *'a perm* (⟨-↔-⟩)
is λ*a b*. *Fun.swap a b id*
⟨*proof*⟩

lemma *apply-swap-simp* [*simp*]:
⟨*a*↔*b*⟩ ⟨$⟩ *a* = *b*
⟨*a*↔*b*⟩ ⟨$⟩ *b* = *a*
⟨*proof*⟩

lemma *apply-swap-same* [*simp*]:
c ≠ *a* ⟹ *c* ≠ *b* ⟹ ⟨*a*↔*b*⟩ ⟨$⟩ *c* = *c*
⟨*proof*⟩

lemma *apply-swap-eq-iff* [*simp*]:
⟨*a*↔*b*⟩ ⟨$⟩ *c* = *a* ⟷ *c* = *b*
⟨*a*↔*b*⟩ ⟨$⟩ *c* = *b* ⟷ *c* = *a*
⟨*proof*⟩

lemma *swap-1* [*simp*]:
⟨*a*↔*a*⟩ = *1*
⟨*proof*⟩

lemma *swap-sym*:
⟨*b*↔*a*⟩ = ⟨*a*↔*b*⟩

⟨*proof*⟩

lemma *swap-self* [*simp*]:
⟨*a*↔*b*⟩ ∗ ⟨*a*↔*b*⟩ = *1*
⟨*proof*⟩

lemma *affected-swap*:
a ≠ *b* ⟹ *affected* ⟨*a*↔*b*⟩ = {*a*, *b*}
⟨*proof*⟩

lemma *inverse-swap* [*simp*]:
inverse ⟨*a*↔*b*⟩ = ⟨*a*↔*b*⟩
⟨*proof*⟩

103.5 Permutations specified by cycles

fun *cycle* :: *'a list* ⇒ *'a perm* (⟨-⟩)
where
⟨[]⟩ = *1*
| ⟨[*a*]⟩ = *1*
| ⟨*a* # *b* # *as*⟩ = ⟨*a* # *as*⟩ ∗ ⟨*a*↔*b*⟩

We do not continue and restrict ourselves to syntax from here. See also introductory note.

103.6 Syntax

bundle *no-permutation-syntax*
begin
no-notation *swap* (⟨-↔-⟩)
no-notation *cycle* (⟨-⟩)
no-notation *apply* (**infixl** ⟨$⟩ *999*)
end

bundle *permutation-syntax*
begin
notation *swap* (⟨-↔-⟩)
notation *cycle* (⟨-⟩)
notation *apply* (**infixl** ⟨$⟩ *999*)
no-notation *apply* (*op* ⟨$⟩)
end

unbundle *no-permutation-syntax*

end

104 Lists with elements distinct as canonical example for datatype invariants

theory *Dlist*
imports *Main*
begin

104.1 The type of distinct lists

typedef *'a dlist* = {*xs*::*'a list*. *distinct xs*}
  morphisms *list-of-dlist Abs-dlist*
⟨*proof*⟩

setup-lifting *type-definition-dlist*

lemma *dlist-eq-iff*:
  dxs = *dys* ⟷ *list-of-dlist dxs* = *list-of-dlist dys*
  ⟨*proof*⟩

lemma *dlist-eqI*:
  list-of-dlist dxs = *list-of-dlist dys* ⟹ *dxs* = *dys*
  ⟨*proof*⟩

Formal, totalized constructor for *'a dlist*:

definition *Dlist* :: *'a list* ⇒ *'a dlist* **where**
  Dlist xs = *Abs-dlist* (*remdups xs*)

lemma *distinct-list-of-dlist* [*simp*, *intro*]:
  distinct (*list-of-dlist dxs*)
  ⟨*proof*⟩

lemma *list-of-dlist-Dlist* [*simp*]:
  list-of-dlist (*Dlist xs*) = *remdups xs*
  ⟨*proof*⟩

lemma *remdups-list-of-dlist* [*simp*]:
  remdups (*list-of-dlist dxs*) = *list-of-dlist dxs*
  ⟨*proof*⟩

lemma *Dlist-list-of-dlist* [*simp*, *code abstype*]:
  Dlist (*list-of-dlist dxs*) = *dxs*
  ⟨*proof*⟩

Fundamental operations:

context
begin

qualified definition *empty* :: *'a dlist* **where**
  empty = *Dlist* []

qualified definition *insert* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
insert x dxs = *Dlist* (*List.insert x* (*list-of-dlist dxs*))

qualified definition *remove* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
remove x dxs = *Dlist* (*remove1 x* (*list-of-dlist dxs*))

qualified definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b dlist* **where**
map f dxs = *Dlist* (*remdups* (*List.map f* (*list-of-dlist dxs*)))

qualified definition *filter* :: (*'a* ⇒ *bool*) ⇒ *'a dlist* ⇒ *'a dlist* **where**
filter P dxs = *Dlist* (*List.filter P* (*list-of-dlist dxs*))

qualified definition *rotate* :: *nat* ⇒ *'a dlist* ⇒ *'a dlist* **where**
rotate n dxs = *Dlist* (*List.rotate n* (*list-of-dlist dxs*))

end

Derived operations:

context
begin

qualified definition *null* :: *'a dlist* ⇒ *bool* **where**
null dxs = *List.null* (*list-of-dlist dxs*)

qualified definition *member* :: *'a dlist* ⇒ *'a* ⇒ *bool* **where**
member dxs = *List.member* (*list-of-dlist dxs*)

qualified definition *length* :: *'a dlist* ⇒ *nat* **where**
length dxs = *List.length* (*list-of-dlist dxs*)

qualified definition *fold* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b* ⇒ *'b* **where**
fold f dxs = *List.fold f* (*list-of-dlist dxs*)

qualified definition *foldr* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b* ⇒ *'b* **where**
foldr f dxs = *List.foldr f* (*list-of-dlist dxs*)

end

104.2 Executable version obeying invariant

lemma *list-of-dlist-empty* [*simp*, *code abstract*]:
list-of-dlist Dlist.empty = []
⟨*proof*⟩

lemma *list-of-dlist-insert* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.insert x dxs*) = *List.insert x* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-remove* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.remove x dxs*) = *remove1 x* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-map* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.map f dxs*) = *remdups* (*List.map f* (*list-of-dlist dxs*))
⟨*proof*⟩

lemma *list-of-dlist-filter* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.filter P dxs*) = *List.filter P* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-rotate* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.rotate n dxs*) = *List.rotate n* (*list-of-dlist dxs*)
⟨*proof*⟩

Explicit executable conversion

definition *dlist-of-list* [*simp*]:
dlist-of-list = *Dlist*

lemma [*code abstract*]:
list-of-dlist (*dlist-of-list xs*) = *remdups xs*
⟨*proof*⟩

Equality

instantiation *dlist* :: (*equal*) *equal*
begin

definition *HOL.equal dxs dys* ⟷ *HOL.equal* (*list-of-dlist dxs*) (*list-of-dlist dys*)

instance
⟨*proof*⟩

end

declare *equal-dlist-def* [*code*]

lemma [*code nbe*]: *HOL.equal* (*dxs* :: 'a::*equal dlist*) *dxs* ⟷ *True*
⟨*proof*⟩

104.3 Induction principle and case distinction

lemma *dlist-induct* [*case-names empty insert*, *induct type*: *dlist*]:
assumes *empty*: *P Dlist.empty*
assumes *insrt*: ⋀*x dxs*. ¬ *Dlist.member dxs x* ⟹ *P dxs* ⟹ *P* (*Dlist.insert x dxs*)
shows *P dxs*
⟨*proof*⟩

lemma *dlist-case* [*cases type*: *dlist*]:
  obtains (*empty*) *dxs* = *Dlist.empty*
    | (*insert*) *x dys* **where** ¬ *Dlist.member dys x* **and** *dxs* = *Dlist.insert x dys*
⟨*proof*⟩

104.4 Functorial structure

functor *map*: *map*
  ⟨*proof*⟩

104.5 Quickcheck generators

quickcheck-generator *dlist predicate*: *distinct constructors*: *Dlist.empty*, *Dlist.insert*

104.6 BNF instance

context begin

qualified fun *wpull* :: ('*a* × '*b*) *list* ⇒ ('*b* × '*c*) *list* ⇒ ('*a* × '*c*) *list*
where
  wpull [] *ys* = []
| *wpull xs* [] = []
| *wpull* ((*a*, *b*) # *xs*) ((*b*′, *c*) # *ys*) =
  (*if b* ∈ *snd* ‘ *set xs then*
     (*a*, *the* (*map-of* (*rev* ((*b*′, *c*) # *ys*)) *b*)) # *wpull xs* ((*b*′, *c*) # *ys*)
   else if b′ ∈ *fst* ‘ *set ys then*
     (*the* (*map-of* (*map prod.swap* (*rev* ((*a*, *b*) # *xs*))) *b*′), *c*) # *wpull* ((*a*, *b*) # *xs*) *ys*
   else (*a*, *c*) # *wpull xs ys*)

qualified lemma *wpull-eq-Nil-iff* [*simp*]: *wpull xs ys* = [] ⟷ *xs* = [] ∨ *ys* = []
⟨*proof*⟩ **lemma** *wpull-induct*
  [*consumes 1*,
   case-names Nil left[*xs eq in-set IH*] *right*[*xs ys eq in-set IH*] *step*[*xs ys eq IH*]]:
  assumes *eq*: *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
  and *Nil*: *P* [] []
  and *left*: ⋀*a b xs b*′ *c ys*.
    ⟦ *b* ∈ *snd* ‘ *set xs*; *remdups* (*map snd xs*) = *remdups* (*map fst* ((*b*′, *c*) # *ys*));
      (*b*, *the* (*map-of* (*rev* ((*b*′, *c*) # *ys*)) *b*)) ∈ *set* ((*b*′, *c*) # *ys*); *P xs* ((*b*′, *c*) # *ys*) ⟧
    ⟹ *P* ((*a*, *b*) # *xs*) ((*b*′, *c*) # *ys*)
  and *right*: ⋀*a b xs b*′ *c ys*.
    ⟦ *b* ∉ *snd* ‘ *set xs*; *b*′ ∈ *fst* ‘ *set ys*;
      remdups (*map snd* ((*a*, *b*) # *xs*)) = *remdups* (*map fst ys*);
      (*the* (*map-of* (*map prod.swap* (*rev* ((*a*, *b*) #*xs*))) *b*′), *b*′) ∈ *set* ((*a*, *b*) # *xs*);
      P ((*a*, *b*) # *xs*) *ys* ⟧
    ⟹ *P* ((*a*, *b*) # *xs*) ((*b*′, *c*) # *ys*)
  and *step*: ⋀*a b xs c ys*.
    ⟦ *b* ∉ *snd* ‘ *set xs*; *b* ∉ *fst* ‘ *set ys*; *remdups* (*map snd xs*) = *remdups* (*map fst ys*);

P xs ys ⟧
⟹ *P* ((*a*, *b*) # *xs*) ((*b*, *c*) # *ys*)
shows *P xs ys*
⟨*proof*⟩ **lemma** *set-wpull-subset*:
assumes *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
shows *set* (*wpull xs ys*) ⊆ *set xs O set ys*
⟨*proof*⟩ **lemma** *set-fst-wpull*:
assumes *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
shows *fst* ‘ *set* (*wpull xs ys*) = *fst* ‘ *set xs*
⟨*proof*⟩ **lemma** *set-snd-wpull*:
assumes *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
shows *snd* ‘ *set* (*wpull xs ys*) = *snd* ‘ *set ys*
⟨*proof*⟩ **lemma** *wpull*:
assumes *distinct xs*
and *distinct ys*
and *set xs* ⊆ {(*x*, *y*). *R x y*}
and *set ys* ⊆ {(*x*, *y*). *S x y*}
and *eq*: *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
shows ∃ *zs*. *distinct zs* ∧ *set zs* ⊆ {(*x*, *y*). (*R OO S*) *x y*} ∧
remdups (*map fst zs*) = *remdups* (*map fst xs*) ∧ *remdups* (*map snd zs*) = *remdups* (*map snd ys*)
⟨*proof*⟩ **lift-definition** *set* :: *'a dlist* ⇒ *'a set* **is** *List.set* ⟨*proof*⟩ **lemma** *map-transfer* [*transfer-rule*]:
(*rel-fun op* = (*rel-fun* (*pcr-dlist op* =) (*pcr-dlist op* =))) (λ*f x*. *remdups* (*List.map f x*)) *Dlist.map*
⟨*proof*⟩

bnf *'a dlist*
map: *Dlist.map*
sets: *set*
bd: *natLeq*
wits: *Dlist.empty*
⟨*proof*⟩

lifting-update *dlist.lifting*
lifting-forget *dlist.lifting*

end

end

105 Fragments on permuations

theory *Perm-Fragments*
imports ~~/*src*/*HOL*/*Library*/*Perm* ~~/*src*/*HOL*/*Library*/*Dlist*
begin

unbundle *permutation-syntax*

On cycles

lemma *cycle-prod-list*:
  ⟨*a* # *as*⟩ = *prod-list* (*map* (λ*b*. ⟨*a*↔*b*⟩) (*rev as*))
  ⟨*proof*⟩

lemma *cycle-append* [*simp*]:
  ⟨*a* # *as* @ *bs*⟩ = ⟨*a* # *bs*⟩ ∗ ⟨*a* # *as*⟩
⟨*proof*⟩

lemma *affected-cycle*:
  affected ⟨*as*⟩ ⊆ *set as*
⟨*proof*⟩

lemma *orbit-cycle-non-elem*:
  assumes *a* ∉ *set as*
  shows *orbit* ⟨*as*⟩ *a* = {*a*}
  ⟨*proof*⟩

lemma *inverse-cycle*:
  assumes *distinct as*
  shows *inverse* ⟨*as*⟩ = ⟨*rev as*⟩
⟨*proof*⟩

lemma *order-cycle-non-elem*:
  assumes *a* ∉ *set as*
  shows *order* ⟨*as*⟩ *a* = *1*
⟨*proof*⟩

lemma *orbit-cycle-elem*:
  assumes *distinct as* **and** *a* ∈ *set as*
  shows *orbit* ⟨*as*⟩ *a* = *set as*
  ⟨*proof*⟩

lemma *order-cycle-elem*:
  assumes *distinct as* **and** *a* ∈ *set as*
  shows *order* ⟨*as*⟩ *a* = *length as*
  ⟨*proof*⟩

Adding fixpoints

definition *fixate* :: *'a* ⇒ *'a perm* ⇒ *'a perm*
where
  fixate a f = (*if a* ∈ *affected f then f* ∗ ⟨*apply* (*inverse f*) *a*↔*a*⟩ *else f*)

lemma *affected-fixate-trivial*:
  assumes *a* ∉ *affected f*
  shows *affected* (*fixate a f*) = *affected f*
  ⟨*proof*⟩

lemma *affected-fixate-binary-circle*:

assumes *order f a* = *2*
shows *affected* (*fixate a f*) = *affected f* − {*a*, *apply f a*} (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *affected-fixate-no-binary-circle*:
assumes *order f a* > *2*
shows *affected* (*fixate a f*) = *affected f* − {*a*} (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *affected-fixate*:
affected (*fixate a f*) ⊆ *affected f* − {*a*}
⟨*proof*⟩

lemma *orbit-fixate-self* [*simp*]:
orbit (*fixate a f*) *a* = {*a*}
⟨*proof*⟩

lemma *order-fixate-self* [*simp*]:
order (*fixate a f*) *a* = *1*
⟨*proof*⟩

lemma
assumes *b* ∉ *orbit f a*
shows *orbit* (*fixate b f*) *a* = *orbit f a*
⟨*proof*⟩

lemma
assumes *b* ∈ *orbit f a* **and** *b* ≠ *a*
shows *orbit* (*fixate b f*) *a* = *orbit f a* − {*b*}
⟨*proof*⟩

Distilling cycles from permutations

inductive-set *orbits* :: *'a perm* ⇒ *'a set set* **for** *f*
where
in-orbitsI: *a* ∈ *affected f* ⟹ *orbit f a* ∈ *orbits f*

lemma *orbits-unfold*:
orbits f = *orbit f* ‘ *affected f*
⟨*proof*⟩

lemma *in-orbit-affected*:
assumes *b* ∈ *orbit f a*
assumes *a* ∈ *affected f*
shows *b* ∈ *affected f*
⟨*proof*⟩

lemma *Union-orbits* [*simp*]:
⋃ *orbits f* = *affected f*
⟨*proof*⟩

lemma *finite-orbits* [*simp*]:
  finite (*orbits f*)
  ⟨*proof*⟩

lemma *card-in-orbits*:
  assumes $A \in$ *orbits f*
  shows *card A* $\geq$ *2*
  ⟨*proof*⟩

lemma *disjoint-orbits*:
  assumes $A \in$ *orbits f* **and** $B \in$ *orbits f* **and** $A \neq B$
  shows $A \cap B = \{\}$
  ⟨*proof*⟩

definition *trace* :: *'a* ⇒ *'a perm* ⇒ *'a list*
where
  trace a f = *map* (λ*n*. *apply* (*f* ^ *n*) *a*) [*0*..<*order f a*]

lemma *set-trace-eq* [*simp*]:
  set (*trace a f*) = *orbit f a*
  ⟨*proof*⟩

definition *seeds* :: *'a perm* ⇒ *'a*::*linorder list*
where
  seeds f = *sorted-list-of-set* (*Min* ‘ *orbits f*)

definition *cycles* :: *'a perm* ⇒ *'a*::*linorder list list*
where
  cycles f = *map* (λ*a*. *trace a f*) (*seeds f*)

lemma (**in** *comm-monoid-list-set*) *sorted-list-of-set*:
  assumes *finite A*
  shows *list.F* (*map h* (*sorted-list-of-set A*)) = *set.F h A*
⟨*proof*⟩

Misc

primrec *subtract* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
  subtract [] *ys* = *ys*
| *subtract* (*x* # *xs*) *ys* = *subtract xs* (*removeAll x ys*)

lemma *length-subtract-less-eq* [*simp*]:
  length (*subtract xs ys*) ≤ *length ys*
⟨*proof*⟩

end

106 Argo

theory *Argo-Examples*
imports *Complex-Main*
begin

This theory is intended to showcase and test different features of the *argo* proof method.

The *argo* proof method can be applied to propositional problems, problems involving equality reasoning and problems of linear real arithmetic.

The *argo* proof method provides two options. To specify an upper limit of the proof methods run time in seconds, use the option *argo-timeout*. To specify the amount of output, use the option *argo-trace* with value *none* for no tracing output, value *basic* for viewing the underlying propositions and some timings, and value *full* for additionally inspecting the proof replay steps.

declare[[*argo-trace* = *full*]]

106.1 Propositional logic

notepad
begin
⟨*proof*⟩
end

106.2 Equality, congruence and predicates

notepad
begin
⟨*proof*⟩
end

106.3 Linear real arithmetic

106.3.1 Negation and subtraction

notepad
begin
⟨*proof*⟩
end

106.3.2 Multiplication

notepad
begin
⟨*proof*⟩
end

106.3.3 Division

notepad
begin
⟨*proof*⟩
end

106.3.4 Addition

notepad
begin
⟨*proof*⟩
end

106.3.5 Minimum and maximum

notepad
begin
⟨*proof*⟩
end

106.3.6 Absolute value

notepad
begin
⟨*proof*⟩
end

106.3.7 Equality

notepad
begin
⟨*proof*⟩
end

106.3.8 Less-equal

notepad
begin
⟨*proof*⟩
end

106.3.9 Less

notepad
begin
⟨*proof*⟩
end

106.3.10 Other examples

notepad
begin
⟨*proof*⟩
end

106.4 Larger examples

declare[[*argo-trace* = *basic*, *argo-timeout* = *60*]]

Translated from TPTP problem library: PUZ015-2.006.dimacs

lemma assumes *1*: ~*x0*
and *2*: ~*x30*
and *3*: ~*x29*
and *4*: ~*x59*
and *5*: *x1* | *x31* | *x0*
and *6*: *x2* | *x32* | *x1*
and *7*: *x3* | *x33* | *x2*
and *8*: *x4* | *x34* | *x3*
and *9*: *x35* | *x4*
and *10*: *x5* | *x36* | *x30*
and *11*: *x6* | *x37* | *x5* | *x31*
and *12*: *x7* | *x38* | *x6* | *x32*
and *13*: *x8* | *x39* | *x7* | *x33*
and *14*: *x9* | *x40* | *x8* | *x34*
and *15*: *x41* | *x9* | *x35*
and *16*: *x10* | *x42* | *x36*
and *17*: *x11* | *x43* | *x10* | *x37*
and *18*: *x12* | *x44* | *x11* | *x38*
and *19*: *x13* | *x45* | *x12* | *x39*
and *20*: *x14* | *x46* | *x13* | *x40*
and *21*: *x47* | *x14* | *x41*
and *22*: *x15* | *x48* | *x42*
and *23*: *x16* | *x49* | *x15* | *x43*
and *24*: *x17* | *x50* | *x16* | *x44*
and *25*: *x18* | *x51* | *x17* | *x45*
and *26*: *x19* | *x52* | *x18* | *x46*
and *27*: *x53* | *x19* | *x47*
and *28*: *x20* | *x54* | *x48*
and *29*: *x21* | *x55* | *x20* | *x49*
and *30*: *x22* | *x56* | *x21* | *x50*
and *31*: *x23* | *x57* | *x22* | *x51*
and *32*: *x24* | *x58* | *x23* | *x52*
and *33*: *x59* | *x24* | *x53*
and *34*: *x25* | *x54*
and *35*: *x26* | *x25* | *x55*
and *36*: *x27* | *x26* | *x56*
and *37*: *x28* | *x27* | *x57*
and *38*: *x29* | *x28* | *x58*

and *39*: ~*x1* | ~*x31*
and *40*: ~*x1* | ~*x0*
and *41*: ~*x31* | ~*x0*
and *42*: ~*x2* | ~*x32*
and *43*: ~*x2* | ~*x1*
and *44*: ~*x32* | ~*x1*
and *45*: ~*x3* | ~*x33*
and *46*: ~*x3* | ~*x2*
and *47*: ~*x33* | ~*x2*
and *48*: ~*x4* | ~*x34*
and *49*: ~*x4* | ~*x3*
and *50*: ~*x34* | ~*x3*
and *51*: ~*x35* | ~*x4*
and *52*: ~*x5* | ~*x36*
and *53*: ~*x5* | ~*x30*
and *54*: ~*x36* | ~*x30*
and *55*: ~*x6* | ~*x37*
and *56*: ~*x6* | ~*x5*
and *57*: ~*x6* | ~*x31*
and *58*: ~*x37* | ~*x5*
and *59*: ~*x37* | ~*x31*
and *60*: ~*x5* | ~*x31*
and *61*: ~*x7* | ~*x38*
and *62*: ~*x7* | ~*x6*
and *63*: ~*x7* | ~*x32*
and *64*: ~*x38* | ~*x6*
and *65*: ~*x38* | ~*x32*
and *66*: ~*x6* | ~*x32*
and *67*: ~*x8* | ~*x39*
and *68*: ~*x8* | ~*x7*
and *69*: ~*x8* | ~*x33*
and *70*: ~*x39* | ~*x7*
and *71*: ~*x39* | ~*x33*
and *72*: ~*x7* | ~*x33*
and *73*: ~*x9* | ~*x40*
and *74*: ~*x9* | ~*x8*
and *75*: ~*x9* | ~*x34*
and *76*: ~*x40* | ~*x8*
and *77*: ~*x40* | ~*x34*
and *78*: ~*x8* | ~*x34*
and *79*: ~*x41* | ~*x9*
and *80*: ~*x41* | ~*x35*
and *81*: ~*x9* | ~*x35*
and *82*: ~*x10* | ~*x42*
and *83*: ~*x10* | ~*x36*
and *84*: ~*x42* | ~*x36*
and *85*: ~*x11* | ~*x43*
and *86*: ~*x11* | ~*x10*
and *87*: ~*x11* | ~*x37*

and *88*: ~*x43* | ~*x10*
and *89*: ~*x43* | ~*x37*
and *90*: ~*x10* | ~*x37*
and *91*: ~*x12* | ~*x44*
and *92*: ~*x12* | ~*x11*
and *93*: ~*x12* | ~*x38*
and *94*: ~*x44* | ~*x11*
and *95*: ~*x44* | ~*x38*
and *96*: ~*x11* | ~*x38*
and *97*: ~*x13* | ~*x45*
and *98*: ~*x13* | ~*x12*
and *99*: ~*x13* | ~*x39*
and *100*: ~*x45* | ~*x12*
and *101*: ~*x45* | ~*x39*
and *102*: ~*x12* | ~*x39*
and *103*: ~*x14* | ~*x46*
and *104*: ~*x14* | ~*x13*
and *105*: ~*x14* | ~*x40*
and *106*: ~*x46* | ~*x13*
and *107*: ~*x46* | ~*x40*
and *108*: ~*x13* | ~*x40*
and *109*: ~*x47* | ~*x14*
and *110*: ~*x47* | ~*x41*
and *111*: ~*x14* | ~*x41*
and *112*: ~*x15* | ~*x48*
and *113*: ~*x15* | ~*x42*
and *114*: ~*x48* | ~*x42*
and *115*: ~*x16* | ~*x49*
and *116*: ~*x16* | ~*x15*
and *117*: ~*x16* | ~*x43*
and *118*: ~*x49* | ~*x15*
and *119*: ~*x49* | ~*x43*
and *120*: ~*x15* | ~*x43*
and *121*: ~*x17* | ~*x50*
and *122*: ~*x17* | ~*x16*
and *123*: ~*x17* | ~*x44*
and *124*: ~*x50* | ~*x16*
and *125*: ~*x50* | ~*x44*
and *126*: ~*x16* | ~*x44*
and *127*: ~*x18* | ~*x51*
and *128*: ~*x18* | ~*x17*
and *129*: ~*x18* | ~*x45*
and *130*: ~*x51* | ~*x17*
and *131*: ~*x51* | ~*x45*
and *132*: ~*x17* | ~*x45*
and *133*: ~*x19* | ~*x52*
and *134*: ~*x19* | ~*x18*
and *135*: ~*x19* | ~*x46*
and *136*: ~*x52* | ~*x18*

and *137*: ~*x52* | ~*x46*
and *138*: ~*x18* | ~*x46*
and *139*: ~*x53* | ~*x19*
and *140*: ~*x53* | ~*x47*
and *141*: ~*x19* | ~*x47*
and *142*: ~*x20* | ~*x54*
and *143*: ~*x20* | ~*x48*
and *144*: ~*x54* | ~*x48*
and *145*: ~*x21* | ~*x55*
and *146*: ~*x21* | ~*x20*
and *147*: ~*x21* | ~*x49*
and *148*: ~*x55* | ~*x20*
and *149*: ~*x55* | ~*x49*
and *150*: ~*x20* | ~*x49*
and *151*: ~*x22* | ~*x56*
and *152*: ~*x22* | ~*x21*
and *153*: ~*x22* | ~*x50*
and *154*: ~*x56* | ~*x21*
and *155*: ~*x56* | ~*x50*
and *156*: ~*x21* | ~*x50*
and *157*: ~*x23* | ~*x57*
and *158*: ~*x23* | ~*x22*
and *159*: ~*x23* | ~*x51*
and *160*: ~*x57* | ~*x22*
and *161*: ~*x57* | ~*x51*
and *162*: ~*x22* | ~*x51*
and *163*: ~*x24* | ~*x58*
and *164*: ~*x24* | ~*x23*
and *165*: ~*x24* | ~*x52*
and *166*: ~*x58* | ~*x23*
and *167*: ~*x58* | ~*x52*
and *168*: ~*x23* | ~*x52*
and *169*: ~*x59* | ~*x24*
and *170*: ~*x59* | ~*x53*
and *171*: ~*x24* | ~*x53*
and *172*: ~*x25* | ~*x54*
and *173*: ~*x26* | ~*x25*
and *174*: ~*x26* | ~*x55*
and *175*: ~*x25* | ~*x55*
and *176*: ~*x27* | ~*x26*
and *177*: ~*x27* | ~*x56*
and *178*: ~*x26* | ~*x56*
and *179*: ~*x28* | ~*x27*
and *180*: ~*x28* | ~*x57*
and *181*: ~*x27* | ~*x57*
and *182*: ~*x29* | ~*x28*
and *183*: ~*x29* | ~*x58*
and *184*: ~*x28* | ~*x58*
shows *False*

⟨*proof*⟩

Translated from TPTP problem library: MSC007-1.008.dimacs

lemma assumes *1*: *x0* | *x1* | *x2* | *x3* | *x4* | *x5* | *x6*
and *2*: *x7* | *x8* | *x9* | *x10* | *x11* | *x12* | *x13*
and *3*: *x14* | *x15* | *x16* | *x17* | *x18* | *x19* | *x20*
and *4*: *x21* | *x22* | *x23* | *x24* | *x25* | *x26* | *x27*
and *5*: *x28* | *x29* | *x30* | *x31* | *x32* | *x33* | *x34*
and *6*: *x35* | *x36* | *x37* | *x38* | *x39* | *x40* | *x41*
and *7*: *x42* | *x43* | *x44* | *x45* | *x46* | *x47* | *x48*
and *8*: *x49* | *x50* | *x51* | *x52* | *x53* | *x54* | *x55*
and *9*: ~*x0* | ~*x7*
and *10*: ~*x0* | ~*x14*
and *11*: ~*x0* | ~*x21*
and *12*: ~*x0* | ~*x28*
and *13*: ~*x0* | ~*x35*
and *14*: ~*x0* | ~*x42*
and *15*: ~*x0* | ~*x49*
and *16*: ~*x7* | ~*x14*
and *17*: ~*x7* | ~*x21*
and *18*: ~*x7* | ~*x28*
and *19*: ~*x7* | ~*x35*
and *20*: ~*x7* | ~*x42*
and *21*: ~*x7* | ~*x49*
and *22*: ~*x14* | ~*x21*
and *23*: ~*x14* | ~*x28*
and *24*: ~*x14* | ~*x35*
and *25*: ~*x14* | ~*x42*
and *26*: ~*x14* | ~*x49*
and *27*: ~*x21* | ~*x28*
and *28*: ~*x21* | ~*x35*
and *29*: ~*x21* | ~*x42*
and *30*: ~*x21* | ~*x49*
and *31*: ~*x28* | ~*x35*
and *32*: ~*x28* | ~*x42*
and *33*: ~*x28* | ~*x49*
and *34*: ~*x35* | ~*x42*
and *35*: ~*x35* | ~*x49*
and *36*: ~*x42* | ~*x49*
and *37*: ~*x1* | ~*x8*
and *38*: ~*x1* | ~*x15*
and *39*: ~*x1* | ~*x22*
and *40*: ~*x1* | ~*x29*
and *41*: ~*x1* | ~*x36*
and *42*: ~*x1* | ~*x43*
and *43*: ~*x1* | ~*x50*
and *44*: ~*x8* | ~*x15*
and *45*: ~*x8* | ~*x22*
and *46*: ~*x8* | ~*x29*

and *47*: ~*x8* | ~*x36*
and *48*: ~*x8* | ~*x43*
and *49*: ~*x8* | ~*x50*
and *50*: ~*x15* | ~*x22*
and *51*: ~*x15* | ~*x29*
and *52*: ~*x15* | ~*x36*
and *53*: ~*x15* | ~*x43*
and *54*: ~*x15* | ~*x50*
and *55*: ~*x22* | ~*x29*
and *56*: ~*x22* | ~*x36*
and *57*: ~*x22* | ~*x43*
and *58*: ~*x22* | ~*x50*
and *59*: ~*x29* | ~*x36*
and *60*: ~*x29* | ~*x43*
and *61*: ~*x29* | ~*x50*
and *62*: ~*x36* | ~*x43*
and *63*: ~*x36* | ~*x50*
and *64*: ~*x43* | ~*x50*
and *65*: ~*x2* | ~*x9*
and *66*: ~*x2* | ~*x16*
and *67*: ~*x2* | ~*x23*
and *68*: ~*x2* | ~*x30*
and *69*: ~*x2* | ~*x37*
and *70*: ~*x2* | ~*x44*
and *71*: ~*x2* | ~*x51*
and *72*: ~*x9* | ~*x16*
and *73*: ~*x9* | ~*x23*
and *74*: ~*x9* | ~*x30*
and *75*: ~*x9* | ~*x37*
and *76*: ~*x9* | ~*x44*
and *77*: ~*x9* | ~*x51*
and *78*: ~*x16* | ~*x23*
and *79*: ~*x16* | ~*x30*
and *80*: ~*x16* | ~*x37*
and *81*: ~*x16* | ~*x44*
and *82*: ~*x16* | ~*x51*
and *83*: ~*x23* | ~*x30*
and *84*: ~*x23* | ~*x37*
and *85*: ~*x23* | ~*x44*
and *86*: ~*x23* | ~*x51*
and *87*: ~*x30* | ~*x37*
and *88*: ~*x30* | ~*x44*
and *89*: ~*x30* | ~*x51*
and *90*: ~*x37* | ~*x44*
and *91*: ~*x37* | ~*x51*
and *92*: ~*x44* | ~*x51*
and *93*: ~*x3* | ~*x10*
and *94*: ~*x3* | ~*x17*
and *95*: ~*x3* | ~*x24*

and *96*: ~*x3* | ~*x31*
and *97*: ~*x3* | ~*x38*
and *98*: ~*x3* | ~*x45*
and *99*: ~*x3* | ~*x52*
and *100*: ~*x10* | ~*x17*
and *101*: ~*x10* | ~*x24*
and *102*: ~*x10* | ~*x31*
and *103*: ~*x10* | ~*x38*
and *104*: ~*x10* | ~*x45*
and *105*: ~*x10* | ~*x52*
and *106*: ~*x17* | ~*x24*
and *107*: ~*x17* | ~*x31*
and *108*: ~*x17* | ~*x38*
and *109*: ~*x17* | ~*x45*
and *110*: ~*x17* | ~*x52*
and *111*: ~*x24* | ~*x31*
and *112*: ~*x24* | ~*x38*
and *113*: ~*x24* | ~*x45*
and *114*: ~*x24* | ~*x52*
and *115*: ~*x31* | ~*x38*
and *116*: ~*x31* | ~*x45*
and *117*: ~*x31* | ~*x52*
and *118*: ~*x38* | ~*x45*
and *119*: ~*x38* | ~*x52*
and *120*: ~*x45* | ~*x52*
and *121*: ~*x4* | ~*x11*
and *122*: ~*x4* | ~*x18*
and *123*: ~*x4* | ~*x25*
and *124*: ~*x4* | ~*x32*
and *125*: ~*x4* | ~*x39*
and *126*: ~*x4* | ~*x46*
and *127*: ~*x4* | ~*x53*
and *128*: ~*x11* | ~*x18*
and *129*: ~*x11* | ~*x25*
and *130*: ~*x11* | ~*x32*
and *131*: ~*x11* | ~*x39*
and *132*: ~*x11* | ~*x46*
and *133*: ~*x11* | ~*x53*
and *134*: ~*x18* | ~*x25*
and *135*: ~*x18* | ~*x32*
and *136*: ~*x18* | ~*x39*
and *137*: ~*x18* | ~*x46*
and *138*: ~*x18* | ~*x53*
and *139*: ~*x25* | ~*x32*
and *140*: ~*x25* | ~*x39*
and *141*: ~*x25* | ~*x46*
and *142*: ~*x25* | ~*x53*
and *143*: ~*x32* | ~*x39*
and *144*: ~*x32* | ~*x46*

and *145*: ~*x32* | ~*x53*
and *146*: ~*x39* | ~*x46*
and *147*: ~*x39* | ~*x53*
and *148*: ~*x46* | ~*x53*
and *149*: ~*x5* | ~*x12*
and *150*: ~*x5* | ~*x19*
and *151*: ~*x5* | ~*x26*
and *152*: ~*x5* | ~*x33*
and *153*: ~*x5* | ~*x40*
and *154*: ~*x5* | ~*x47*
and *155*: ~*x5* | ~*x54*
and *156*: ~*x12* | ~*x19*
and *157*: ~*x12* | ~*x26*
and *158*: ~*x12* | ~*x33*
and *159*: ~*x12* | ~*x40*
and *160*: ~*x12* | ~*x47*
and *161*: ~*x12* | ~*x54*
and *162*: ~*x19* | ~*x26*
and *163*: ~*x19* | ~*x33*
and *164*: ~*x19* | ~*x40*
and *165*: ~*x19* | ~*x47*
and *166*: ~*x19* | ~*x54*
and *167*: ~*x26* | ~*x33*
and *168*: ~*x26* | ~*x40*
and *169*: ~*x26* | ~*x47*
and *170*: ~*x26* | ~*x54*
and *171*: ~*x33* | ~*x40*
and *172*: ~*x33* | ~*x47*
and *173*: ~*x33* | ~*x54*
and *174*: ~*x40* | ~*x47*
and *175*: ~*x40* | ~*x54*
and *176*: ~*x47* | ~*x54*
and *177*: ~*x6* | ~*x13*
and *178*: ~*x6* | ~*x20*
and *179*: ~*x6* | ~*x27*
and *180*: ~*x6* | ~*x34*
and *181*: ~*x6* | ~*x41*
and *182*: ~*x6* | ~*x48*
and *183*: ~*x6* | ~*x55*
and *184*: ~*x13* | ~*x20*
and *185*: ~*x13* | ~*x27*
and *186*: ~*x13* | ~*x34*
and *187*: ~*x13* | ~*x41*
and *188*: ~*x13* | ~*x48*
and *189*: ~*x13* | ~*x55*
and *190*: ~*x20* | ~*x27*
and *191*: ~*x20* | ~*x34*
and *192*: ~*x20* | ~*x41*
and *193*: ~*x20* | ~*x48*

and *194*: ~*x20* | ~*x55*
and *195*: ~*x27* | ~*x34*
and *196*: ~*x27* | ~*x41*
and *197*: ~*x27* | ~*x48*
and *198*: ~*x27* | ~*x55*
and *199*: ~*x34* | ~*x41*
and *200*: ~*x34* | ~*x48*
and *201*: ~*x34* | ~*x55*
and *202*: ~*x41* | ~*x48*
and *203*: ~*x41* | ~*x55*
and *204*: ~*x48* | ~*x55*
shows *False*
⟨*proof*⟩

lemma $0 \leq (yc::real) \wedge$
$0 \leq yd \wedge 0 \leq yb \wedge 0 \leq ya \Longrightarrow$
$0 \leq yf \wedge$
$0 \leq xh \wedge 0 \leq ye \wedge 0 \leq yg \Longrightarrow$
$0 \leq yw \wedge 0 \leq xs \wedge 0 \leq yu \Longrightarrow$
$0 \leq aea \wedge 0 \leq aee \wedge 0 \leq aed \Longrightarrow$
$0 \leq zy \wedge 0 \leq xz \wedge 0 \leq zw \Longrightarrow$
$0 \leq zb \wedge$
$0 \leq za \wedge 0 \leq yy \wedge 0 \leq yz \Longrightarrow$
$0 \leq zp \wedge 0 \leq zo \wedge 0 \leq yq \Longrightarrow$
$0 \leq adp \wedge 0 \leq aeb \wedge 0 \leq aec \Longrightarrow$
$0 \leq acm \wedge 0 \leq aco \wedge 0 \leq acn \Longrightarrow$
$0 \leq abl \Longrightarrow$
$0 \leq zr \wedge 0 \leq zq \wedge 0 \leq abh \Longrightarrow$
$0 \leq abq \wedge 0 \leq zd \wedge 0 \leq abo \Longrightarrow$
$0 \leq acd \wedge$
$0 \leq acc \wedge 0 \leq xi \wedge 0 \leq acb \Longrightarrow$
$0 \leq acp \wedge 0 \leq acr \wedge 0 \leq acq \Longrightarrow$
$0 \leq xw \wedge$
$0 \leq xr \wedge 0 \leq xv \wedge 0 \leq xu \Longrightarrow$
$0 \leq zc \wedge 0 \leq acg \wedge 0 \leq ach \Longrightarrow$
$0 \leq zt \wedge 0 \leq zs \wedge 0 \leq xy \Longrightarrow$
$0 \leq ady \wedge 0 \leq adw \wedge 0 \leq zg \Longrightarrow$
$0 \leq abd \wedge$
$0 \leq abc \wedge 0 \leq yr \wedge 0 \leq abb \Longrightarrow$
$0 \leq adi \wedge$
$0 \leq x \wedge 0 \leq adh \wedge 0 \leq xa \Longrightarrow$
$0 \leq aak \wedge 0 \leq aai \wedge 0 \leq aad \Longrightarrow$
$0 \leq aba \wedge 0 \leq zh \wedge 0 \leq aay \Longrightarrow$
$0 \leq abg \wedge 0 \leq ys \wedge 0 \leq abe \Longrightarrow$
$0 \leq abs1 \wedge$
$0 \leq yt \wedge 0 \leq abr \wedge 0 \leq zu \Longrightarrow$
$0 \leq abv \wedge$
$0 \leq zn \wedge 0 \leq abw \wedge 0 \leq zm \Longrightarrow$

$0 \leq adl \wedge 0 \leq adn \implies$
$0 \leq acf \wedge 0 \leq aca \implies$
$0 \leq ads \wedge 0 \leq aaq \implies$
$0 \leq ada \implies$
$0 \leq aaf \wedge 0 \leq aac \wedge 0 \leq aag \implies$
$0 \leq aal \wedge$
$0 \leq acu \wedge 0 \leq acs \wedge 0 \leq act \implies$
$0 \leq aas \wedge 0 \leq xb \wedge 0 \leq aat \implies$
$0 \leq zk \wedge 0 \leq zj \wedge 0 \leq zi \implies$
$0 \leq ack \wedge$
$0 \leq acj \wedge 0 \leq xc \wedge 0 \leq aci \implies$
$0 \leq aav \wedge 0 \leq aah \wedge 0 \leq xd \implies$
$0 \leq abt \wedge$
$0 \leq xo \wedge 0 \leq abu \wedge 0 \leq xn \implies$
$0 \leq adc \wedge$
$0 \leq abz \wedge 0 \leq adc \wedge 0 \leq abz \implies$
$0 \leq xt \wedge$
$0 \leq zz \wedge 0 \leq aab \wedge 0 \leq aaa \implies$
$0 \leq adq \wedge$
$0 \leq xl \wedge 0 \leq adr \wedge 0 \leq adb \implies$
$0 \leq zf \wedge 0 \leq yh \wedge 0 \leq yi \implies$
$0 \leq aao \wedge 0 \leq aam \wedge 0 \leq xe \implies$
$0 \leq abk \wedge$
$0 \leq aby \wedge 0 \leq abj \wedge 0 \leq abx \implies$
$0 \leq yp \implies$
$0 \leq yl \wedge 0 \leq yj \wedge 0 \leq ym \implies$
$0 \leq acw \implies$
$0 \leq adk \wedge$
$0 \leq adg \wedge 0 \leq adj \wedge 0 \leq adf \implies$
$0 \leq adv \wedge 0 \leq xf \wedge 0 \leq adu \implies$
$yc + yd + yb + ya = 1 \implies$
$yf + xh + ye + yg = 1 \implies$
$yw + xs + yu = 1 \implies$
$aea + aee + aed = 1 \implies$
$zy + xz + zw = 1 \implies$
$zb + za + yy + yz = 1 \implies$
$zp + zo + yq = 1 \implies$
$adp + aeb + aec = 1 \implies$
$acm + aco + acn = 1 \implies$
$abl + abl = 1 \implies$
$zr + zq + abh = 1 \implies$
$abq + zd + abo = 1 \implies$
$acd + acc + xi + acb = 1 \implies$
$acp + acr + acq = 1 \implies$
$xw + xr + xv + xu = 1 \implies$
$zc + acg + ach = 1 \implies$
$zt + zs + xy = 1 \implies$
$ady + adw + zg = 1 \implies$
$abd + abc + yr + abb = 1 \implies$

$$
\begin{array}{l}
adi + x + adh + xa = 1 \implies \\
aak + aai + aad = 1 \implies \\
aba + zh + aay = 1 \implies \\
abg + ys + abe = 1 \implies \\
abs1 + yt + abr + zu = 1 \implies \\
abv + zn + abw + zm = 1 \implies \\
adl + adn = 1 \implies \\
acf + aca = 1 \implies \\
ads + aaq = 1 \implies \\
ada + ada = 1 \implies \\
aaf + aac + aag = 1 \implies \\
aal + acu + acs + act = 1 \implies \\
aas + xb + aat = 1 \implies \\
zk + zj + zi = 1 \implies \\
ack + acj + xc + aci = 1 \implies \\
aav + aah + xd = 1 \implies \\
abt + xo + abu + xn = 1 \implies \\
adc + abz + adc + abz = 1 \implies \\
xt + zz + aab + aaa = 1 \implies \\
adq + xl + adr + adb = 1 \implies \\
zf + yh + yi = 1 \implies \\
aao + aam + xe = 1 \implies \\
abk + aby + abj + abx = 1 \implies \\
yp + yp = 1 \implies \\
yl + yj + ym = 1 \implies \\
acw + acw + acw + acw = 1 \implies \\
adk + adg + adj + adf = 1 \implies \\
adv + xf + adu = 1 \implies \\
yd = 0 \lor yb = 0 \implies \\
xh = 0 \lor ye = 0 \implies \\
yy = 0 \lor za = 0 \implies \\
acc = 0 \lor xi = 0 \implies \\
xv = 0 \lor xr = 0 \implies \\
yr = 0 \lor abc = 0 \implies \\
zn = 0 \lor abw = 0 \implies \\
xo = 0 \lor abu = 0 \implies \\
xl = 0 \lor adr = 0 \implies \\
(yr + abd < abl \lor \\
\ yr + (abd + abb) < 1) \lor \\
yr + abd = abl \land \\
yr + (abd + abb) = 1 \implies \\
adb + adr < xn + abu \lor \\
adb + adr = xn + abu \implies \\
(abl < abt \lor abl < abt + xo) \lor \\
abl = abt \land abl = abt + xo \implies \\
yd + yc < abc + abd \lor \\
yd + yc = abc + abd \implies \\
aca < abb + yr \lor aca = abb + yr \implies \\
acb + acc < xu + xv \lor
\end{array}
$$

$$
\begin{array}{l}
acb + acc = xu + xv \Longrightarrow \\
(yq < xu + xr \vee \\
\ yq + zp < xu + (xr + xw)) \vee \\
yq = xu + xr \wedge \\
yq + zp = xu + (xr + xw) \Longrightarrow \\
(zw < xw \vee \\
\ zw < xw + xv \vee \\
\ zw + zy < xw + (xv + xu)) \vee \\
zw = xw \wedge \\
zw = xw + xv \wedge \\
zw + zy = xw + (xv + xu) \Longrightarrow \\
xs + yw < zs + zt \vee \\
xs + yw = zs + zt \Longrightarrow \\
aab + xt < ye + yf \vee \\
aab + xt = ye + yf \Longrightarrow \\
(ya + yb < yg + ye \vee \\
\ ya + (yb + yc) < yg + (ye + yf)) \vee \\
ya + yb = yg + ye \wedge \\
ya + (yb + yc) = yg + (ye + yf) \Longrightarrow \\
(xu + xv < acb + acc \vee \\
\ xu + (xv + xw) < acb + (acc + acd)) \vee \\
xu + xv = acb + acc \wedge \\
xu + (xv + xw) = acb + (acc + acd) \Longrightarrow \\
(zs < xz + zy \vee \\
\ zs + xy < xz + (zy + zw)) \vee \\
zs = xz + zy \wedge \\
zs + xy = xz + (zy + zw) \Longrightarrow \\
(zs + zt < xz + zy \vee \\
\ zs + (zt + xy) < xz + (zy + zw)) \vee \\
zs + zt = xz + zy \wedge \\
zs + (zt + xy) = xz + (zy + zw) \Longrightarrow \\
yg + ye < ya + yb \vee \\
yg + ye = ya + yb \Longrightarrow \\
(abd < yc \vee abd + abc < yc + yd) \vee \\
abd = yc \wedge abd + abc = yc + yd \Longrightarrow \\
(ye + yf < adr + adq \vee \\
\ ye + (yf + yg) < adr + (adq + adb)) \vee \\
ye + yf = adr + adq \wedge \\
ye + (yf + yg) = adr + (adq + adb) \Longrightarrow \\
yh + yi < ym + yj \vee \\
yh + yi = ym + yj \Longrightarrow \\
(abq < yl \vee abq + abo < yl + ym) \vee \\
abq = yl \wedge abq + abo = yl + ym \Longrightarrow \\
(yp < zp \vee \\
\ yp < zp + zo \vee 1 < zp + (zo + yq)) \vee \\
yp = zp \wedge \\
yp = zp + zo \wedge 1 = zp + (zo + yq) \Longrightarrow \\
(abb + yr < aca \vee \\
\ abb + (yr + abd) < aca + acf) \vee
\end{array}
$$

$$
\begin{array}{l}
abb + yr = aca \wedge \\
abb + (yr + abd) = aca + acf \Longrightarrow \\
adw + zg < abe + ys \vee \\
adw + zg = abe + ys \Longrightarrow \\
zd + abq < ys + abg \vee \\
zd + abq = ys + abg \Longrightarrow \\
yt + abs1 < aby + abk \vee \\
yt + abs1 = aby + abk \Longrightarrow \\
(yu < abx \vee \\
\ yu < abx + aby \vee \\
\ yu + yw < abx + (aby + abk)) \vee \\
yu = abx \wedge \\
yu = abx + aby \wedge \\
yu + yw = abx + (aby + abk) \Longrightarrow \\
aaf < adv \vee aaf = adv \Longrightarrow \\
abj + abk < yy + zb \vee \\
abj + abk = yy + zb \Longrightarrow \\
(abb < yz \vee \\
\ abb + abc < yz + za \vee \\
\ abb + (abc + abd) < yz + (za + zb)) \vee \\
abb = yz \wedge \\
abb + abc = yz + za \wedge \\
abb + (abc + abd) = yz + (za + zb) \Longrightarrow \\
(acg + zc < zd + abq \vee \\
\ acg + (zc + ach) \\
\ < zd + (abq + abo)) \vee \\
acg + zc = zd + abq \wedge \\
acg + (zc + ach) = \\
zd + (abq + abo) \Longrightarrow \\
zf < acm \vee zf = acm \Longrightarrow \\
(zg + ady < acn + acm \vee \\
\ zg + (ady + adw) \\
\ < acn + (acm + aco)) \vee \\
zg + ady = acn + acm \wedge \\
zg + (ady + adw) = \\
acn + (acm + aco) \Longrightarrow \\
aay + zh < zi + zj \vee \\
aay + zh = zi + zj \Longrightarrow \\
zy < zk \vee zy = zk \Longrightarrow \\
(adn < zm + zn \vee \\
\ adn + adl < zm + (zn + abv)) \vee \\
adn = zm + zn \wedge \\
adn + adl = zm + (zn + abv) \Longrightarrow \\
zo + zp < zs + zt \vee \\
zo + zp = zs + zt \Longrightarrow \\
zq + zr < zs + zt \vee \\
zq + zr = zs + zt \Longrightarrow \\
(aai < adi \vee aai < adi + adh) \vee \\
aai = adi \wedge aai = adi + adh \Longrightarrow
\end{array}
$$

$$
\begin{array}{l}
(abr < acj \vee \\
\ abr + (abs1 + zu) \\
\ < acj + (aci + ack)) \vee \\
abr = acj \wedge \\
abr + (abs1 + zu) = \\
acj + (aci + ack) \Longrightarrow \\
(abl < zw \vee 1 < zw + zy) \vee \\
abl = zw \wedge 1 = zw + zy \Longrightarrow \\
(zz + aaa < act + acu \vee \\
\ zz + (aaa + aab) \\
\ < act + (acu + aal)) \vee \\
zz + aaa = act + acu \wedge \\
zz + (aaa + aab) = \\
act + (acu + aal) \Longrightarrow \\
(aam < aac \vee aam + aao < aac + aaf) \vee \\
aam = aac \wedge aam + aao = aac + aaf \Longrightarrow \\
(aak < aaf \vee aak + aad < aaf + aag) \vee \\
aak = aaf \wedge aak + aad = aaf + aag \Longrightarrow \\
(aah < aai \vee aah + aav < aai + aak) \vee \\
aah = aai \wedge aah + aav = aai + aak \Longrightarrow \\
act + (acu + aal) < aam + aao \vee \\
act + (acu + aal) = aam + aao \Longrightarrow \\
(ads < aat \vee 1 < aat + aas) \vee \\
ads = aat \wedge 1 = aat + aas \Longrightarrow \\
(aba < aas \vee aba + aay < aas + aat) \vee \\
aba = aas \wedge aba + aay = aas + aat \Longrightarrow \\
acm < aav \vee acm = aav \Longrightarrow \\
(ada < aay \vee 1 < aay + aba) \vee \\
ada = aay \wedge 1 = aay + aba \Longrightarrow \\
abb + (abc + abd) < abe + abg \vee \\
abb + (abc + abd) = abe + abg \Longrightarrow \\
(abh < abj \vee abh < abj + abk) \vee \\
abh = abj \wedge abh = abj + abk \Longrightarrow \\
1 < abo + abq \vee 1 = abo + abq \Longrightarrow \\
(acj < abr \vee acj + aci < abr + abs1) \vee \\
acj = abr \wedge acj + aci = abr + abs1 \Longrightarrow \\
(abt < abv \vee abt + abu < abv + abw) \vee \\
abt = abv \wedge abt + abu = abv + abw \Longrightarrow \\
(abx < adc \vee abx + aby < adc + abz) \vee \\
abx = adc \wedge abx + aby = adc + abz \Longrightarrow \\
(acf < acd \vee \\
\ acf < acd + acc \vee \\
\ 1 < acd + (acc + acb)) \vee \\
acf = acd \wedge \\
acf = acd + acc \wedge \\
1 = acd + (acc + acb) \Longrightarrow \\
acc + acd < acf \vee acc + acd = acf \Longrightarrow \\
(acg < acq \vee acg + ach < acq + acr) \vee \\
acg = acq \wedge acg + ach = acq + acr \Longrightarrow
\end{array}
$$

$aci + (acj + ack) < acr + acp \vee$
$aci + (acj + ack) = acr + acp \Longrightarrow$
$(acm < acp \vee$
$\ acm + acn < acp + acq \vee$
$\ acm + (acn + aco)$
$\ < acp + (acq + acr)) \vee$
$acm = acp \wedge$
$acm + acn = acp + acq \wedge$
$acm + (acn + aco) =$
$acp + (acq + acr) \Longrightarrow$
$(acs + act < acw + acw \vee$
$\ acs + (act + acu)$
$\ < acw + (acw + acw)) \vee$
$acs + act = acw + acw \wedge$
$acs + (act + acu) =$
$acw + (acw + acw) \Longrightarrow$
$(ada < adb + adr \vee$
$\ 1 < adb + (adr + adq)) \vee$
$ada = adb + adr \wedge$
$1 = adb + (adr + adq) \Longrightarrow$
$(adc + adc < adf + adg \vee$
$\ adc + (adc + abz)$
$\ < adf + (adg + adk)) \vee$
$adc + adc = adf + adg \wedge$
$adc + (adc + abz) =$
$adf + (adg + adk) \Longrightarrow$
$adh + adi < adj + adk \vee$
$adh + adi = adj + adk \Longrightarrow$
$(adl < aec \vee 1 < aec + adp) \vee$
$adl = aec \wedge 1 = aec + adp \Longrightarrow$
$(adq < ads \vee adq + adr < ads) \vee$
$adq = ads \wedge adq + adr = ads \Longrightarrow$
$adu + adv < aed + aea \vee$
$adu + adv = aed + aea \Longrightarrow$
$(adw < aee \vee adw + ady < aee + aea) \vee$
$adw = aee \wedge adw + ady = aee + aea \Longrightarrow$
$(aeb < aed \vee aeb + aec < aed + aee) \vee$
$aeb = aed \wedge aeb + aec = aed + aee \Longrightarrow$
$False$
$\langle proof \rangle$

end

107 Numeral Syntax for Types

theory *Numeral-Type*
imports *Cardinality*
begin

107.1 Numeral Types

typedef *num0* = *UNIV* :: *nat set* ⟨*proof*⟩
typedef *num1* = *UNIV* :: *unit set* ⟨*proof*⟩

typedef *'a bit0* = {*0* ..< *2* ∗ *int CARD*(*'a*::*finite*)}
⟨*proof*⟩

typedef *'a bit1* = {*0* ..< *1* + *2* ∗ *int CARD*(*'a*::*finite*)}
⟨*proof*⟩

lemma *card-num0* [*simp*]: *CARD* (*num0*) = *0*
⟨*proof*⟩

lemma *infinite-num0*: ¬ *finite* (*UNIV* :: *num0 set*)
⟨*proof*⟩

lemma *card-num1* [*simp*]: *CARD*(*num1*) = *1*
⟨*proof*⟩

lemma *card-bit0* [*simp*]: *CARD*(*'a bit0*) = *2* ∗ *CARD*(*'a*::*finite*)
⟨*proof*⟩

lemma *card-bit1* [*simp*]: *CARD*(*'a bit1*) = *Suc* (*2* ∗ *CARD*(*'a*::*finite*))
⟨*proof*⟩

instance *num1* :: *finite*
⟨*proof*⟩

instance *bit0* :: (*finite*) *card2*
⟨*proof*⟩

instance *bit1* :: (*finite*) *card2*
⟨*proof*⟩

107.2 Locales for for modular arithmetic subtypes

locale *mod-type* =
fixes *n* :: *int*
and *Rep* :: *'a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*} ⇒ *int*
and *Abs* :: *int* ⇒ *'a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*}
assumes *type*: *type-definition Rep Abs* {*0*..<*n*}
and *size1*: *1* < *n*
and *zero-def*: *0* = *Abs 0*
and *one-def*: *1* = *Abs 1*
and *add-def*: *x* + *y* = *Abs* ((*Rep x* + *Rep y*) *mod n*)
and *mult-def*: *x* ∗ *y* = *Abs* ((*Rep x* ∗ *Rep y*) *mod n*)
and *diff-def*: *x* − *y* = *Abs* ((*Rep x* − *Rep y*) *mod n*)
and *minus-def*: − *x* = *Abs* ((− *Rep x*) *mod n*)
begin

lemma *size0*: $0 < n$
⟨*proof*⟩

lemmas *definitions* =
zero-def one-def add-def mult-def minus-def diff-def

lemma *Rep-less-n*: $Rep\ x < n$
⟨*proof*⟩

lemma *Rep-le-n*: $Rep\ x \leq n$
⟨*proof*⟩

lemma *Rep-inject-sym*: $x = y \longleftrightarrow Rep\ x = Rep\ y$
⟨*proof*⟩

lemma *Rep-inverse*: $Abs\ (Rep\ x) = x$
⟨*proof*⟩

lemma *Abs-inverse*: $m \in \{0..<n\} \Longrightarrow Rep\ (Abs\ m) = m$
⟨*proof*⟩

lemma *Rep-Abs-mod*: $Rep\ (Abs\ (m\ mod\ n)) = m\ mod\ n$
⟨*proof*⟩

lemma *Rep-Abs-0*: $Rep\ (Abs\ 0) = 0$
⟨*proof*⟩

lemma *Rep-0*: $Rep\ 0 = 0$
⟨*proof*⟩

lemma *Rep-Abs-1*: $Rep\ (Abs\ 1) = 1$
⟨*proof*⟩

lemma *Rep-1*: $Rep\ 1 = 1$
⟨*proof*⟩

lemma *Rep-mod*: $Rep\ x\ mod\ n = Rep\ x$
⟨*proof*⟩

lemmas *Rep-simps* =
Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1

lemma *comm-ring-1*: $OFCLASS('a, comm\text{-}ring\text{-}1\text{-}class)$
⟨*proof*⟩

end

locale *mod-ring* = *mod-type n Rep Abs*

for $n :: int$
and $Rep :: 'a::\{comm\text{-}ring\text{-}1\} \Rightarrow int$
and $Abs :: int \Rightarrow 'a::\{comm\text{-}ring\text{-}1\}$
begin

lemma *of-nat-eq*: $of\text{-}nat\ k = Abs\ (int\ k\ mod\ n)$
⟨*proof*⟩

lemma *of-int-eq*: $of\text{-}int\ z = Abs\ (z\ mod\ n)$
⟨*proof*⟩

lemma *Rep-numeral*:
$Rep\ (numeral\ w) = numeral\ w\ mod\ n$
⟨*proof*⟩

lemma *iszero-numeral*:
$iszero\ (numeral\ w::'a) \longleftrightarrow numeral\ w\ mod\ n = 0$
⟨*proof*⟩

lemma *cases*:
assumes *1*: $\bigwedge z.\ [\![(x::'a) = of\text{-}int\ z;\ 0 \le z;\ z < n]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *induct*:
$(\bigwedge z.\ [\![0 \le z;\ z < n]\!] \Longrightarrow P\ (of\text{-}int\ z)) \Longrightarrow P\ (x::'a)$
⟨*proof*⟩

end

107.3 Ring class instances

Unfortunately *ring-1* instance is not possible for *num1*, since 0 and 1 are not distinct.

instantiation *num1* :: {*comm-ring*,*comm-monoid-mult*,*numeral*}
begin

lemma *num1-eq-iff*: $(x::num1) = (y::num1) \longleftrightarrow True$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation
bit0 **and** *bit1* :: (*finite*) {*zero*,*one*,*plus*,*times*,*uminus*,*minus*}
begin

definition *Abs-bit0'* :: *int* ⇒ *'a bit0* **where**
  Abs-bit0' x = *Abs-bit0* (*x mod int CARD*(*'a bit0*))

definition *Abs-bit1'* :: *int* ⇒ *'a bit1* **where**
  Abs-bit1' x = *Abs-bit1* (*x mod int CARD*(*'a bit1*))

definition *0* = *Abs-bit0 0*
definition *1* = *Abs-bit0 1*
definition *x* + *y* = *Abs-bit0'* (*Rep-bit0 x* + *Rep-bit0 y*)
definition *x* ∗ *y* = *Abs-bit0'* (*Rep-bit0 x* ∗ *Rep-bit0 y*)
definition *x* − *y* = *Abs-bit0'* (*Rep-bit0 x* − *Rep-bit0 y*)
definition − *x* = *Abs-bit0'* (− *Rep-bit0 x*)

definition *0* = *Abs-bit1 0*
definition *1* = *Abs-bit1 1*
definition *x* + *y* = *Abs-bit1'* (*Rep-bit1 x* + *Rep-bit1 y*)
definition *x* ∗ *y* = *Abs-bit1'* (*Rep-bit1 x* ∗ *Rep-bit1 y*)
definition *x* − *y* = *Abs-bit1'* (*Rep-bit1 x* − *Rep-bit1 y*)
definition − *x* = *Abs-bit1'* (− *Rep-bit1 x*)

instance ⟨*proof*⟩

end

interpretation *bit0*:
  mod-type int CARD(*'a*::*finite bit0*)
    Rep-bit0 :: *'a*::*finite bit0* ⇒ *int*
    Abs-bit0 :: *int* ⇒ *'a*::*finite bit0*
⟨*proof*⟩

interpretation *bit1*:
  mod-type int CARD(*'a*::*finite bit1*)
    Rep-bit1 :: *'a*::*finite bit1* ⇒ *int*
    Abs-bit1 :: *int* ⇒ *'a*::*finite bit1*
⟨*proof*⟩

instance *bit0* :: (*finite*) *comm-ring-1*
  ⟨*proof*⟩

instance *bit1* :: (*finite*) *comm-ring-1*
  ⟨*proof*⟩

interpretation *bit0*:
  mod-ring int CARD(*'a*::*finite bit0*)
    Rep-bit0 :: *'a*::*finite bit0* ⇒ *int*
    Abs-bit0 :: *int* ⇒ *'a*::*finite bit0*
  ⟨*proof*⟩

interpretation *bit1*:

mod-ring int CARD('a::finite bit1)
Rep-bit1 :: 'a::finite bit1 ⇒ int
Abs-bit1 :: int ⇒ 'a::finite bit1
⟨*proof*⟩

Set up cases, induction, and arithmetic

lemmas *bit0-cases* [*case-names of-int*, *cases type*: *bit0*] = *bit0.cases*
lemmas *bit1-cases* [*case-names of-int*, *cases type*: *bit1*] = *bit1.cases*

lemmas *bit0-induct* [*case-names of-int*, *induct type*: *bit0*] = *bit0.induct*
lemmas *bit1-induct* [*case-names of-int*, *induct type*: *bit1*] = *bit1.induct*

lemmas *bit0-iszero-numeral* [*simp*] = *bit0.iszero-numeral*
lemmas *bit1-iszero-numeral* [*simp*] = *bit1.iszero-numeral*

lemmas [*simp*] = *eq-numeral-iff-iszero* [**where** *'a='a bit0*] **for** *dummy :: 'a::finite*
lemmas [*simp*] = *eq-numeral-iff-iszero* [**where** *'a='a bit1*] **for** *dummy :: 'a::finite*

107.4 Order instances

instantiation *bit0* **and** *bit1* :: (*finite*) *linorder* **begin**
definition $a < b \longleftrightarrow$ *Rep-bit0 a < Rep-bit0 b*
definition $a \leq b \longleftrightarrow$ *Rep-bit0 a ≤ Rep-bit0 b*
definition $a < b \longleftrightarrow$ *Rep-bit1 a < Rep-bit1 b*
definition $a \leq b \longleftrightarrow$ *Rep-bit1 a ≤ Rep-bit1 b*

instance
⟨*proof*⟩
end

lemma (**in** *preorder*) *tranclp-less*: *op* $<^{++}$ = *op* <
⟨*proof*⟩

instance *bit0* **and** *bit1* :: (*finite*) *wellorder*
⟨*proof*⟩

107.5 Code setup and type classes for code generation

Code setup for *num0* and *num1*

definition *Num0* :: *num0* **where** *Num0* = *Abs-num0 0*
code-datatype *Num0*

instantiation *num0* :: *equal* **begin**
definition *equal-num0* :: *num0* ⇒ *num0* ⇒ *bool*
where *equal-num0* = *op* =
instance ⟨*proof*⟩
end

lemma *equal-num0-code* [*code*]:

equal-class.equal Num0 Num0 = True
⟨*proof*⟩

code-datatype *1* :: *num1*

instantiation *num1* :: *equal* **begin**
definition *equal-num1* :: *num1* ⇒ *num1* ⇒ *bool*
where *equal-num1 = op =*
instance ⟨*proof*⟩
end

lemma *equal-num1-code* [*code*]:
equal-class.equal (*1* :: *num1*) *1 = True*
⟨*proof*⟩

instantiation *num1* :: *enum* **begin**
definition *enum-class.enum* = [*1* :: *num1*]
definition *enum-class.enum-all P = P* (*1* :: *num1*)
definition *enum-class.enum-ex P = P* (*1* :: *num1*)
instance
⟨*proof*⟩
end

instantiation *num0* **and** *num1* :: *card-UNIV* **begin**
definition *finite-UNIV = Phantom*(*num0*) *False*
definition *card-UNIV = Phantom*(*num0*) *0*
definition *finite-UNIV = Phantom*(*num1*) *True*
definition *card-UNIV = Phantom*(*num1*) *1*
instance
⟨*proof*⟩
end

Code setup for *'a bit0* and *'a bit1*

declare
bit0.Rep-inverse[*code abstype*]
bit0.Rep-0[*code abstract*]
bit0.Rep-1[*code abstract*]

lemma *Abs-bit0'-code* [*code abstract*]:
Rep-bit0 (*Abs-bit0' x* :: *'a* :: *finite bit0*) = *x mod int* (*CARD*(*'a bit0*))
⟨*proof*⟩

lemma *inj-on-Abs-bit0*:
inj-on (*Abs-bit0* :: *int* ⇒ *'a bit0*) {*0..<2* ∗ *int CARD*(*'a* :: *finite*)}
⟨*proof*⟩

declare
bit1.Rep-inverse[*code abstype*]
bit1.Rep-0[*code abstract*]

bit1.Rep-1[*code abstract*]

lemma *Abs-bit1'-code* [*code abstract*]:
Rep-bit1 (*Abs-bit1'* *x* :: *'a* :: *finite bit1*) = *x mod int* (*CARD*(*'a bit1*))
⟨*proof*⟩

lemma *inj-on-Abs-bit1*:
inj-on (*Abs-bit1* :: *int* ⇒ *'a bit1*) {*0..<1* + *2* ∗ *int CARD*(*'a* :: *finite*)}
⟨*proof*⟩

instantiation *bit0* **and** *bit1* :: (*finite*) *equal* **begin**

definition *equal-class.equal x y* ⟷ *Rep-bit0 x* = *Rep-bit0 y*
definition *equal-class.equal x y* ⟷ *Rep-bit1 x* = *Rep-bit1 y*

instance
⟨*proof*⟩

end

instantiation *bit0* :: (*finite*) *enum* **begin**
definition (*enum-class.enum* :: *'a bit0 list*) = *map* (*Abs-bit0'* ∘ *int*) (*upt 0* (*CARD*(*'a bit0*)))
definition *enum-class.enum-all P* = (∀ *b* :: *'a bit0* ∈ *set enum-class.enum*. *P b*)
definition *enum-class.enum-ex P* = (∃ *b* :: *'a bit0* ∈ *set enum-class.enum*. *P b*)

instance
⟨*proof*⟩

end

instantiation *bit1* :: (*finite*) *enum* **begin**
definition (*enum-class.enum* :: *'a bit1 list*) = *map* (*Abs-bit1'* ∘ *int*) (*upt 0* (*CARD*(*'a bit1*)))
definition *enum-class.enum-all P* = (∀ *b* :: *'a bit1* ∈ *set enum-class.enum*. *P b*)
definition *enum-class.enum-ex P* = (∃ *b* :: *'a bit1* ∈ *set enum-class.enum*. *P b*)

instance
⟨*proof*⟩

end

instantiation *bit0* **and** *bit1* :: (*finite*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a bit0*) *True*
definition *finite-UNIV* = *Phantom*(*'a bit1*) *True*
instance ⟨*proof*⟩
end

instantiation *bit0* **and** *bit1* :: ({*finite*,*card-UNIV*}) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*('*a bit0*) (*2* ∗ *of-phantom* (*card-UNIV* :: '*a card-UNIV*))
definition *card-UNIV* = *Phantom*('*a bit1*) (*1* + *2* ∗ *of-phantom* (*card-UNIV* :: '*a card-UNIV*))
instance ⟨*proof*⟩
end

107.6 Syntax

syntax
-*NumeralType* :: *num-token* => *type* (-)
-*NumeralType0* :: *type* (*0*)
-*NumeralType1* :: *type* (*1*)

translations
(*type*) *1* == (*type*) *num1*
(*type*) *0* == (*type*) *num0*

⟨*ML*⟩

107.7 Examples

lemma *CARD*(*0*) = *0* ⟨*proof*⟩
lemma *CARD*(*17*) = *17* ⟨*proof*⟩
lemma *8* ∗ *11* ^ *3* − *6* = (*2*::*5*) ⟨*proof*⟩

end

108 Assigning lengths to types by type classes

theory *Type-Length*
imports *Numeral-Type*
begin

The aim of this is to allow any type as index type, but to provide a default instantiation for numeral types. This independence requires some duplication with the definitions in `Numeral_Type.thy`.

class *len0* =
fixes *len-of* :: '*a itself* ⇒ *nat*

syntax -*type-length* :: *type* ⇒ *nat* ((*1LENGTH*/(*1*'(-'))))

translations *LENGTH*('*a*) ⇀
CONST len-of (*CONST Pure.type* :: '*a itself*)

⟨*ML*⟩

Some theorems are only true on words with length greater 0.

class *len* = *len0* +
  assumes *len-gt-0* [*iff*]: *0* < *LENGTH*('*a*)

instantiation *num0* **and** *num1* :: *len0*
begin

definition *len-num0*: *len-of* (- :: *num0 itself*) = *0*
definition *len-num1*: *len-of* (- :: *num1 itself*) = *1*

instance ⟨*proof*⟩

end

instantiation *bit0* **and** *bit1* :: (*len0*) *len0*
begin

definition *len-bit0*: *len-of* (- :: '*a*::*len0 bit0 itself*) = *2* ∗ *LENGTH*('*a*)
definition *len-bit1*: *len-of* (- :: '*a*::*len0 bit1 itself*) = *2* ∗ *LENGTH*('*a*) + *1*

instance ⟨*proof*⟩

end

lemmas *len-of-numeral-defs* [*simp*] = *len-num0 len-num1 len-bit0 len-bit1*

instance *num1* :: *len*
  ⟨*proof*⟩
instance *bit0* :: (*len*) *len*
  ⟨*proof*⟩
instance *bit1* :: (*len0*) *len*
  ⟨*proof*⟩

end

109 Proof of concept for algebraically founded bit word types

theory *Word-Type*
  imports
    Main
    ~~/*src*/*HOL*/*Library*/*Type-Length*
begin

109.1 Truncating bit representations of numeric types

class *semiring-bits* = *semiring-div-parity* +
  assumes *semiring-bits*: (*1* + *2* ∗ *a*) *mod of-nat* (*2* ∗ *n*) = *1* + *2* ∗ (*a mod of-nat* *n*)

begin

definition *bitrunc* :: *nat* ⇒ *'a* ⇒ *'a*
  where *bitrunc-eq-mod*: *bitrunc n a* = *a mod of-nat* (*2* ^ *n*)

lemma *bitrunc-bitrunc* [*simp*]:
  bitrunc n (*bitrunc n a*) = *bitrunc n a*
  ⟨*proof*⟩

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

lemma *bitrunc-Suc* [*simp*]:
  bitrunc (*Suc n*) *a* = *bitrunc n* (*a div 2*) ∗ *2* + *a mod 2*
⟨*proof*⟩

lemma *bitrunc-of-0* [*simp*]:
  bitrunc n 0 = *0*
  ⟨*proof*⟩

lemma *bitrunc-plus*:
  bitrunc n (*bitrunc n a* + *bitrunc n b*) = *bitrunc n* (*a* + *b*)
  ⟨*proof*⟩

lemma *bitrunc-of-1-eq-0-iff* [*simp*]:
  bitrunc n 1 = *0* ⟷ *n* = *0*
  ⟨*proof*⟩

end

instance *nat* :: *semiring-bits*
  ⟨*proof*⟩

instance *int* :: *semiring-bits*
  ⟨*proof*⟩

lemma *bitrunc-uminus*:
  fixes *k* :: *int*
  shows *bitrunc n* (− (*bitrunc n k*)) = *bitrunc n* (− *k*)
  ⟨*proof*⟩

lemma *bitrunc-minus*:
  fixes *k l* :: *int*
  shows *bitrunc n* (*bitrunc n k* − *bitrunc n l*) = *bitrunc n* (*k* − *l*)
  ⟨*proof*⟩

lemma *bitrunc-nonnegative* [*simp*]:
  fixes *k* :: *int*

shows *bitrunc n k* ≥ *0*
⟨*proof*⟩

definition *signed-bitrunc* :: *nat* ⇒ *int* ⇒ *int*
where *signed-bitrunc-eq-bitrunc*:
signed-bitrunc n k = *bitrunc* (*Suc n*) (*k* + *2* ^ *n*) − *2* ^ *n*

lemma *signed-bitrunc-eq-bitrunc′*:
assumes *n* > *0*
shows *signed-bitrunc* (*n* − *Suc 0*) *k* = *bitrunc n* (*k* + *2* ^ (*n* − *1*)) − *2* ^ (*n* − *1*)
⟨*proof*⟩

lemma *signed-bitrunc-0* [*simp*]:
signed-bitrunc 0 k = − (*k mod 2*)
⟨*proof*⟩

lemma *signed-bitrunc-Suc* [*simp*]:
signed-bitrunc (*Suc n*) *k* = *signed-bitrunc n* (*k div 2*) ∗ *2* + *k mod 2*
⟨*proof*⟩

lemma *signed-bitrunc-of-0* [*simp*]:
signed-bitrunc n 0 = *0*
⟨*proof*⟩

lemma *signed-bitrunc-of-minus-1* [*simp*]:
signed-bitrunc n (− *1*) = − *1*
⟨*proof*⟩

lemma *signed-bitrunc-eq-iff-bitrunc-eq*:
assumes *n* > *0*
shows *signed-bitrunc* (*n* − *Suc 0*) *k* = *signed-bitrunc* (*n* − *Suc 0*) *l* ⟷ *bitrunc n k* = *bitrunc n l* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

109.2 Bit strings as quotient type

109.2.1 Basic properties

quotient-type (**overloaded**) *′a word* = *int* / *λk l*. *bitrunc LENGTH*(*′a*) *k* = *bitrunc LENGTH*(*′a*::*len0*) *l*
⟨*proof*⟩

instantiation *word* :: (*len0*) {*semiring-numeral*, *comm-semiring-0*, *comm-ring*}
begin

lift-definition *zero-word* :: *′a word*
is *0*
⟨*proof*⟩

lift-definition *one-word* :: *'a word*
is *1*
⟨*proof*⟩

lift-definition *plus-word* :: *'a word* ⇒ *'a word* ⇒ *'a word*
is *plus*
⟨*proof*⟩

lift-definition *uminus-word* :: *'a word* ⇒ *'a word*
is *uminus*
⟨*proof*⟩

lift-definition *minus-word* :: *'a word* ⇒ *'a word* ⇒ *'a word*
is *minus*
⟨*proof*⟩

lift-definition *times-word* :: *'a word* ⇒ *'a word* ⇒ *'a word*
is *times*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *word* :: (*len*) *comm-ring-1*
⟨*proof*⟩

109.2.2 Conversions

lemma [*transfer-rule*]:
rel-fun HOL.eq pcr-word int of-nat
⟨*proof*⟩

lemma [*transfer-rule*]:
rel-fun HOL.eq pcr-word (λ*k*. *k*) *of-int*
⟨*proof*⟩

context *semiring-1*
begin

lift-definition *unsigned* :: *'b*::*len0 word* ⇒ *'a*
is *of-nat* ∘ *nat* ∘ *bitrunc LENGTH*(*'b*)
⟨*proof*⟩

lemma *unsigned-0* [*simp*]:
unsigned 0 = 0
⟨*proof*⟩

end

context *semiring-char-0*
begin

lemma *word-eq-iff-unsigned*:
a = b ⟷ unsigned a = unsigned b
⟨*proof*⟩

end

context *ring-1*
begin

lift-definition *signed* :: *'b::len word ⇒ 'a*
is *of-int ∘ signed-bitrunc (LENGTH('b) − 1)*
⟨*proof*⟩

lemma *signed-0* [*simp*]:
signed 0 = 0
⟨*proof*⟩

end

lemma *unsigned-of-nat* [*simp*]:
unsigned (of-nat n :: 'a word) = bitrunc LENGTH('a::len) n
⟨*proof*⟩

lemma *of-nat-unsigned* [*simp*]:
of-nat (unsigned a) = a
⟨*proof*⟩

lemma *of-int-unsigned* [*simp*]:
of-int (unsigned a) = a
⟨*proof*⟩

context *ring-char-0*
begin

lemma *word-eq-iff-signed*:
a = b ⟷ signed a = signed b
⟨*proof*⟩

end

lemma *signed-of-int* [*simp*]:
signed (of-int k :: 'a word) = signed-bitrunc (LENGTH('a::len) − 1) k
⟨*proof*⟩

lemma *of-int-signed* [*simp*]:
of-int (*signed a*) = *a*
⟨*proof*⟩

109.2.3 Properties

109.2.4 Division

instantiation *word* :: (*len0*) *modulo*
begin

lift-definition *divide-word* :: *'a word* ⇒ *'a word* ⇒ *'a word*
is λ*a b*. *bitrunc LENGTH*(*'a*) *a div bitrunc LENGTH*(*'a*) *b*
⟨*proof*⟩

lift-definition *modulo-word* :: *'a word* ⇒ *'a word* ⇒ *'a word*
is λ*a b*. *bitrunc LENGTH*(*'a*) *a mod bitrunc LENGTH*(*'a*) *b*
⟨*proof*⟩

instance ⟨*proof*⟩

end

109.2.5 Orderings

instantiation *word* :: (*len0*) *linorder*
begin

lift-definition *less-eq-word* :: *'a word* ⇒ *'a word* ⇒ *bool*
is λ*a b*. *bitrunc LENGTH*(*'a*) *a* ≤ *bitrunc LENGTH*(*'a*) *b*
⟨*proof*⟩

lift-definition *less-word* :: *'a word* ⇒ *'a word* ⇒ *bool*
is λ*a b*. *bitrunc LENGTH*(*'a*) *a* < *bitrunc LENGTH*(*'a*) *b*
⟨*proof*⟩

instance
⟨*proof*⟩

end

context *linordered-semidom*
begin

lemma *word-less-eq-iff-unsigned*:
a ≤ *b* ⟷ *unsigned a* ≤ *unsigned b*
⟨*proof*⟩

lemma *word-less-iff-unsigned*:
a < *b* ⟷ *unsigned a* < *unsigned b*

⟨*proof*⟩

end

end

110 Meson test cases

theory *Meson-Test*
imports *Main*
begin

WARNING: there are many potential conflicts between variables used below and constants declared in HOL!

hide-const (**open**) *implies union inter subset quotient sum*

Test data for the MESON proof procedure (Excludes the equality problems 51, 52, 56, 58)

110.1 Interactive examples

lemma *problem-25*:
(∃ *x*. *P x*) & (∀ *x*. *L x* −−> ~ (*M x* & *R x*)) & (∀ *x*. *P x* −−> (*M x* & *L x*)) & ((∀ *x*. *P x* −−> *Q x*) | (∃ *x*. *P x* & *R x*)) −−> (∃ *x*. *Q x* & *P x*)
⟨*proof*⟩
⟨*ML*⟩
⟨*proof*⟩

lemma *problem-26*:
((∃ *x*. *p x*) = (∃ *x*. *q x*)) & (∀ *x*. ∀ *y*. *p x* & *q y* −−> (*r x* = *s y*)) −−> ((∀ *x*. *p x* −−> *r x*) = (∀ *x*. *q x* −−> *s x*))
⟨*proof*⟩
⟨*ML*⟩
⟨*proof*⟩

lemma *problem-43*: — NOW PROVED AUTOMATICALLY!!
(∀ *x*. ∀ *y*. *q x y* = (∀ *z*. *p z x* = (*p z y*::*bool*))) −−> (∀ *x*. (∀ *y*. *q x y* = (*q y x*::*bool*)))
⟨*proof*⟩
⟨*ML*⟩
⟨*proof*⟩

MORE and MUCH HARDER test data for the MESON proof procedure (courtesy John Harrison).

abbreviation *EQU001-0-ax equal* ≡ (∀ *X*. *equal*(*X*::'*a*,*X*)) &
(∀ *Y X*. *equal*(*X*::'*a*,*Y*) −−> *equal*(*Y*::'*a*,*X*)) &
(∀ *Y X Z*. *equal*(*X*::'*a*,*Y*) & *equal*(*Y*::'*a*,*Z*) −−> *equal*(*X*::'*a*,*Z*))

abbreviation *BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum* $\equiv$
$(\forall\ X\ Y.\ sum(X::'a,Y,add(X::'a,Y)))$ &
$(\forall\ X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall\ Y\ X\ Z.\ sum(X::'a,Y,Z) \longrightarrow sum(Y::'a,X,Z))$ &
$(\forall\ Y\ X\ Z.\ product(X::'a,Y,Z) \longrightarrow product(Y::'a,X,Z))$ &
$(\forall\ X.\ sum(additive\text{-}identity::'a,X,X))$ &
$(\forall\ X.\ sum(X::'a,additive\text{-}identity,X))$ &
$(\forall\ X.\ product(multiplicative\text{-}identity::'a,X,X))$ &
$(\forall\ X.\ product(X::'a,multiplicative\text{-}identity,X))$ &
$(\forall\ Y\ Z\ X\ V3\ V1\ V2\ V4.\ product(X::'a,Y,V1)\ \&\ product(X::'a,Z,V2)\ \&\ sum(Y::'a,Z,V3)\ \&\ product(X::'a,V3,V4) \longrightarrow sum(V1::'a,V2,V4))$ &
$(\forall\ Y\ Z\ V1\ V2\ X\ V3\ V4.\ product(X::'a,Y,V1)\ \&\ product(X::'a,Z,V2)\ \&\ sum(Y::'a,Z,V3)\ \&\ sum(V1::'a,V2,V4) \longrightarrow product(X::'a,V3,V4))$ &
$(\forall\ Y\ Z\ V3\ X\ V1\ V2\ V4.\ product(Y::'a,X,V1)\ \&\ product(Z::'a,X,V2)\ \&\ sum(Y::'a,Z,V3)\ \&\ product(V3::'a,X,V4) \longrightarrow sum(V1::'a,V2,V4))$ &
$(\forall\ Y\ Z\ V1\ V2\ V3\ X\ V4.\ product(Y::'a,X,V1)\ \&\ product(Z::'a,X,V2)\ \&\ sum(Y::'a,Z,V3)\ \&\ sum(V1::'a,V2,V4) \longrightarrow product(V3::'a,X,V4))$ &
$(\forall\ Y\ Z\ X\ V3\ V1\ V2\ V4.\ sum(X::'a,Y,V1)\ \&\ sum(X::'a,Z,V2)\ \&\ product(Y::'a,Z,V3)\ \&\ sum(X::'a,V3,V4) \longrightarrow product(V1::'a,V2,V4))$ &
$(\forall\ Y\ Z\ V1\ V2\ X\ V3\ V4.\ sum(X::'a,Y,V1)\ \&\ sum(X::'a,Z,V2)\ \&\ product(Y::'a,Z,V3)\ \&\ product(V1::'a,V2,V4) \longrightarrow sum(X::'a,V3,V4))$ &
$(\forall\ Y\ Z\ V3\ X\ V1\ V2\ V4.\ sum(Y::'a,X,V1)\ \&\ sum(Z::'a,X,V2)\ \&\ product(Y::'a,Z,V3)\ \&\ sum(V3::'a,X,V4) \longrightarrow product(V1::'a,V2,V4))$ &
$(\forall\ Y\ Z\ V1\ V2\ V3\ X\ V4.\ sum(Y::'a,X,V1)\ \&\ sum(Z::'a,X,V2)\ \&\ product(Y::'a,Z,V3)\ \&\ product(V1::'a,V2,V4) \longrightarrow sum(V3::'a,X,V4))$ &
$(\forall\ X.\ sum(INVERSE(X),X,multiplicative\text{-}identity))$ &
$(\forall\ X.\ sum(X::'a,INVERSE(X),multiplicative\text{-}identity))$ &
$(\forall\ X.\ product(INVERSE(X),X,additive\text{-}identity))$ &
$(\forall\ X.\ product(X::'a,INVERSE(X),additive\text{-}identity))$ &
$(\forall\ X\ Y\ U\ V.\ sum(X::'a,Y,U)\ \&\ sum(X::'a,Y,V) \longrightarrow equal(U::'a,V))$ &
$(\forall\ X\ Y\ U\ V.\ product(X::'a,Y,U)\ \&\ product(X::'a,Y,V) \longrightarrow equal(U::'a,V))$

abbreviation *BOO002-0-eq INVERSE multiply add product sum equal* $\equiv$
$(\forall\ X\ Y\ W\ Z.\ equal(X::'a,Y)\ \&\ sum(X::'a,W,Z) \longrightarrow sum(Y::'a,W,Z))$ &
$(\forall\ X\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ sum(W::'a,X,Z) \longrightarrow sum(W::'a,Y,Z))$ &
$(\forall\ X\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ sum(W::'a,Z,X) \longrightarrow sum(W::'a,Z,Y))$ &
$(\forall\ X\ Y\ W\ Z.\ equal(X::'a,Y)\ \&\ product(X::'a,W,Z) \longrightarrow product(Y::'a,W,Z))$
&
$(\forall\ X\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ product(W::'a,X,Z) \longrightarrow product(W::'a,Y,Z))$
&
$(\forall\ X\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ product(W::'a,Z,X) \longrightarrow product(W::'a,Z,Y))$
&
$(\forall\ X\ Y\ W.\ equal(X::'a,Y) \longrightarrow equal(add(X::'a,W),add(Y::'a,W)))$ &
$(\forall\ X\ W\ Y.\ equal(X::'a,Y) \longrightarrow equal(add(W::'a,X),add(W::'a,Y)))$ &
$(\forall\ X\ Y\ W.\ equal(X::'a,Y) \longrightarrow equal(multiply(X::'a,W),multiply(Y::'a,W)))$
&
$(\forall\ X\ W\ Y.\ equal(X::'a,Y) \longrightarrow equal(multiply(W::'a,X),multiply(W::'a,Y)))$
&

$(\forall\, X\ Y.\ equal(X::'a,Y)\ --> equal(INVERSE(X),INVERSE(Y)))$

lemma *BOO003-1*:
EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
$(\sim product(x::'a,x,x))\ --> False$
⟨*proof*⟩

lemma *BOO004-1*:
EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
$(\sim sum(x::'a,x,x))\ --> False$
⟨*proof*⟩

lemma *BOO005-1*:
EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
$(\sim sum(x::'a,multiplicative\text{-}identity,multiplicative\text{-}identity))\ --> False$
⟨*proof*⟩

lemma *BOO006-1*:
EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
$(\sim product(x::'a,additive\text{-}identity,additive\text{-}identity))\ --> False$
⟨*proof*⟩

lemma *BOO011-1*:
EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
$(\sim equal(INVERSE(additive\text{-}identity),multiplicative\text{-}identity))\ --> False$
⟨*proof*⟩

abbreviation *CAT003-0-ax f1 compos codomain domain equal there-exists equivalent* ≡

$(\forall\ Y\ X.\ equivalent(X::'a,Y)\ --{>}\ there\text{-}exists(X))\ \&$
$(\forall\ X\ Y.\ equivalent(X::'a,Y)\ --{>}\ equal(X::'a,Y))\ \&$
$(\forall\ X\ Y.\ there\text{-}exists(X)\ \&\ equal(X::'a,Y)\ --{>}\ equivalent(X::'a,Y))\ \&$
$(\forall\ X.\ there\text{-}exists(domain(X))\ --{>}\ there\text{-}exists(X))\ \&$
$(\forall\ X.\ there\text{-}exists(codomain(X))\ --{>}\ there\text{-}exists(X))\ \&$
$(\forall\ Y\ X.\ there\text{-}exists(compos(X::'a,Y))\ --{>}\ there\text{-}exists(domain(X)))\ \&$
$(\forall\ X\ Y.\ there\text{-}exists(compos(X::'a,Y))\ --{>}\ equal(domain(X),codomain(Y)))$
$\&$
$(\forall\ X\ Y.\ there\text{-}exists(domain(X))\ \&\ equal(domain(X),codomain(Y))\ --{>}\ there\text{-}exists(compos(X::'a,Y)))$
$\&$
$(\forall\ X\ Y\ Z.\ equal(compos(X::'a,compos(Y::'a,Z)),compos(compos(X::'a,Y),Z)))$
$\&$
$(\forall\ X.\ equal(compos(X::'a,domain(X)),X))\ \&$
$(\forall\ X.\ equal(compos(codomain(X),X),X))\ \&$
$(\forall\ X\ Y.\ equivalent(X::'a,Y)\ --{>}\ there\text{-}exists(Y))\ \&$
$(\forall\ X\ Y.\ there\text{-}exists(X)\ \&\ there\text{-}exists(Y)\ \&\ equal(X::'a,Y)\ --{>}\ equivalent(X::'a,Y))$
$\&$
$(\forall\ Y\ X.\ there\text{-}exists(compos(X::'a,Y))\ --{>}\ there\text{-}exists(codomain(X)))\ \&$
$(\forall\ X\ Y.\ there\text{-}exists(f1(X::'a,Y))\ |\ equal(X::'a,Y))\ \&$
$(\forall\ X\ Y.\ equal(X::'a,f1(X::'a,Y))\ |\ equal(Y::'a,f1(X::'a,Y))\ |\ equal(X::'a,Y))$
$\&$
$(\forall\ X\ Y.\ equal(X::'a,f1(X::'a,Y))\ \&\ equal(Y::'a,f1(X::'a,Y))\ --{>}\ equal(X::'a,Y))$

abbreviation *CAT003-0-eq f1 compos codomain domain equivalent there-exists equal* $\equiv$
$(\forall\ X\ Y.\ equal(X::'a,Y)\ \&\ there\text{-}exists(X)\ --{>}\ there\text{-}exists(Y))\ \&$
$(\forall\ X\ Y\ Z.\ equal(X::'a,Y)\ \&\ equivalent(X::'a,Z)\ --{>}\ equivalent(Y::'a,Z))\ \&$
$(\forall\ X\ Z\ Y.\ equal(X::'a,Y)\ \&\ equivalent(Z::'a,X)\ --{>}\ equivalent(Z::'a,Y))\ \&$
$(\forall\ X\ Y.\ equal(X::'a,Y)\ --{>}\ equal(domain(X),domain(Y)))\ \&$
$(\forall\ X\ Y.\ equal(X::'a,Y)\ --{>}\ equal(codomain(X),codomain(Y)))\ \&$
$(\forall\ X\ Y\ Z.\ equal(X::'a,Y)\ --{>}\ equal(compos(X::'a,Z),compos(Y::'a,Z)))\ \&$
$(\forall\ X\ Z\ Y.\ equal(X::'a,Y)\ --{>}\ equal(compos(Z::'a,X),compos(Z::'a,Y)))\ \&$
$(\forall\ A\ B\ C.\ equal(A::'a,B)\ --{>}\ equal(f1(A::'a,C),f1(B::'a,C)))\ \&$
$(\forall\ D\ F'\ E.\ equal(D::'a,E)\ --{>}\ equal(f1(F'::'a,D),f1(F'::'a,E)))$

lemma *CAT001-3*:
EQU001-0-ax equal $\&$
CAT003-0-ax f1 compos codomain domain equal there-exists equivalent $\&$
CAT003-0-eq f1 compos codomain domain equivalent there-exists equal $\&$
$(there\text{-}exists(compos(a::'a,b)))\ \&$
$(\forall\ Y\ X\ Z.\ equal(compos(compos(a::'a,b),X),Y)\ \&\ equal(compos(compos(a::'a,b),Z),Y)$
$--{>}\ equal(X::'a,Z))\ \&$
$(there\text{-}exists(compos(b::'a,h)))\ \&$
$(equal(compos(b::'a,h),compos(b::'a,g)))\ \&$
$(\sim equal(h::'a,g))\ --{>}\ False$
$\langle proof\rangle$

lemma *CAT003-3*:
EQU001-0-ax equal &
CAT003-0-ax f1 compos codomain domain equal there-exists equivalent &
CAT003-0-eq f1 compos codomain domain equivalent there-exists equal &
(*there-exists*(*compos*(*a*::'*a*,*b*))) &
(∀ *Y X Z*. *equal*(*compos*(*X*::'*a*,*compos*(*a*::'*a*,*b*)),*Y*) & *equal*(*compos*(*Z*::'*a*,*compos*(*a*::'*a*,*b*)),*Y*) −−> *equal*(*X*::'*a*,*Z*)) &
(*there-exists*(*h*)) &
(*equal*(*compos*(*h*::'*a*,*a*),*compos*(*g*::'*a*,*a*))) &
(~*equal*(*g*::'*a*,*h*)) −−> *False*
⟨*proof*⟩

abbreviation *CAT001-0-ax equal codomain domain identity-map compos product defined* ≡
(∀ *X Y*. *defined*(*X*::'*a*,*Y*) −−> *product*(*X*::'*a*,*Y*,*compos*(*X*::'*a*,*Y*))) &
(∀ *Z X Y*. *product*(*X*::'*a*,*Y*,*Z*) −−> *defined*(*X*::'*a*,*Y*)) &
(∀ *X Xy Y Z*. *product*(*X*::'*a*,*Y*,*Xy*) & *defined*(*Xy*::'*a*,*Z*) −−> *defined*(*Y*::'*a*,*Z*)) &
(∀ *Y Xy Z X Yz*. *product*(*X*::'*a*,*Y*,*Xy*) & *product*(*Y*::'*a*,*Z*,*Yz*) & *defined*(*Xy*::'*a*,*Z*) −−> *defined*(*X*::'*a*,*Yz*)) &
(∀ *Xy Y Z X Yz Xyz*. *product*(*X*::'*a*,*Y*,*Xy*) & *product*(*Xy*::'*a*,*Z*,*Xyz*) & *product*(*Y*::'*a*,*Z*,*Yz*) −−> *product*(*X*::'*a*,*Yz*,*Xyz*)) &
(∀ *Z Yz X Y*. *product*(*Y*::'*a*,*Z*,*Yz*) & *defined*(*X*::'*a*,*Yz*) −−> *defined*(*X*::'*a*,*Y*)) &
(∀ *Y X Yz Xy Z*. *product*(*Y*::'*a*,*Z*,*Yz*) & *product*(*X*::'*a*,*Y*,*Xy*) & *defined*(*X*::'*a*,*Yz*) −−> *defined*(*Xy*::'*a*,*Z*)) &
(∀ *Yz X Y Xy Z Xyz*. *product*(*Y*::'*a*,*Z*,*Yz*) & *product*(*X*::'*a*,*Yz*,*Xyz*) & *product*(*X*::'*a*,*Y*,*Xy*) −−> *product*(*Xy*::'*a*,*Z*,*Xyz*)) &
(∀ *Y X Z*. *defined*(*X*::'*a*,*Y*) & *defined*(*Y*::'*a*,*Z*) & *identity-map*(*Y*) −−> *defined*(*X*::'*a*,*Z*)) &
(∀ *X*. *identity-map*(*domain*(*X*))) &
(∀ *X*. *identity-map*(*codomain*(*X*))) &
(∀ *X*. *defined*(*X*::'*a*,*domain*(*X*))) &
(∀ *X*. *defined*(*codomain*(*X*),*X*)) &
(∀ *X*. *product*(*X*::'*a*,*domain*(*X*),*X*)) &
(∀ *X*. *product*(*codomain*(*X*),*X*,*X*)) &
(∀ *X Y*. *defined*(*X*::'*a*,*Y*) & *identity-map*(*X*) −−> *product*(*X*::'*a*,*Y*,*Y*)) &
(∀ *Y X*. *defined*(*X*::'*a*,*Y*) & *identity-map*(*Y*) −−> *product*(*X*::'*a*,*Y*,*X*)) &
(∀ *X Y Z W*. *product*(*X*::'*a*,*Y*,*Z*) & *product*(*X*::'*a*,*Y*,*W*) −−> *equal*(*Z*::'*a*,*W*))

abbreviation *CAT001-0-eq compos defined identity-map codomain domain product equal* ≡
(∀ *X Y Z W*. *equal*(*X*::'*a*,*Y*) & *product*(*X*::'*a*,*Z*,*W*) −−> *product*(*Y*::'*a*,*Z*,*W*)) &
(∀ *X Z Y W*. *equal*(*X*::'*a*,*Y*) & *product*(*Z*::'*a*,*X*,*W*) −−> *product*(*Z*::'*a*,*Y*,*W*)) &
(∀ *X Z W Y*. *equal*(*X*::'*a*,*Y*) & *product*(*Z*::'*a*,*W*,*X*) −−> *product*(*Z*::'*a*,*W*,*Y*)) &
(∀ *X Y*. *equal*(*X*::'*a*,*Y*) −−> *equal*(*domain*(*X*),*domain*(*Y*))) &

$(\forall\ X\ Y.\ equal(X::'a,Y)\ --{>}\ equal(codomain(X),codomain(Y)))\ \&$
$(\forall\ X\ Y.\ equal(X::'a,Y)\ \&\ identity\text{-}map(X)\ --{>}\ identity\text{-}map(Y))\ \&$
$(\forall\ X\ Y\ Z.\ equal(X::'a,Y)\ \&\ defined(X::'a,Z)\ --{>}\ defined(Y::'a,Z))\ \&$
$(\forall\ X\ Z\ Y.\ equal(X::'a,Y)\ \&\ defined(Z::'a,X)\ --{>}\ defined(Z::'a,Y))\ \&$
$(\forall\ X\ Z\ Y.\ equal(X::'a,Y)\ --{>}\ equal(compos(Z::'a,X),compos(Z::'a,Y)))\ \&$
$(\forall\ X\ Y\ Z.\ equal(X::'a,Y)\ --{>}\ equal(compos(X::'a,Z),compos(Y::'a,Z)))$

lemma *CAT005-1*:
EQU001-0-ax equal &
CAT001-0-ax equal codomain domain identity-map compos product defined &
CAT001-0-eq compos defined identity-map codomain domain product equal &
$(defined(a::'a,d))\ \&$
$(identity\text{-}map(d))\ \&$
$(\sim equal(domain(a),d))\ --{>}\ False$
$\langle proof\rangle$

lemma *CAT007-1*:
EQU001-0-ax equal &
CAT001-0-ax equal codomain domain identity-map compos product defined &
CAT001-0-eq compos defined identity-map codomain domain product equal &
$(equal(domain(a),codomain(b)))\ \&$
$(\sim defined(a::'a,b))\ --{>}\ False$
$\langle proof\rangle$

lemma *CAT018-1*:
EQU001-0-ax equal &
CAT001-0-ax equal codomain domain identity-map compos product defined &
CAT001-0-eq compos defined identity-map codomain domain product equal &
$(defined(a::'a,b))\ \&$
$(defined(b::'a,c))\ \&$
$(\sim defined(a::'a,compos(b::'a,c)))\ --{>}\ False$
$\langle proof\rangle$

lemma *COL001-2*:
EQU001-0-ax equal &
$(\forall\ X\ Y\ Z.\ equal(apply(apply(apply(s::'a,X),Y),Z),apply(apply(X::'a,Z),apply(Y::'a,Z))))$
&
$(\forall\ Y\ X.\ equal(apply(apply(k::'a,X),Y),X))\ \&$
$(\forall\ X\ Y\ Z.\ equal(apply(apply(apply(b::'a,X),Y),Z),apply(X::'a,apply(Y::'a,Z))))$
&
$(\forall\ X.\ equal(apply(i::'a,X),X))\ \&$
$(\forall\ A\ B\ C.\ equal(A::'a,B)\ --{>}\ equal(apply(A::'a,C),apply(B::'a,C)))\ \&$
$(\forall\ D\ F'\ E.\ equal(D::'a,E)\ --{>}\ equal(apply(F'::'a,D),apply(F'::'a,E)))\ \&$
$(\forall\ X.\ equal(apply(apply(apply(s::'a,apply(b::'a,X)),i),apply(apply(s::'a,apply(b::'a,X)),i)),apply(x::'a,apply($
&

$(\forall\, Y.\ {\sim}\mathit{equal}(Y{::}'a,\mathit{apply}(\mathit{combinator}{::}'a,Y)))\ {-\!-\!>}\ \mathit{False}$
$\langle \mathit{proof} \rangle$

lemma *COL023-1*:
$\mathit{EQU001\text{-}0\text{-}ax\ equal}\ \&$
$(\forall\, X\ Y\ Z.\ \mathit{equal}(\mathit{apply}(\mathit{apply}(\mathit{apply}(b{::}'a,X),Y),Z),\mathit{apply}(X{::}'a,\mathit{apply}(Y{::}'a,Z))))$
$\&$
$(\forall\, X\ Y\ Z.\ \mathit{equal}(\mathit{apply}(\mathit{apply}(\mathit{apply}(n{::}'a,X),Y),Z),\mathit{apply}(\mathit{apply}(\mathit{apply}(X{::}'a,Z),Y),Z)))$
$\&$
$(\forall\, A\ B\ C.\ \mathit{equal}(A{::}'a,B)\ {-\!-\!>}\ \mathit{equal}(\mathit{apply}(A{::}'a,C),\mathit{apply}(B{::}'a,C)))\ \&$
$(\forall\, D\ F'\ E.\ \mathit{equal}(D{::}'a,E)\ {-\!-\!>}\ \mathit{equal}(\mathit{apply}(F'{::}'a,D),\mathit{apply}(F'{::}'a,E)))\ \&$
$(\forall\, Y.\ {\sim}\mathit{equal}(Y{::}'a,\mathit{apply}(\mathit{combinator}{::}'a,Y)))\ {-\!-\!>}\ \mathit{False}$
$\langle \mathit{proof} \rangle$

lemma *COL032-1*:
$\mathit{EQU001\text{-}0\text{-}ax\ equal}\ \&$
$(\forall\, X.\ \mathit{equal}(\mathit{apply}(m{::}'a,X),\mathit{apply}(X{::}'a,X)))\ \&$
$(\forall\, Y\ X\ Z.\ \mathit{equal}(\mathit{apply}(\mathit{apply}(\mathit{apply}(q{::}'a,X),Y),Z),\mathit{apply}(Y{::}'a,\mathit{apply}(X{::}'a,Z))))$
$\&$
$(\forall\, A\ B\ C.\ \mathit{equal}(A{::}'a,B)\ {-\!-\!>}\ \mathit{equal}(\mathit{apply}(A{::}'a,C),\mathit{apply}(B{::}'a,C)))\ \&$
$(\forall\, D\ F'\ E.\ \mathit{equal}(D{::}'a,E)\ {-\!-\!>}\ \mathit{equal}(\mathit{apply}(F'{::}'a,D),\mathit{apply}(F'{::}'a,E)))\ \&$
$(\forall\, G\ H.\ \mathit{equal}(G{::}'a,H)\ {-\!-\!>}\ \mathit{equal}(f(G),f(H)))\ \&$
$(\forall\, Y.\ {\sim}\mathit{equal}(\mathit{apply}(Y{::}'a,f(Y)),\mathit{apply}(f(Y),\mathit{apply}(Y{::}'a,f(Y)))))\ {-\!-\!>}\ \mathit{False}$
$\langle \mathit{proof} \rangle$

lemma *COL052-2*:
$\mathit{EQU001\text{-}0\text{-}ax\ equal}\ \&$
$(\forall\, X\ Y\ W.\ \mathit{equal}(\mathit{response}(\mathit{compos}(X{::}'a,Y),W),\mathit{response}(X{::}'a,\mathit{response}(Y{::}'a,W))))$
$\&$
$(\forall\, X\ Y.\ \mathit{agreeable}(X)\ {-\!-\!>}\ \mathit{equal}(\mathit{response}(X{::}'a,\mathit{common\text{-}bird}(Y)),\mathit{response}(Y{::}'a,\mathit{common\text{-}bird}(Y))))$
$\&$
$(\forall\, Z\ X.\ \mathit{equal}(\mathit{response}(X{::}'a,Z),\mathit{response}(\mathit{compatible}(X),Z))\ {-\!-\!>}\ \mathit{agreeable}(X))$
$\&$
$(\forall\, A\ B.\ \mathit{equal}(A{::}'a,B)\ {-\!-\!>}\ \mathit{equal}(\mathit{common\text{-}bird}(A),\mathit{common\text{-}bird}(B)))\ \&$
$(\forall\, C\ D.\ \mathit{equal}(C{::}'a,D)\ {-\!-\!>}\ \mathit{equal}(\mathit{compatible}(C),\mathit{compatible}(D)))\ \&$
$(\forall\, Q\ R.\ \mathit{equal}(Q{::}'a,R)\ \&\ \mathit{agreeable}(Q)\ {-\!-\!>}\ \mathit{agreeable}(R))\ \&$
$(\forall\, A\ B\ C.\ \mathit{equal}(A{::}'a,B)\ {-\!-\!>}\ \mathit{equal}(\mathit{compos}(A{::}'a,C),\mathit{compos}(B{::}'a,C)))\ \&$
$(\forall\, D\ F'\ E.\ \mathit{equal}(D{::}'a,E)\ {-\!-\!>}\ \mathit{equal}(\mathit{compos}(F'{::}'a,D),\mathit{compos}(F'{::}'a,E)))\ \&$
$(\forall\, G\ H\ I'.\ \mathit{equal}(G{::}'a,H)\ {-\!-\!>}\ \mathit{equal}(\mathit{response}(G{::}'a,I'),\mathit{response}(H{::}'a,I')))\ \&$
$(\forall\, J\ L\ K'.\ \mathit{equal}(J{::}'a,K')\ {-\!-\!>}\ \mathit{equal}(\mathit{response}(L{::}'a,J),\mathit{response}(L{::}'a,K')))\ \&$
$(\mathit{agreeable}(c))\ \&$
$({\sim}\mathit{agreeable}(a))\ \&$
$(\mathit{equal}(c{::}'a,\mathit{compos}(a{::}'a,b)))\ {-\!-\!>}\ \mathit{False}$
$\langle \mathit{proof} \rangle$

lemma *COL075-2*:
EQU001-0-ax equal &
(∀ *Y X*. *equal*(*apply*(*apply*(*k*::'*a*,*X*),*Y*),*X*)) &
(∀ *X Y Z*. *equal*(*apply*(*apply*(*apply*(*abstraction*::'*a*,*X*),*Y*),*Z*),*apply*(*apply*(*X*::'*a*,*apply*(*k*::'*a*,*Z*)),*apply*(*Y*::'*a*,*Z*
&
(∀ *D E F'*. *equal*(*D*::'*a*,*E*) −−> *equal*(*apply*(*D*::'*a*,*F'*),*apply*(*E*::'*a*,*F'*))) &
(∀ *G I' H*. *equal*(*G*::'*a*,*H*) −−> *equal*(*apply*(*I'*::'*a*,*G*),*apply*(*I'*::'*a*,*H*))) &
(∀ *A B*. *equal*(*A*::'*a*,*B*) −−> *equal*(*b*(*A*),*b*(*B*))) &
(∀ *C D*. *equal*(*C*::'*a*,*D*) −−> *equal*(*c*(*C*),*c*(*D*))) &
(∀ *Y*. ~*equal*(*apply*(*apply*(*Y*::'*a*,*b*(*Y*)),*c*(*Y*)),*apply*(*b*(*Y*),*b*(*Y*)))) −−> *False*
⟨*proof*⟩

lemma *COM001-1*:
(∀ *Goal-state Start-state*. *follows*(*Goal-state*::'*a*,*Start-state*) −−> *succeeds*(*Goal-state*::'*a*,*Start-state*))
&
(∀ *Goal-state Intermediate-state Start-state*. *succeeds*(*Goal-state*::'*a*,*Intermediate-state*)
& *succeeds*(*Intermediate-state*::'*a*,*Start-state*) −−> *succeeds*(*Goal-state*::'*a*,*Start-state*))
&
(∀ *Start-state Label Goal-state*. *has*(*Start-state*::'*a*,*goto*(*Label*)) & *labels*(*Label*::'*a*,*Goal-state*)
−−> *succeeds*(*Goal-state*::'*a*,*Start-state*)) &
(∀ *Start-state Condition Goal-state*. *has*(*Start-state*::'*a*,*ifthen*(*Condition*::'*a*,*Goal-state*))
−−> *succeeds*(*Goal-state*::'*a*,*Start-state*)) &
(*labels*(*loop*::'*a*,*p3*)) &
(*has*(*p3*::'*a*,*ifthen*(*equal*(*register-j*::'*a*,*n*),*p4*))) &
(*has*(*p4*::'*a*,*goto*(*out*))) &
(*follows*(*p5*::'*a*,*p4*)) &
(*follows*(*p8*::'*a*,*p3*)) &
(*has*(*p8*::'*a*,*goto*(*loop*))) &
(~*succeeds*(*p3*::'*a*,*p3*)) −−> *False*
⟨*proof*⟩

lemma *COM002-1*:
(∀ *Goal-state Start-state*. *follows*(*Goal-state*::'*a*,*Start-state*) −−> *succeeds*(*Goal-state*::'*a*,*Start-state*))
&
(∀ *Goal-state Intermediate-state Start-state*. *succeeds*(*Goal-state*::'*a*,*Intermediate-state*)
& *succeeds*(*Intermediate-state*::'*a*,*Start-state*) −−> *succeeds*(*Goal-state*::'*a*,*Start-state*))
&
(∀ *Start-state Label Goal-state*. *has*(*Start-state*::'*a*,*goto*(*Label*)) & *labels*(*Label*::'*a*,*Goal-state*)
−−> *succeeds*(*Goal-state*::'*a*,*Start-state*)) &
(∀ *Start-state Condition Goal-state*. *has*(*Start-state*::'*a*,*ifthen*(*Condition*::'*a*,*Goal-state*))
−−> *succeeds*(*Goal-state*::'*a*,*Start-state*)) &
(*has*(*p1*::'*a*,*assign*(*register-j*::'*a*,*num0*))) &
(*follows*(*p2*::'*a*,*p1*)) &
(*has*(*p2*::'*a*,*assign*(*register-k*::'*a*,*num1*))) &
(*labels*(*loop*::'*a*,*p3*)) &
(*follows*(*p3*::'*a*,*p2*)) &
(*has*(*p3*::'*a*,*ifthen*(*equal*(*register-j*::'*a*,*n*),*p4*))) &

(*has*(*p4*::'*a*,*goto*(*out*))) &
(*follows*(*p5*::'*a*,*p4*)) &
(*follows*(*p6*::'*a*,*p3*)) &
(*has*(*p6*::'*a*,*assign*(*register-k*::'*a*,*mtimes*(*num2*::'*a*,*register-k*)))) &
(*follows*(*p7*::'*a*,*p6*)) &
(*has*(*p7*::'*a*,*assign*(*register-j*::'*a*,*mplus*(*register-j*::'*a*,*num1*)))) &
(*follows*(*p8*::'*a*,*p7*)) &
(*has*(*p8*::'*a*,*goto*(*loop*))) &
(~*succeeds*(*p3*::'*a*,*p3*)) −−> *False*
⟨*proof*⟩

lemma *COM002-2*:
(∀ *Goal-state Start-state*. ~(*fails*(*Goal-state*::'*a*,*Start-state*) & *follows*(*Goal-state*::'*a*,*Start-state*)))
&
(∀ *Goal-state Intermediate-state Start-state*. *fails*(*Goal-state*::'*a*,*Start-state*) −−>
fails(*Goal-state*::'*a*,*Intermediate-state*) | *fails*(*Intermediate-state*::'*a*,*Start-state*)) &
(∀ *Start-state Label Goal-state*. ~(*fails*(*Goal-state*::'*a*,*Start-state*) & *has*(*Start-state*::'*a*,*goto*(*Label*))
& *labels*(*Label*::'*a*,*Goal-state*))) &
(∀ *Start-state Condition Goal-state*. ~(*fails*(*Goal-state*::'*a*,*Start-state*) & *has*(*Start-state*::'*a*,*ifthen*(*Condition*::
&
(*has*(*p1*::'*a*,*assign*(*register-j*::'*a*,*num0*))) &
(*follows*(*p2*::'*a*,*p1*)) &
(*has*(*p2*::'*a*,*assign*(*register-k*::'*a*,*num1*))) &
(*labels*(*loop*::'*a*,*p3*)) &
(*follows*(*p3*::'*a*,*p2*)) &
(*has*(*p3*::'*a*,*ifthen*(*equal*(*register-j*::'*a*,*n*),*p4*))) &
(*has*(*p4*::'*a*,*goto*(*out*))) &
(*follows*(*p5*::'*a*,*p4*)) &
(*follows*(*p6*::'*a*,*p3*)) &
(*has*(*p6*::'*a*,*assign*(*register-k*::'*a*,*mtimes*(*num2*::'*a*,*register-k*)))) &
(*follows*(*p7*::'*a*,*p6*)) &
(*has*(*p7*::'*a*,*assign*(*register-j*::'*a*,*mplus*(*register-j*::'*a*,*num1*)))) &
(*follows*(*p8*::'*a*,*p7*)) &
(*has*(*p8*::'*a*,*goto*(*loop*))) &
(*fails*(*p3*::'*a*,*p3*)) −−> *False*
⟨*proof*⟩

lemma *COM003-2*:
(∀ *X Y Z*. *program-decides*(*X*) & *program*(*Y*) −−> *decides*(*X*::'*a*,*Y*,*Z*)) &
(∀ *X*. *program-decides*(*X*) | *program*(*f2*(*X*))) &
(∀ *X*. *decides*(*X*::'*a*,*f2*(*X*),*f1*(*X*)) −−> *program-decides*(*X*)) &
(∀ *X*. *program-program-decides*(*X*) −−> *program*(*X*)) &
(∀ *X*. *program-program-decides*(*X*) −−> *program-decides*(*X*)) &
(∀ *X*. *program*(*X*) & *program-decides*(*X*) −−> *program-program-decides*(*X*)) &
(∀ *X*. *algorithm-program-decides*(*X*) −−> *algorithm*(*X*)) &
(∀ *X*. *algorithm-program-decides*(*X*) −−> *program-decides*(*X*)) &
(∀ *X*. *algorithm*(*X*) & *program-decides*(*X*) −−> *algorithm-program-decides*(*X*))

$$
\begin{array}{l}
\& \\
\quad (\forall\, Y\ X.\ \mathit{program\text{-}halts2}(X::'a, Y) \longrightarrow \mathit{program}(X)) \ \& \\
\quad (\forall\, X\ Y.\ \mathit{program\text{-}halts2}(X::'a, Y) \longrightarrow \mathit{halts2}(X::'a, Y)) \ \& \\
\quad (\forall\, X\ Y.\ \mathit{program}(X) \ \&\ \mathit{halts2}(X::'a, Y) \longrightarrow \mathit{program\text{-}halts2}(X::'a, Y)) \ \& \\
\quad (\forall\, W\ X\ Y\ Z.\ \mathit{halts3\text{-}outputs}(X::'a, Y, Z, W) \longrightarrow \mathit{halts3}(X::'a, Y, Z)) \ \& \\
\quad (\forall\, Y\ Z\ X\ W.\ \mathit{halts3\text{-}outputs}(X::'a, Y, Z, W) \longrightarrow \mathit{outputs}(X::'a, W)) \ \& \\
\quad (\forall\, Y\ Z\ X\ W.\ \mathit{halts3}(X::'a, Y, Z) \ \&\ \mathit{outputs}(X::'a, W) \longrightarrow \mathit{halts3\text{-}outputs}(X::'a, Y, Z, W)) \\
\& \\
\quad (\forall\, Y\ X.\ \mathit{program\text{-}not\text{-}halts2}(X::'a, Y) \longrightarrow \mathit{program}(X)) \ \& \\
\quad (\forall\, X\ Y.\ \sim(\mathit{program\text{-}not\text{-}halts2}(X::'a, Y) \ \&\ \mathit{halts2}(X::'a, Y))) \ \& \\
\quad (\forall\, X\ Y.\ \mathit{program}(X) \longrightarrow \mathit{program\text{-}not\text{-}halts2}(X::'a, Y) \mid \mathit{halts2}(X::'a, Y)) \ \& \\
\quad (\forall\, W\ X\ Y.\ \mathit{halts2\text{-}outputs}(X::'a, Y, W) \longrightarrow \mathit{halts2}(X::'a, Y)) \ \& \\
\quad (\forall\, Y\ X\ W.\ \mathit{halts2\text{-}outputs}(X::'a, Y, W) \longrightarrow \mathit{outputs}(X::'a, W)) \ \& \\
\quad (\forall\, Y\ X\ W.\ \mathit{halts2}(X::'a, Y) \ \&\ \mathit{outputs}(X::'a, W) \longrightarrow \mathit{halts2\text{-}outputs}(X::'a, Y, W)) \\
\& \\
\quad (\forall\, X\ W\ Y\ Z.\ \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a, Y, Z, W) \longrightarrow \mathit{program\text{-}halts2}(Y::'a, Z)) \\
\& \\
\quad (\forall\, X\ Y\ Z\ W.\ \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a, Y, Z, W) \longrightarrow \mathit{halts3\text{-}outputs}(X::'a, Y, Z, W)) \\
\& \\
\quad (\forall\, X\ \ Y\ \ Z\ \ W.\ \ \mathit{program\text{-}halts2}(Y::'a, Z) \ \&\ \ \mathit{halts3\text{-}outputs}(X::'a, Y, Z, W) \ \longrightarrow \\
\mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a, Y, Z, W)) \ \& \\
\quad (\forall\, X\ W\ Y\ Z.\ \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a, Y, Z, W) \longrightarrow \mathit{program\text{-}not\text{-}halts2}(Y::'a, Z)) \\
\& \\
\quad (\forall\, X\ Y\ Z\ W.\ \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a, Y, Z, W) \longrightarrow \mathit{halts3\text{-}outputs}(X::'a, Y, Z, W)) \\
\& \\
\quad (\forall\, X\ Y\ Z\ W.\ \mathit{program\text{-}not\text{-}halts2}(Y::'a, Z) \ \&\ \mathit{halts3\text{-}outputs}(X::'a, Y, Z, W) \longrightarrow \\
\mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a, Y, Z, W)) \ \& \\
\quad (\forall\, X\ W\ Y.\ \mathit{program\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a, Y, W) \longrightarrow \mathit{program\text{-}halts2}(Y::'a, Y)) \\
\& \\
\quad (\forall\, X\ Y\ W.\ \mathit{program\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a, Y, W) \longrightarrow \mathit{halts2\text{-}outputs}(X::'a, Y, W)) \\
\& \\
\quad (\forall\, X\ Y\ W.\ \mathit{program\text{-}halts2}(Y::'a, Y) \ \&\ \mathit{halts2\text{-}outputs}(X::'a, Y, W) \longrightarrow \mathit{program\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a \\
\& \\
\quad (\forall\, X\ W\ Y.\ \mathit{program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a, Y, W) \longrightarrow \mathit{program\text{-}not\text{-}halts2}(Y::'a, Y)) \\
\& \\
\quad (\forall\, X\ Y\ W.\ \mathit{program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a, Y, W) \longrightarrow \mathit{halts2\text{-}outputs}(X::'a, Y, W)) \\
\& \\
\quad (\forall\, X\ \ Y\ \ W.\ \ \mathit{program\text{-}not\text{-}halts2}(Y::'a, Y) \ \&\ \ \mathit{halts2\text{-}outputs}(X::'a, Y, W) \ \longrightarrow \\
\mathit{program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a, Y, W)) \ \& \\
\quad (\forall\, X.\ \mathit{algorithm\text{-}program\text{-}decides}(X) \longrightarrow \mathit{program\text{-}program\text{-}decides}(c1)) \ \& \\
\quad (\forall\, W\ Y\ Z.\ \mathit{program\text{-}program\text{-}decides}(W) \longrightarrow \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a, Y, Z, \mathit{good})) \\
\& \\
\quad (\forall\, W\ Y\ Z.\ \mathit{program\text{-}program\text{-}decides}(W) \longrightarrow \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a, Y, Z, \mathit{bad})) \\
\& \\
\quad (\forall\, W.\ \mathit{program}(W) \ \&\ \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a, \mathit{f3}(W), \mathit{f3}(W), \mathit{good}) \\
\&\ \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a, \mathit{f3}(W), \mathit{f3}(W), \mathit{bad}) \longrightarrow \mathit{program}(c2)) \\
\& \\
\quad (\forall\, W\ Y.\ \mathit{program}(W) \ \&\ \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a, \mathit{f3}(W), \mathit{f3}(W), \mathit{good}) \\
\&\ \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a, \mathit{f3}(W), \mathit{f3}(W), \mathit{bad}) \longrightarrow \mathit{program\text{-}halts2\text{-}halts2\text{-}outputs}(c2::'a, Y, g
\end{array}
$$

$$
\begin{array}{l}
\& \\
\quad (\forall\, W\ Y.\ program(W)\ \&\ program\text{-}halts2\text{-}halts3\text{-}outputs(W::'a,f3(W),f3(W),good) \\
\& \ program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs(W::'a,f3(W),f3(W),bad) \longrightarrow program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs(c2::'a, \\
\& \\
\quad (\forall\, V.\ program(V)\ \&\ program\text{-}halts2\text{-}halts2\text{-}outputs(V::'a,f4(V),good)\ \&\ program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs(V \\
\longrightarrow program(c3))\ \& \\
\quad (\forall\, V\ Y.\ program(V)\ \&\ program\text{-}halts2\text{-}halts2\text{-}outputs(V::'a,f4(V),good)\ \&\ program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs \\
\& \ program\text{-}halts2(Y::'a,Y) \longrightarrow halts2(c3::'a,Y))\ \& \\
\quad (\forall\, V\ Y.\ program(V)\ \&\ program\text{-}halts2\text{-}halts2\text{-}outputs(V::'a,f4(V),good)\ \&\ program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs \\
\longrightarrow program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs(c3::'a,Y,bad))\ \& \\
\quad (algorithm\text{-}program\text{-}decides(c4)) \longrightarrow False \\
\quad \langle proof \rangle
\end{array}
$$

lemma *COM004-1*:

$$
\begin{array}{l}
\quad EQU001\text{-}0\text{-}ax\ equal\ \& \\
\quad (\forall\, C\ D\ P\ Q\ X\ Y.\ failure\text{-}node(X::'a,or(C::'a,P))\ \&\ failure\text{-}node(Y::'a,or(D::'a,Q)) \\
\& \ contradictory(P::'a,Q)\ \&\ siblings(X::'a,Y) \longrightarrow failure\text{-}node(parent\text{-}of(X::'a,Y),or(C::'a,D))) \\
\& \\
\quad (\forall\, X.\ contradictory(negate(X),X))\ \& \\
\quad (\forall\, X.\ contradictory(X::'a,negate(X)))\ \& \\
\quad (\forall\, X.\ siblings(left\text{-}child\text{-}of(X),right\text{-}child\text{-}of(X)))\ \& \\
\quad (\forall\, D\ E.\ equal(D::'a,E) \longrightarrow equal(left\text{-}child\text{-}of(D),left\text{-}child\text{-}of(E)))\ \& \\
\quad (\forall\, F'\ G.\ equal(F'::'a,G) \longrightarrow equal(negate(F'),negate(G)))\ \& \\
\quad (\forall\, H\ I'\ J.\ equal(H::'a,I') \longrightarrow equal(or(H::'a,J),or(I'::'a,J)))\ \& \\
\quad (\forall\, K'\ M\ L.\ equal(K'::'a,L) \longrightarrow equal(or(M::'a,K'),or(M::'a,L)))\ \& \\
\quad (\forall\, N\ O'\ P.\ equal(N::'a,O') \longrightarrow equal(parent\text{-}of(N::'a,P),parent\text{-}of(O'::'a,P))) \\
\& \\
\quad (\forall\, Q\ S'\ R.\ equal(Q::'a,R) \longrightarrow equal(parent\text{-}of(S'::'a,Q),parent\text{-}of(S'::'a,R))) \\
\& \\
\quad (\forall\, T'\ U.\ equal(T'::'a,U) \longrightarrow equal(right\text{-}child\text{-}of(T'),right\text{-}child\text{-}of(U)))\ \& \\
\quad (\forall\, V\ W\ X.\ equal(V::'a,W)\ \&\ contradictory(V::'a,X) \longrightarrow contradictory(W::'a,X)) \\
\& \\
\quad (\forall\, Y\ A1\ Z.\ equal(Y::'a,Z)\ \&\ contradictory(A1::'a,Y) \longrightarrow contradictory(A1::'a,Z)) \\
\& \\
\quad (\forall\, B1\ C1\ D1.\ equal(B1::'a,C1)\ \&\ failure\text{-}node(B1::'a,D1) \longrightarrow failure\text{-}node(C1::'a,D1)) \\
\& \\
\quad (\forall\, E1\ G1\ F1.\ equal(E1::'a,F1)\ \&\ failure\text{-}node(G1::'a,E1) \longrightarrow failure\text{-}node(G1::'a,F1)) \\
\& \\
\quad (\forall\, H1\ I1\ J1.\ equal(H1::'a,I1)\ \&\ siblings(H1::'a,J1) \longrightarrow siblings(I1::'a,J1))\ \& \\
\quad (\forall\, K1\ M1\ L1.\ equal(K1::'a,L1)\ \&\ siblings(M1::'a,K1) \longrightarrow siblings(M1::'a,L1)) \\
\& \\
\quad (failure\text{-}node(n\text{-}left::'a,or(EMPTY::'a,atom)))\ \& \\
\quad (failure\text{-}node(n\text{-}right::'a,or(EMPTY::'a,negate(atom))))\ \& \\
\quad (equal(n\text{-}left::'a,left\text{-}child\text{-}of(n)))\ \& \\
\quad (equal(n\text{-}right::'a,right\text{-}child\text{-}of(n)))\ \& \\
\quad (\forall\, Z.\ \sim failure\text{-}node(Z::'a,or(EMPTY::'a,EMPTY))) \longrightarrow False \\
\quad \langle proof \rangle
\end{array}
$$

abbreviation *GEO001-0-ax continuous lower-dimension-point-3 lower-dimension-point-2 lower-dimension-point-1 extension euclid2 euclid1 outer-pasch equidistant equal between* ≡
(∀ *X Y*. *between*(*X*::'*a*,*Y*,*X*) −−> *equal*(*X*::'*a*,*Y*)) &
(∀ *V X Y Z*. *between*(*X*::'*a*,*Y*,*V*) & *between*(*Y*::'*a*,*Z*,*V*) −−> *between*(*X*::'*a*,*Y*,*Z*)) &
(∀ *Y X V Z*. *between*(*X*::'*a*,*Y*,*Z*) & *between*(*X*::'*a*,*Y*,*V*) −−> *equal*(*X*::'*a*,*Y*) | *between*(*X*::'*a*,*Z*,*V*) | *between*(*X*::'*a*,*V*,*Z*)) &
(∀ *Y X*. *equidistant*(*X*::'*a*,*Y*,*Y*,*X*)) &
(∀ *Z X Y*. *equidistant*(*X*::'*a*,*Y*,*Z*,*Z*) −−> *equal*(*X*::'*a*,*Y*)) &
(∀ *X Y Z V V2 W*. *equidistant*(*X*::'*a*,*Y*,*Z*,*V*) & *equidistant*(*X*::'*a*,*Y*,*V2*,*W*) −−> *equidistant*(*Z*::'*a*,*V*,*V2*,*W*)) &
(∀ *W X Z V Y*. *between*(*X*::'*a*,*W*,*V*) & *between*(*Y*::'*a*,*V*,*Z*) −−> *between*(*X*::'*a*,*outer-pasch*(*W*::'*a*,*X*,*Y*,*Z*,*W*
&
(∀ *W X Y Z V*. *between*(*X*::'*a*,*W*,*V*) & *between*(*Y*::'*a*,*V*,*Z*) −−> *between*(*Z*::'*a*,*W*,*outer-pasch*(*W*::'*a*,*X*,*Y*,*Z*
&
(∀ *W X Y Z V*. *between*(*X*::'*a*,*V*,*W*) & *between*(*Y*::'*a*,*V*,*Z*) −−> *equal*(*X*::'*a*,*V*) | *between*(*X*::'*a*,*Z*,*euclid1*(*W*::'*a*,*X*,*Y*,*Z*,*V*))) &
(∀ *W X Y Z V*. *between*(*X*::'*a*,*V*,*W*) & *between*(*Y*::'*a*,*V*,*Z*) −−> *equal*(*X*::'*a*,*V*) | *between*(*X*::'*a*,*Y*,*euclid2*(*W*::'*a*,*X*,*Y*,*Z*,*V*))) &
(∀ *W X Y Z V*. *between*(*X*::'*a*,*V*,*W*) & *between*(*Y*::'*a*,*V*,*Z*) −−> *equal*(*X*::'*a*,*V*) | *between*(*euclid1*(*W*::'*a*,*X*,*Y*,*Z*,*V*),*W*,*euclid2*(*W*::'*a*,*X*,*Y*,*Z*,*V*))) &
(∀ *X1 Y1 X Y Z V Z1 V1*. *equidistant*(*X*::'*a*,*Y*,*X1*,*Y1*) & *equidistant*(*Y*::'*a*,*Z*,*Y1*,*Z1*) & *equidistant*(*X*::'*a*,*V*,*X1*,*V1*) & *equidistant*(*Y*::'*a*,*V*,*Y1*,*V1*) & *between*(*X*::'*a*,*Y*,*Z*) & *between*(*X1*::'*a*,*Y1*,*Z1*) −−> *equal*(*X*::'*a*,*Y*) | *equidistant*(*Z*::'*a*,*V*,*Z1*,*V1*)) &
(∀ *X Y W V*. *between*(*X*::'*a*,*Y*,*extension*(*X*::'*a*,*Y*,*W*,*V*))) &
(∀ *X Y W V*. *equidistant*(*Y*::'*a*,*extension*(*X*::'*a*,*Y*,*W*,*V*),*W*,*V*)) &
(~*between*(*lower-dimension-point-1*::'*a*,*lower-dimension-point-2*,*lower-dimension-point-3*)) &
(~*between*(*lower-dimension-point-2*::'*a*,*lower-dimension-point-3*,*lower-dimension-point-1*)) &
(~*between*(*lower-dimension-point-3*::'*a*,*lower-dimension-point-1*,*lower-dimension-point-2*)) &
(∀ *Z X Y W V*. *equidistant*(*X*::'*a*,*W*,*X*,*V*) & *equidistant*(*Y*::'*a*,*W*,*Y*,*V*) & *equidistant*(*Z*::'*a*,*W*,*Z*,*V*) −−> *between*(*X*::'*a*,*Y*,*Z*) | *between*(*Y*::'*a*,*Z*,*X*) | *between*(*Z*::'*a*,*X*,*Y*) | *equal*(*W*::'*a*,*V*)) &
(∀ *X Y Z X1 Z1 V*. *equidistant*(*V*::'*a*,*X*,*V*,*X1*) & *equidistant*(*V*::'*a*,*Z*,*V*,*Z1*) & *between*(*V*::'*a*,*X*,*Z*) & *between*(*X*::'*a*,*Y*,*Z*) −−> *equidistant*(*V*::'*a*,*Y*,*Z*,*continuous*(*X*::'*a*,*Y*,*Z*,*X1*,*Z1*,*V*))) &
(∀ *X Y Z X1 V Z1*. *equidistant*(*V*::'*a*,*X*,*V*,*X1*) & *equidistant*(*V*::'*a*,*Z*,*V*,*Z1*) & *between*(*V*::'*a*,*X*,*Z*) & *between*(*X*::'*a*,*Y*,*Z*) −−> *between*(*X1*::'*a*,*continuous*(*X*::'*a*,*Y*,*Z*,*X1*,*Z1*,*V*),*Z1*))

abbreviation *GEO001-0-eq continuous extension euclid2 euclid1 outer-pasch equidistant between equal* ≡
(∀ *X Y W Z*. *equal*(*X*::'*a*,*Y*) & *between*(*X*::'*a*,*W*,*Z*) −−> *between*(*Y*::'*a*,*W*,*Z*)) &
(∀ *X W Y Z*. *equal*(*X*::'*a*,*Y*) & *between*(*W*::'*a*,*X*,*Z*) −−> *between*(*W*::'*a*,*Y*,*Z*)) &
(∀ *X W Z Y*. *equal*(*X*::'*a*,*Y*) & *between*(*W*::'*a*,*Z*,*X*) −−> *between*(*W*::'*a*,*Z*,*Y*))

$$
\begin{array}{l}
\& \\
\quad (\forall\, X\ Y\ V\ W\ Z.\ equal(X::'a,Y)\ \&\ equidistant(X::'a,V,W,Z)\ -->\ equidis- \\
tant(Y::'a,V,W,Z))\ \& \\
\quad (\forall\, X\ V\ Y\ W\ Z.\ equal(X::'a,Y)\ \&\ equidistant(V::'a,X,W,Z)\ -->\ equidis- \\
tant(V::'a,Y,W,Z))\ \& \\
\quad (\forall\, X\ V\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ equidistant(V::'a,W,X,Z)\ -->\ equidis- \\
tant(V::'a,W,Y,Z))\ \& \\
\quad (\forall\, X\ V\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ equidistant(V::'a,W,Z,X)\ -->\ equidis- \\
tant(V::'a,W,Z,Y))\ \& \\
\ (\forall\, X\ Y\ V1\ V2\ V3\ V4.\ equal(X::'a,Y)\ -->\ equal(outer\text{-}pasch(X::'a,V1,V2,V3,V4),outer\text{-}pasch(Y::'a,V1,V \\
\& \\
\ (\forall\, X\ V1\ Y\ V2\ V3\ V4.\ equal(X::'a,Y)\ -->\ equal(outer\text{-}pasch(V1::'a,X,V2,V3,V4),outer\text{-}pasch(V1::'a,Y,V \\
\& \\
\ (\forall\, X\ V1\ V2\ Y\ V3\ V4.\ equal(X::'a,Y)\ -->\ equal(outer\text{-}pasch(V1::'a,V2,X,V3,V4),outer\text{-}pasch(V1::'a,V2, \\
\& \\
\ (\forall\, X\ V1\ V2\ V3\ Y\ V4.\ equal(X::'a,Y)\ -->\ equal(outer\text{-}pasch(V1::'a,V2,V3,X,V4),outer\text{-}pasch(V1::'a,V2, \\
\& \\
\ (\forall\, X\ V1\ V2\ V3\ V4\ Y.\ equal(X::'a,Y)\ -->\ equal(outer\text{-}pasch(V1::'a,V2,V3,V4,X),outer\text{-}pasch(V1::'a,V2, \\
\& \\
\ (\forall\, A\ B\ C\ D\ E\ F'.\ equal(A::'a,B)\ -->\ equal(euclid1(A::'a,C,D,E,F'),euclid1(B::'a,C,D,E,F'))) \\
\& \\
\ (\forall\, G\ I'\ H\ J\ K'\ L.\ equal(G::'a,H)\ -->\ equal(euclid1(I'::'a,G,J,K',L),euclid1(I'::'a,H,J,K',L))) \\
\& \\
\ (\forall\, M\ O'\ P\ N\ Q\ R.\ equal(M::'a,N)\ -->\ equal(euclid1(O'::'a,P,M,Q,R),euclid1(O'::'a,P,N,Q,R))) \\
\& \\
\ (\forall\, S'\ U\ V\ W\ T'\ X.\ equal(S'::'a,T')\ -->\ equal(euclid1(U::'a,V,W,S',X),euclid1(U::'a,V,W,T',X))) \\
\& \\
\ (\forall\, Y\ A1\ B1\ C1\ D1\ Z.\ equal(Y::'a,Z)\ -->\ equal(euclid1(A1::'a,B1,C1,D1,Y),euclid1(A1::'a,B1,C1,D1,Z) \\
\& \\
\ (\forall\, E1\ F1\ G1\ H1\ I1\ J1.\ equal(E1::'a,F1)\ -->\ equal(euclid2(E1::'a,G1,H1,I1,J1),euclid2(F1::'a,G1,H1,I1, \\
\& \\
\ (\forall\, K1\ M1\ L1\ N1\ O1\ P1.\ equal(K1::'a,L1)\ -->\ equal(euclid2(M1::'a,K1,N1,O1,P1),euclid2(M1::'a,L1,N1, \\
\& \\
\ (\forall\, Q1\ S1\ T1\ R1\ U1\ V1.\ equal(Q1::'a,R1)\ -->\ equal(euclid2(S1::'a,T1,Q1,U1,V1),euclid2(S1::'a,T1,R1,U \\
\& \\
\ (\forall\, W1\ Y1\ Z1\ A2\ X1\ B2.\ equal(W1::'a,X1)\ -->\ equal(euclid2(Y1::'a,Z1,A2,W1,B2),euclid2(Y1::'a,Z1,A2 \\
\& \\
\ (\forall\, C2\ E2\ F2\ G2\ H2\ D2.\ equal(C2::'a,D2)\ -->\ equal(euclid2(E2::'a,F2,G2,H2,C2),euclid2(E2::'a,F2,G2,H \\
\& \\
\ (\forall\, X\ Y\ V1\ V2\ V3.\ equal(X::'a,Y)\ -->\ equal(extension(X::'a,V1,V2,V3),extension(Y::'a,V1,V2,V3))) \\
\& \\
\ (\forall\, X\ V1\ Y\ V2\ V3.\ equal(X::'a,Y)\ -->\ equal(extension(V1::'a,X,V2,V3),extension(V1::'a,Y,V2,V3))) \\
\& \\
\ (\forall\, X\ V1\ V2\ Y\ V3.\ equal(X::'a,Y)\ -->\ equal(extension(V1::'a,V2,X,V3),extension(V1::'a,V2,Y,V3))) \\
\& \\
\ (\forall\, X\ V1\ V2\ V3\ Y.\ equal(X::'a,Y)\ -->\ equal(extension(V1::'a,V2,V3,X),extension(V1::'a,V2,V3,Y))) \\
\& \\
\ (\forall\, X\ Y\ V1\ V2\ V3\ V4\ V5.\ equal(X::'a,Y)\ -->\ equal(continuous(X::'a,V1,V2,V3,V4,V5),continuous(Y::'a, \\
\&
\end{array}
$$

(∀ X V1 Y V2 V3 V4 V5. equal(X::'a,Y) −−> equal(continuous(V1::'a,X,V2,V3,V4,V5),continuous(V1::'a
&
(∀ X V1 V2 Y V3 V4 V5. equal(X::'a,Y) −−> equal(continuous(V1::'a,V2,X,V3,V4,V5),continuous(V1::'a
&
(∀ X V1 V2 V3 Y V4 V5. equal(X::'a,Y) −−> equal(continuous(V1::'a,V2,V3,X,V4,V5),continuous(V1::'a
&
(∀ X V1 V2 V3 V4 Y V5. equal(X::'a,Y) −−> equal(continuous(V1::'a,V2,V3,V4,X,V5),continuous(V1::'a
&
(∀ X V1 V2 V3 V4 V5 Y. equal(X::'a,Y) −−> equal(continuous(V1::'a,V2,V3,V4,V5,X),continuous(V1::'a

lemma *GEO003-1*:
EQU001-0-ax equal &
GEO001-0-ax continuous lower-dimension-point-3 lower-dimension-point-2
lower-dimension-point-1 extension euclid2 euclid1 outer-pasch equidistant equal
between &
GEO001-0-eq continuous extension euclid2 euclid1 outer-pasch equidistant between
equal &
(~between(a::'a,b,b)) −−> *False*
⟨*proof*⟩

abbreviation *GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3*
lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension
between equal equidistant ≡
(∀ Y X. equidistant(X::'a,Y,Y,X)) &
(∀ X Y Z V V2 W. equidistant(X::'a,Y,Z,V) & equidistant(X::'a,Y,V2,W)
−−> equidistant(Z::'a,V,V2,W)) &
(∀ Z X Y. equidistant(X::'a,Y,Z,Z) −−> equal(X::'a,Y)) &
(∀ X Y W V. between(X::'a,Y,extension(X::'a,Y,W,V))) &
(∀ X Y W V. equidistant(Y::'a,extension(X::'a,Y,W,V),W,V)) &
(∀ X1 Y1 X Y Z V Z1 V1. equidistant(X::'a,Y,X1,Y1) & equidistant(Y::'a,Z,Y1,Z1)
& equidistant(X::'a,V,X1,V1) & equidistant(Y::'a,V,Y1,V1) & between(X::'a,Y,Z)
& between(X1::'a,Y1,Z1) −−> equal(X::'a,Y) | equidistant(Z::'a,V,Z1,V1)) &
(∀ X Y. between(X::'a,Y,X) −−> equal(X::'a,Y)) &
(∀ U V W X Y. between(U::'a,V,W) & between(Y::'a,X,W) −−> between(V::'a,inner-pasch(U::'a,V,W,X
&
(∀ V W X Y U. between(U::'a,V,W) & between(Y::'a,X,W) −−> between(X::'a,inner-pasch(U::'a,V,W,X,
&
(~between(lower-dimension-point-1::'a,lower-dimension-point-2,lower-dimension-point-3))
&
(~between(lower-dimension-point-2::'a,lower-dimension-point-3,lower-dimension-point-1))
&
(~between(lower-dimension-point-3::'a,lower-dimension-point-1,lower-dimension-point-2))
&
(∀ Z X Y W V. equidistant(X::'a,W,X,V) & equidistant(Y::'a,W,Y,V) & equidis-
tant(Z::'a,W,Z,V) −−> between(X::'a,Y,Z) | between(Y::'a,Z,X) | between(Z::'a,X,Y)
| equal(W::'a,V)) &
(∀ U V W X Y. between(U::'a,W,Y) & between(V::'a,W,X) −−> equal(U::'a,W)

$$
\begin{array}{l}
\mid between(U::'a,V,euclid1(U::'a,V,W,X,Y))) \;\&\\
\;(\forall\, U\ V\ W\ X\ Y.\ between(U::'a,W,Y)\ \&\ between(V::'a,W,X) \longrightarrow equal(U::'a,W)\\
\mid between(U::'a,X,euclid2(U::'a,V,W,X,Y))) \;\&\\
\;(\forall\, U\ V\ W\ X\ Y.\ between(U::'a,W,Y)\ \&\ between(V::'a,W,X) \longrightarrow equal(U::'a,W)\\
\mid between(euclid1(U::'a,V,W,X,Y),Y,euclid2(U::'a,V,W,X,Y))) \;\&\\
\;(\forall\, U\ V\ V1\ W\ X\ X1.\ equidistant(U::'a,V,U,V1)\ \&\ equidistant(U::'a,X,U,X1)\ \&\\
between(U::'a,V,X)\ \&\ between(V::'a,W,X) \longrightarrow between(V1::'a,continuous(U::'a,V,V1,W,X,X1),X1))\\
\&\\
\;(\forall\, U\ V\ V1\ W\ X\ X1.\ equidistant(U::'a,V,U,V1)\ \&\ equidistant(U::'a,X,U,X1)\ \&\\
between(U::'a,V,X)\ \&\ between(V::'a,W,X) \longrightarrow equidistant(U::'a,W,U,continuous(U::'a,V,V1,W,X,X1))\\
\&\\
\;(\forall\, X\ Y\ W\ Z.\ equal(X::'a,Y)\ \&\ between(X::'a,W,Z) \longrightarrow between(Y::'a,W,Z))\\
\&\\
\;(\forall\, X\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ between(W::'a,X,Z) \longrightarrow between(W::'a,Y,Z))\\
\&\\
\;(\forall\, X\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ between(W::'a,Z,X) \longrightarrow between(W::'a,Z,Y))\\
\&\\
\;(\forall\, X\ Y\ V\ W\ Z.\ equal(X::'a,Y)\ \&\ equidistant(X::'a,V,W,Z) \longrightarrow equidis-\\
tant(Y::'a,V,W,Z)) \;\&\\
\;(\forall\, X\ V\ Y\ W\ Z.\ equal(X::'a,Y)\ \&\ equidistant(V::'a,X,W,Z) \longrightarrow equidis-\\
tant(V::'a,Y,W,Z)) \;\&\\
\;(\forall\, X\ V\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ equidistant(V::'a,W,X,Z) \longrightarrow equidis-\\
tant(V::'a,W,Y,Z)) \;\&\\
\;(\forall\, X\ V\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ equidistant(V::'a,W,Z,X) \longrightarrow equidis-\\
tant(V::'a,W,Z,Y)) \;\&\\
\;(\forall\, X\ Y\ V1\ V2\ V3\ V4.\ equal(X::'a,Y) \longrightarrow equal(inner\text{-}pasch(X::'a,V1,V2,V3,V4),inner\text{-}pasch(Y::'a,V1,V\\
\&\\
\;(\forall\, X\ V1\ Y\ V2\ V3\ V4.\ equal(X::'a,Y) \longrightarrow equal(inner\text{-}pasch(V1::'a,X,V2,V3,V4),inner\text{-}pasch(V1::'a,Y,V\\
\&\\
\;(\forall\, X\ V1\ V2\ Y\ V3\ V4.\ equal(X::'a,Y) \longrightarrow equal(inner\text{-}pasch(V1::'a,V2,X,V3,V4),inner\text{-}pasch(V1::'a,V2,\\
\&\\
\;(\forall\, X\ V1\ V2\ V3\ Y\ V4.\ equal(X::'a,Y) \longrightarrow equal(inner\text{-}pasch(V1::'a,V2,V3,X,V4),inner\text{-}pasch(V1::'a,V2,\\
\&\\
\;(\forall\, X\ V1\ V2\ V3\ V4\ Y.\ equal(X::'a,Y) \longrightarrow equal(inner\text{-}pasch(V1::'a,V2,V3,V4,X),inner\text{-}pasch(V1::'a,V2,\\
\&\\
\;(\forall\, A\ B\ C\ D\ E\ F'.\ equal(A::'a,B) \longrightarrow equal(euclid1(A::'a,C,D,E,F'),euclid1(B::'a,C,D,E,F')))\\
\&\\
\;(\forall\, G\ I'\ H\ J\ K'\ L.\ equal(G::'a,H) \longrightarrow equal(euclid1(I'::'a,G,J,K',L),euclid1(I'::'a,H,J,K',L)))\\
\&\\
\;(\forall\, M\ O'\ P\ N\ Q\ R.\ equal(M::'a,N) \longrightarrow equal(euclid1(O'::'a,P,M,Q,R),euclid1(O'::'a,P,N,Q,R)))\\
\&\\
\;(\forall\, S'\ U\ V\ W\ T'\ X.\ equal(S'::'a,T') \longrightarrow equal(euclid1(U::'a,V,W,S',X),euclid1(U::'a,V,W,T',X)))\\
\&\\
\;(\forall\, Y\ A1\ B1\ C1\ D1\ Z.\ equal(Y::'a,Z) \longrightarrow equal(euclid1(A1::'a,B1,C1,D1,Y),euclid1(A1::'a,B1,C1,D1,Z)\\
\&\\
\;(\forall\, E1\ F1\ G1\ H1\ I1\ J1.\ equal(E1::'a,F1) \longrightarrow equal(euclid2(E1::'a,G1,H1,I1,J1),euclid2(F1::'a,G1,H1,I1,\\
\&\\
\;(\forall\, K1\ M1\ L1\ N1\ O1\ P1.\ equal(K1::'a,L1) \longrightarrow equal(euclid2(M1::'a,K1,N1,O1,P1),euclid2(M1::'a,L1,N1,\\
\&
\end{array}
$$

$(\forall\ Q1\ S1\ T1\ R1\ U1\ V1.\ equal(Q1::'a,R1) \longrightarrow equal(euclid2(S1::'a,T1,Q1,U1,V1),euclid2(S1::'a,T1,R1,U$
&
$(\forall\ W1\ Y1\ Z1\ A2\ X1\ B2.\ equal(W1::'a,X1) \longrightarrow equal(euclid2(Y1::'a,Z1,A2,W1,B2),euclid2(Y1::'a,Z1,A2$
&
$(\forall\ C2\ E2\ F2\ G2\ H2\ D2.\ equal(C2::'a,D2) \longrightarrow equal(euclid2(E2::'a,F2,G2,H2,C2),euclid2(E2::'a,F2,G2,H$
&
$(\forall\ X\ Y\ V1\ V2\ V3.\ equal(X::'a,Y) \longrightarrow equal(extension(X::'a,V1,V2,V3),extension(Y::'a,V1,V2,V3)))$
&
$(\forall\ X\ V1\ Y\ V2\ V3.\ equal(X::'a,Y) \longrightarrow equal(extension(V1::'a,X,V2,V3),extension(V1::'a,Y,V2,V3)))$
&
$(\forall\ X\ V1\ V2\ Y\ V3.\ equal(X::'a,Y) \longrightarrow equal(extension(V1::'a,V2,X,V3),extension(V1::'a,V2,Y,V3)))$
&
$(\forall\ X\ V1\ V2\ V3\ Y.\ equal(X::'a,Y) \longrightarrow equal(extension(V1::'a,V2,V3,X),extension(V1::'a,V2,V3,Y)))$
&
$(\forall\ X\ Y\ V1\ V2\ V3\ V4\ V5.\ equal(X::'a,Y) \longrightarrow equal(continuous(X::'a,V1,V2,V3,V4,V5),continuous(Y::'a,$
&
$(\forall\ X\ V1\ Y\ V2\ V3\ V4\ V5.\ equal(X::'a,Y) \longrightarrow equal(continuous(V1::'a,X,V2,V3,V4,V5),continuous(V1::'a$
&
$(\forall\ X\ V1\ V2\ Y\ V3\ V4\ V5.\ equal(X::'a,Y) \longrightarrow equal(continuous(V1::'a,V2,X,V3,V4,V5),continuous(V1::'a$
&
$(\forall\ X\ V1\ V2\ V3\ Y\ V4\ V5.\ equal(X::'a,Y) \longrightarrow equal(continuous(V1::'a,V2,V3,X,V4,V5),continuous(V1::'a$
&
$(\forall\ X\ V1\ V2\ V3\ V4\ Y\ V5.\ equal(X::'a,Y) \longrightarrow equal(continuous(V1::'a,V2,V3,V4,X,V5),continuous(V1::'a$
&
$(\forall\ X\ V1\ V2\ V3\ V4\ V5\ Y.\ equal(X::'a,Y) \longrightarrow equal(continuous(V1::'a,V2,V3,V4,V5,X),continuous(V1::'a$

lemma *GEO017-2*:
EQU001-0-ax equal &
GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3
lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension
between equal equidistant &
$(equidistant(u::'a,v,w,x))$ &
$(\sim equidistant(u::'a,v,x,w)) \longrightarrow False$
⟨*proof*⟩

lemma *GEO027-3*:
EQU001-0-ax equal &
GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3
lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension
between equal equidistant &
$(\forall\ U\ V.\ equal(reflection(U::'a,V),extension(U::'a,V,U,V)))$ &
$(\forall\ X\ Y\ Z.\ equal(X::'a,Y) \longrightarrow equal(reflection(X::'a,Z),reflection(Y::'a,Z)))$ &
$(\forall\ A1\ C1\ B1.\ equal(A1::'a,B1) \longrightarrow equal(reflection(C1::'a,A1),reflection(C1::'a,B1)))$
&
$(\forall\ U\ V.\ equidistant(U::'a,V,U,V))$ &
$(\forall\ W\ X\ U\ V.\ equidistant(U::'a,V,W,X) \longrightarrow equidistant(W::'a,X,U,V))$ &
$(\forall\ V\ U\ W\ X.\ equidistant(U::'a,V,W,X) \longrightarrow equidistant(V::'a,U,W,X))$ &

$(\forall\ U\ V\ X\ W.\ equidistant(U::'a,V,W,X)\ \longrightarrow\ equidistant(U::'a,V,X,W))\ \&$
$(\forall\ V\ U\ X\ W.\ equidistant(U::'a,V,W,X)\ \longrightarrow\ equidistant(V::'a,U,X,W))\ \&$
$(\forall\ W\ X\ V\ U.\ equidistant(U::'a,V,W,X)\ \longrightarrow\ equidistant(W::'a,X,V,U))\ \&$
$(\forall\ X\ W\ U\ V.\ equidistant(U::'a,V,W,X)\ \longrightarrow\ equidistant(X::'a,W,U,V))\ \&$
$(\forall\ X\ W\ V\ U.\ equidistant(U::'a,V,W,X)\ \longrightarrow\ equidistant(X::'a,W,V,U))\ \&$
$(\forall\ W\ X\ U\ V\ Y\ Z.\ equidistant(U::'a,V,W,X)\ \&\ equidistant(W::'a,X,Y,Z)\ \longrightarrow$
$equidistant(U::'a,V,Y,Z))\ \&$
$(\forall\ U\ V\ W.\ equal(V::'a,extension(U::'a,V,W,W)))\ \&$
$(\forall\ W\ X\ U\ V\ Y.\ equal(Y::'a,extension(U::'a,V,W,X))\ \longrightarrow\ between(U::'a,V,Y))$
$\&$
$(\forall\ U\ V.\ between(U::'a,V,reflection(U::'a,V)))\ \&$
$(\forall\ U\ V.\ equidistant(V::'a,reflection(U::'a,V),U,V))\ \&$
$(\forall\ U\ V.\ equal(U::'a,V)\ \longrightarrow\ equal(V::'a,reflection(U::'a,V)))\ \&$
$(\forall\ U.\ equal(U::'a,reflection(U::'a,U)))\ \&$
$(\forall\ U\ V.\ equal(V::'a,reflection(U::'a,V))\ \longrightarrow\ equal(U::'a,V))\ \&$
$(\forall\ V\ V1\ U\ W\ U1\ W1.\ equidistant(U::'a,V,U1,V1)\ \&\ equidistant(V::'a,W,V1,W1)$
$\&\ between(U::'a,V,W)\ \&\ between(U1::'a,V1,W1)\ \longrightarrow\ equidistant(U::'a,W,U1,W1))$
$\&$
$(\forall\ U\ V\ W\ X.\ between(U::'a,V,W)\ \&\ between(U::'a,V,X)\ \&\ equidistant(V::'a,W,V,X)$
$\longrightarrow\ equal(U::'a,V)\ |\ equal(W::'a,X))\ \&$
$(between(u::'a,v,w))\ \&$
$(\sim equal(u::'a,v))\ \&$
$(\sim equal(w::'a,extension(u::'a,v,v,w)))\ \longrightarrow\ False$
$\langle proof\rangle$

lemma *GEO058-2*:
EQU001-0-ax equal &
GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3
lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension
between equal equidistant &
$(\forall\ U\ V.\ equal(reflection(U::'a,V),extension(U::'a,V,U,V)))\ \&$
$(\forall\ X\ Y\ Z.\ equal(X::'a,Y)\ \longrightarrow\ equal(reflection(X::'a,Z),reflection(Y::'a,Z)))\ \&$
$(\forall\ A1\ C1\ B1.\ equal(A1::'a,B1)\ \longrightarrow\ equal(reflection(C1::'a,A1),reflection(C1::'a,B1)))$
$\&$
$(equal(v::'a,reflection(u::'a,v)))\ \&$
$(\sim equal(u::'a,v))\ \longrightarrow\ False$
$\langle proof\rangle$

lemma *GEO079-1*:
$(\forall\ U\ V\ W\ X\ Y\ Z.\ right\text{-}angle(U::'a,V,W)\ \&\ right\text{-}angle(X::'a,Y,Z)\ \longrightarrow\ eq(U::'a,V,W,X,Y,Z))$
$\&$
$(\forall\ U\ V\ W\ X\ Y\ Z.\ CONGRUENT(U::'a,V,W,X,Y,Z)\ \longrightarrow\ eq(U::'a,V,W,X,Y,Z))$
$\&$
$(\forall\ V\ W\ U\ X.\ trapezoid(U::'a,V,W,X)\ \longrightarrow\ parallel(V::'a,W,U,X))\ \&$
$(\forall\ U\ V\ X\ Y.\ parallel(U::'a,V,X,Y)\ \longrightarrow\ eq(X::'a,V,U,V,X,Y))\ \&$
$(trapezoid(a::'a,b,c,d))\ \&$

$(\sim eq(a::'a,c,b,c,a,d))$ $--> False$
$\langle proof \rangle$

abbreviation *GRP003-0-ax equal multiply INVERSE identity product* $\equiv$
$(\forall X.\ product(identity::'a,X,X))$ &
$(\forall X.\ product(X::'a,identity,X))$ &
$(\forall X.\ product(INVERSE(X),X,identity))$ &
$(\forall X.\ product(X::'a,INVERSE(X),identity))$ &
$(\forall X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall X\ Y\ Z\ W.\ product(X::'a,Y,Z)$ & $product(X::'a,Y,W)$ $--> equal(Z::'a,W))$
&
$(\forall Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(U::'a,Z,W)$
$--> product(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(X::'a,V,W)$
$--> product(U::'a,Z,W))$

abbreviation *GRP003-0-eq product multiply INVERSE equal* $\equiv$
$(\forall X\ Y.\ equal(X::'a,Y)$ $--> equal(INVERSE(X),INVERSE(Y)))$ &
$(\forall X\ Y\ W.\ equal(X::'a,Y)$ $--> equal(multiply(X::'a,W),multiply(Y::'a,W)))$
&
$(\forall X\ W\ Y.\ equal(X::'a,Y)$ $--> equal(multiply(W::'a,X),multiply(W::'a,Y)))$
&
$(\forall X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $product(X::'a,W,Z)$ $--> product(Y::'a,W,Z))$
&
$(\forall X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $product(W::'a,X,Z)$ $--> product(W::'a,Y,Z))$
&
$(\forall X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $product(W::'a,Z,X)$ $--> product(W::'a,Z,Y))$

lemma *GRP001-1*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall X.\ product(X::'a,X,identity))$ &
$(product(a::'a,b,c))$ &
$(\sim product(b::'a,a,c))$ $--> False$
$\langle proof \rangle$

lemma *GRP008-1*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall A\ B.\ equal(A::'a,B)$ $--> equal(h(A),h(B)))$ &
$(\forall C\ D.\ equal(C::'a,D)$ $--> equal(j(C),j(D)))$ &
$(\forall A\ B.\ equal(A::'a,B)$ & $q(A)$ $--> q(B))$ &
$(\forall B\ A\ C.\ q(A)$ & $product(A::'a,B,C)$ $--> product(B::'a,A,C))$ &
$(\forall A.\ product(j(A),A,h(A))$ | $product(A::'a,j(A),h(A))$ | $q(A))$ &
$(\forall A.\ product(j(A),A,h(A))$ & $product(A::'a,j(A),h(A))$ $--> q(A))$ &

$(\sim q(identity)) \longrightarrow False$
$\langle proof \rangle$

lemma *GRP013-1*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall A.\ product(A::'a,A,identity))$ &
$(product(a::'a,b,c))$ &
$(product(INVERSE(a),INVERSE(b),d))$ &
$(\forall A\ C\ B.\ product(INVERSE(A),INVERSE(B),C) \longrightarrow product(A::'a,C,B))$ &
$(\sim product(c::'a,d,identity)) \longrightarrow False$
$\langle proof \rangle$

lemma *GRP037-3*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall A\ B\ C.\ subgroup\text{-}member(A)$ & $subgroup\text{-}member(B)$ & $product(A::'a,INVERSE(B),C)$
$\longrightarrow subgroup\text{-}member(C))$ &
$(\forall A\ B.\ equal(A::'a,B)$ & $subgroup\text{-}member(A) \longrightarrow subgroup\text{-}member(B))$ &
$(\forall A.\ subgroup\text{-}member(A) \longrightarrow product(Gidentity::'a,A,A))$ &
$(\forall A.\ subgroup\text{-}member(A) \longrightarrow product(A::'a,Gidentity,A))$ &
$(\forall A.\ subgroup\text{-}member(A) \longrightarrow product(A::'a,Ginverse(A),Gidentity))$ &
$(\forall A.\ subgroup\text{-}member(A) \longrightarrow product(Ginverse(A),A,Gidentity))$ &
$(\forall A.\ subgroup\text{-}member(A) \longrightarrow subgroup\text{-}member(Ginverse(A)))$ &
$(\forall A\ B.\ equal(A::'a,B) \longrightarrow equal(Ginverse(A),Ginverse(B)))$ &
$(\forall A\ C\ D\ B.\ product(A::'a,B,C)$ & $product(A::'a,D,C) \longrightarrow equal(D::'a,B))$ &
$(\forall B\ C\ D\ A.\ product(A::'a,B,C)$ & $product(D::'a,B,C) \longrightarrow equal(D::'a,A))$ &
$(subgroup\text{-}member(a))$ &
$(subgroup\text{-}member(Gidentity))$ &
$(\sim equal(INVERSE(a),Ginverse(a))) \longrightarrow False$
$\langle proof \rangle$

lemma *GRP031-2*:
$(\forall X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall X\ Y\ Z\ W.\ product(X::'a,Y,Z)$ & $product(X::'a,Y,W) \longrightarrow equal(Z::'a,W))$
&
$(\forall Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(U::'a,Z,W)$
$\longrightarrow product(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(X::'a,V,W)$
$\longrightarrow product(U::'a,Z,W))$ &
$(\forall A.\ product(A::'a,INVERSE(A),identity))$ &
$(\forall A.\ product(A::'a,identity,A))$ &
$(\forall A.\ \sim product(A::'a,a,identity)) \longrightarrow False$
$\langle proof \rangle$

lemma *GRP034-4*:
(∀ *X Y*. *product*(*X*::'*a*,*Y*,*multiply*(*X*::'*a*,*Y*))) &
(∀ *X*. *product*(*identity*::'*a*,*X*,*X*)) &
(∀ *X*. *product*(*X*::'*a*,*identity*,*X*)) &
(∀ *X*. *product*(*X*::'*a*,*INVERSE*(*X*),*identity*)) &
(∀ *Y U Z X V W*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*Y*::'*a*,*Z*,*V*) & *product*(*U*::'*a*,*Z*,*W*)
−−> *product*(*X*::'*a*,*V*,*W*)) &
(∀ *Y X V U Z W*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*Y*::'*a*,*Z*,*V*) & *product*(*X*::'*a*,*V*,*W*)
−−> *product*(*U*::'*a*,*Z*,*W*)) &
(∀ *B A C*. *subgroup-member*(*A*) & *subgroup-member*(*B*) & *product*(*B*::'*a*,*INVERSE*(*A*),*C*)
−−> *subgroup-member*(*C*)) &
(*subgroup-member*(*a*)) &
(~*subgroup-member*(*INVERSE*(*a*))) −−> *False*
⟨*proof*⟩

lemma *GRP047-2*:
(∀ *X*. *product*(*identity*::'*a*,*X*,*X*)) &
(∀ *X*. *product*(*INVERSE*(*X*),*X*,*identity*)) &
(∀ *X Y*. *product*(*X*::'*a*,*Y*,*multiply*(*X*::'*a*,*Y*))) &
(∀ *X Y Z W*. *product*(*X*::'*a*,*Y*,*Z*) & *product*(*X*::'*a*,*Y*,*W*) −−> *equal*(*Z*::'*a*,*W*))
&
(∀ *Y U Z X V W*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*Y*::'*a*,*Z*,*V*) & *product*(*U*::'*a*,*Z*,*W*)
−−> *product*(*X*::'*a*,*V*,*W*)) &
(∀ *Y X V U Z W*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*Y*::'*a*,*Z*,*V*) & *product*(*X*::'*a*,*V*,*W*)
−−> *product*(*U*::'*a*,*Z*,*W*)) &
(∀ *X W Z Y*. *equal*(*X*::'*a*,*Y*) & *product*(*W*::'*a*,*Z*,*X*) −−> *product*(*W*::'*a*,*Z*,*Y*))
&
(*equal*(*a*::'*a*,*b*)) &
(~*equal*(*multiply*(*c*::'*a*,*a*),*multiply*(*c*::'*a*,*b*))) −−> *False*
⟨*proof*⟩

lemma *GRP130-1-002*:
(*group-element*(*e-1*)) &
(*group-element*(*e-2*)) &
(~*equal*(*e-1*::'*a*,*e-2*)) &
(~*equal*(*e-2*::'*a*,*e-1*)) &
(∀ *X Y*. *group-element*(*X*) & *group-element*(*Y*) −−> *product*(*X*::'*a*,*Y*,*e-1*) |
product(*X*::'*a*,*Y*,*e-2*)) &
(∀ *X Y W Z*. *product*(*X*::'*a*,*Y*,*W*) & *product*(*X*::'*a*,*Y*,*Z*) −−> *equal*(*W*::'*a*,*Z*))
&
(∀ *X Y W Z*. *product*(*X*::'*a*,*W*,*Y*) & *product*(*X*::'*a*,*Z*,*Y*) −−> *equal*(*W*::'*a*,*Z*))
&
(∀ *Y X W Z*. *product*(*W*::'*a*,*Y*,*X*) & *product*(*Z*::'*a*,*Y*,*X*) −−> *equal*(*W*::'*a*,*Z*))
&
(∀ *Z1 Z2 Y X*. *product*(*X*::'*a*,*Y*,*Z1*) & *product*(*X*::'*a*,*Z1*,*Z2*) −−> *product*(*Z2*::'*a*,*Y*,*X*))

$--> False$
$\langle proof \rangle$

abbreviation *GRP004-0-ax INVERSE identity multiply equal* $\equiv$
$(\forall X.\ equal(multiply(identity::'a,X),X))$ &
$(\forall X.\ equal(multiply(INVERSE(X),X),identity))$ &
$(\forall X\ Y\ Z.\ equal(multiply(multiply(X::'a,Y),Z),multiply(X::'a,multiply(Y::'a,Z))))$
&
$(\forall A\ B.\ equal(A::'a,B) --> equal(INVERSE(A),INVERSE(B)))$ &
$(\forall C\ D\ E.\ equal(C::'a,D) --> equal(multiply(C::'a,E),multiply(D::'a,E)))$ &
$(\forall F'\ H\ G.\ equal(F'::'a,G) --> equal(multiply(H::'a,F'),multiply(H::'a,G)))$

abbreviation *GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal*
$\equiv$
$(\forall Y\ X.\ equal(greatest\text{-}lower\text{-}bound(X::'a,Y),greatest\text{-}lower\text{-}bound(Y::'a,X)))$ &
$(\forall Y\ X.\ equal(least\text{-}upper\text{-}bound(X::'a,Y),least\text{-}upper\text{-}bound(Y::'a,X)))$ &
$(\forall X\ Y\ Z.\ equal(greatest\text{-}lower\text{-}bound(X::'a,greatest\text{-}lower\text{-}bound(Y::'a,Z)),greatest\text{-}lower\text{-}bound(greatest\text{-}lowe$
&
$(\forall X\ Y\ Z.\ equal(least\text{-}upper\text{-}bound(X::'a,least\text{-}upper\text{-}bound(Y::'a,Z)),least\text{-}upper\text{-}bound(least\text{-}upper\text{-}bound(X:$
&
$(\forall X.\ equal(least\text{-}upper\text{-}bound(X::'a,X),X))$ &
$(\forall X.\ equal(greatest\text{-}lower\text{-}bound(X::'a,X),X))$ &
$(\forall Y\ X.\ equal(least\text{-}upper\text{-}bound(X::'a,greatest\text{-}lower\text{-}bound(X::'a,Y)),X))$ &
$(\forall Y\ X.\ equal(greatest\text{-}lower\text{-}bound(X::'a,least\text{-}upper\text{-}bound(X::'a,Y)),X))$ &
$(\forall Y\ X\ Z.\ equal(multiply(X::'a,least\text{-}upper\text{-}bound(Y::'a,Z)),least\text{-}upper\text{-}bound(multiply(X::'a,Y),multiply(X:$
&
$(\forall Y\ X\ Z.\ equal(multiply(X::'a,greatest\text{-}lower\text{-}bound(Y::'a,Z)),greatest\text{-}lower\text{-}bound(multiply(X::'a,Y),multi$
&
$(\forall Y\ Z\ X.\ equal(multiply(least\text{-}upper\text{-}bound(Y::'a,Z),X),least\text{-}upper\text{-}bound(multiply(Y::'a,X),multiply(Z::'a,$
&
$(\forall Y\ Z\ X.\ equal(multiply(greatest\text{-}lower\text{-}bound(Y::'a,Z),X),greatest\text{-}lower\text{-}bound(multiply(Y::'a,X),multiply($
&
$(\forall A\ B\ C.\ equal(A::'a,B) --> equal(greatest\text{-}lower\text{-}bound(A::'a,C),greatest\text{-}lower\text{-}bound(B::'a,C)))$
&
$(\forall A\ C\ B.\ equal(A::'a,B) --> equal(greatest\text{-}lower\text{-}bound(C::'a,A),greatest\text{-}lower\text{-}bound(C::'a,B)))$
&
$(\forall A\ B\ C.\ equal(A::'a,B) --> equal(least\text{-}upper\text{-}bound(A::'a,C),least\text{-}upper\text{-}bound(B::'a,C)))$
&
$(\forall A\ C\ B.\ equal(A::'a,B) --> equal(least\text{-}upper\text{-}bound(C::'a,A),least\text{-}upper\text{-}bound(C::'a,B)))$
&
$(\forall A\ B\ C.\ equal(A::'a,B) --> equal(multiply(A::'a,C),multiply(B::'a,C)))$ &
$(\forall A\ C\ B.\ equal(A::'a,B) --> equal(multiply(C::'a,A),multiply(C::'a,B)))$

lemma *GRP156-1*:
EQU001-0-ax equal &
GRP004-0-ax INVERSE identity multiply equal &
GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal &
$(equal(least\text{-}upper\text{-}bound(a::'a,b),b))$ &

(~ *equal*(*greatest-lower-bound*(*multiply*(*a*::'*a*,*c*),*multiply*(*b*::'*a*,*c*)),*multiply*(*a*::'*a*,*c*)))
−−> *False*
⟨*proof*⟩

lemma *GRP168-1*:
EQU001-0-ax equal &
GRP004-0-ax INVERSE identity multiply equal &
GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal &
(*equal*(*least-upper-bound*(*a*::'*a*,*b*),*b*)) &
(~ *equal*(*least-upper-bound*(*multiply*(*INVERSE*(*c*),*multiply*(*a*::'*a*,*c*)),*multiply*(*INVERSE*(*c*),*multiply*(*b*::'*a*,*c*))
−−> *False*
⟨*proof*⟩

abbreviation *HEN002-0-ax identity Zero Divide equal mless-equal* ≡
(∀ *X Y*. *mless-equal*(*X*::'*a*,*Y*) −−> *equal*(*Divide*(*X*::'*a*,*Y*),*Zero*)) &
(∀ *X Y*. *equal*(*Divide*(*X*::'*a*,*Y*),*Zero*) −−> *mless-equal*(*X*::'*a*,*Y*)) &
(∀ *Y X*. *mless-equal*(*Divide*(*X*::'*a*,*Y*),*X*)) &
(∀ *X Y Z*. *mless-equal*(*Divide*(*Divide*(*X*::'*a*,*Z*),*Divide*(*Y*::'*a*,*Z*)),*Divide*(*Divide*(*X*::'*a*,*Y*),*Z*)))
&
(∀ *X*. *mless-equal*(*Zero*::'*a*,*X*)) &
(∀ *X Y*. *mless-equal*(*X*::'*a*,*Y*) & *mless-equal*(*Y*::'*a*,*X*) −−> *equal*(*X*::'*a*,*Y*)) &
(∀ *X*. *mless-equal*(*X*::'*a*,*identity*))

abbreviation *HEN002-0-eq mless-equal Divide equal* ≡
(∀ *A B C*. *equal*(*A*::'*a*,*B*) −−> *equal*(*Divide*(*A*::'*a*,*C*),*Divide*(*B*::'*a*,*C*))) &
(∀ *D F' E*. *equal*(*D*::'*a*,*E*) −−> *equal*(*Divide*(*F'*::'*a*,*D*),*Divide*(*F'*::'*a*,*E*))) &
(∀ *G H I'*. *equal*(*G*::'*a*,*H*) & *mless-equal*(*G*::'*a*,*I'*) −−> *mless-equal*(*H*::'*a*,*I'*)) &
(∀ *J L K'*. *equal*(*J*::'*a*,*K'*) & *mless-equal*(*L*::'*a*,*J*) −−> *mless-equal*(*L*::'*a*,*K'*))

lemma *HEN003-3*:
EQU001-0-ax equal &
HEN002-0-ax identity Zero Divide equal mless-equal &
HEN002-0-eq mless-equal Divide equal &
(~ *equal*(*Divide*(*a*::'*a*,*a*),*Zero*)) −−> *False*
⟨*proof*⟩

lemma *HEN007-2*:
EQU001-0-ax equal &
(∀ *X Y*. *mless-equal*(*X*::'*a*,*Y*) −−> *quotient*(*X*::'*a*,*Y*,*Zero*)) &
(∀ *X Y*. *quotient*(*X*::'*a*,*Y*,*Zero*) −−> *mless-equal*(*X*::'*a*,*Y*)) &
(∀ *Y Z X*. *quotient*(*X*::'*a*,*Y*,*Z*) −−> *mless-equal*(*Z*::'*a*,*X*)) &
(∀ *Y X V3 V2 V1 Z V4 V5*. *quotient*(*X*::'*a*,*Y*,*V1*) & *quotient*(*Y*::'*a*,*Z*,*V2*) &
quotient(*X*::'*a*,*Z*,*V3*) & *quotient*(*V3*::'*a*,*V2*,*V4*) & *quotient*(*V1*::'*a*,*Z*,*V5*) −−>
mless-equal(*V4*::'*a*,*V5*)) &
(∀ *X*. *mless-equal*(*Zero*::'*a*,*X*)) &
(∀ *X Y*. *mless-equal*(*X*::'*a*,*Y*) & *mless-equal*(*Y*::'*a*,*X*) −−> *equal*(*X*::'*a*,*Y*)) &

$(\forall X.\ mless\text{-}equal(X::'a,identity))$ &
$(\forall X\ Y.\ quotient(X::'a,Y,Divide(X::'a,Y)))$ &
$(\forall X\ Y\ Z\ W.\ quotient(X::'a,Y,Z)$ & $quotient(X::'a,Y,W) \longrightarrow equal(Z::'a,W))$
&
$(\forall X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $quotient(X::'a,W,Z) \longrightarrow quotient(Y::'a,W,Z))$
&
$(\forall X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $quotient(W::'a,X,Z) \longrightarrow quotient(W::'a,Y,Z))$
&
$(\forall X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $quotient(W::'a,Z,X) \longrightarrow quotient(W::'a,Z,Y))$
&
$(\forall X\ Z\ Y.\ equal(X::'a,Y)$ & $mless\text{-}equal(Z::'a,X) \longrightarrow mless\text{-}equal(Z::'a,Y))$ &
$(\forall X\ Y\ Z.\ equal(X::'a,Y)$ & $mless\text{-}equal(X::'a,Z) \longrightarrow mless\text{-}equal(Y::'a,Z))$ &
$(\forall X\ Y\ W.\ equal(X::'a,Y) \longrightarrow equal(Divide(X::'a,W),Divide(Y::'a,W)))$ &
$(\forall X\ W\ Y.\ equal(X::'a,Y) \longrightarrow equal(Divide(W::'a,X),Divide(W::'a,Y)))$ &
$(\forall X.\ quotient(X::'a,identity,Zero))$ &
$(\forall X.\ quotient(Zero::'a,X,Zero))$ &
$(\forall X.\ quotient(X::'a,X,Zero))$ &
$(\forall X.\ quotient(X::'a,Zero,X))$ &
$(\forall Y\ X\ Z.\ mless\text{-}equal(X::'a,Y)$ & $mless\text{-}equal(Y::'a,Z) \longrightarrow mless\text{-}equal(X::'a,Z))$
&
$(\forall W1\ X\ Z\ W2\ Y.\ quotient(X::'a,Y,W1)$ & $mless\text{-}equal(W1::'a,Z)$ & $quotient(X::'a,Z,W2)$
$\longrightarrow mless\text{-}equal(W2::'a,Y))$ &
$(mless\text{-}equal(x::'a,y))$ &
$(quotient(z::'a,y,zQy))$ &
$(quotient(z::'a,x,zQx))$ &
$(\sim mless\text{-}equal(zQy::'a,zQx)) \longrightarrow False$
$\langle proof \rangle$

lemma *HEN008-4*:
EQU001-0-ax equal &
HEN002-0-ax identity Zero Divide equal mless-equal &
HEN002-0-eq mless-equal Divide equal &
$(\forall X.\ equal(Divide(X::'a,identity),Zero))$ &
$(\forall X.\ equal(Divide(Zero::'a,X),Zero))$ &
$(\forall X.\ equal(Divide(X::'a,X),Zero))$ &
$(equal(Divide(a::'a,Zero),a))$ &
$(\forall Y\ X\ Z.\ mless\text{-}equal(X::'a,Y)$ & $mless\text{-}equal(Y::'a,Z) \longrightarrow mless\text{-}equal(X::'a,Z))$
&
$(\forall X\ Z\ Y.\ mless\text{-}equal(Divide(X::'a,Y),Z) \longrightarrow mless\text{-}equal(Divide(X::'a,Z),Y))$
&
$(\forall Y\ Z\ X.\ mless\text{-}equal(X::'a,Y) \longrightarrow mless\text{-}equal(Divide(Z::'a,Y),Divide(Z::'a,X)))$
&
$(mless\text{-}equal(a::'a,b))$ &
$(\sim mless\text{-}equal(Divide(a::'a,c),Divide(b::'a,c))) \longrightarrow False$
$\langle proof \rangle$

lemma *HEN009-5*:

EQU001-0-ax equal &
(∀ *Y X*. *equal*(*Divide*(*Divide*(*X*::'*a*,*Y*),*X*),*Zero*)) &
(∀ *X Y Z*. *equal*(*Divide*(*Divide*(*Divide*(*X*::'*a*,*Z*),*Divide*(*Y*::'*a*,*Z*)),*Divide*(*Divide*(*X*::'*a*,*Y*),*Z*)),*Zero*))
&
(∀ *X*. *equal*(*Divide*(*Zero*::'*a*,*X*),*Zero*)) &
(∀ *X Y*. *equal*(*Divide*(*X*::'*a*,*Y*),*Zero*) & *equal*(*Divide*(*Y*::'*a*,*X*),*Zero*) −−> *equal*(*X*::'*a*,*Y*))
&
(∀ *X*. *equal*(*Divide*(*X*::'*a*,*identity*),*Zero*)) &
(∀ *A B C*. *equal*(*A*::'*a*,*B*) −−> *equal*(*Divide*(*A*::'*a*,*C*),*Divide*(*B*::'*a*,*C*))) &
(∀ *D F' E*. *equal*(*D*::'*a*,*E*) −−> *equal*(*Divide*(*F'*::'*a*,*D*),*Divide*(*F'*::'*a*,*E*))) &
(∀ *Y X Z*. *equal*(*Divide*(*X*::'*a*,*Y*),*Zero*) & *equal*(*Divide*(*Y*::'*a*,*Z*),*Zero*) −−>
equal(*Divide*(*X*::'*a*,*Z*),*Zero*)) &
(∀ *X Z Y*. *equal*(*Divide*(*Divide*(*X*::'*a*,*Y*),*Z*),*Zero*) −−> *equal*(*Divide*(*Divide*(*X*::'*a*,*Z*),*Y*),*Zero*))
&
(∀ *Y Z X*. *equal*(*Divide*(*X*::'*a*,*Y*),*Zero*) −−> *equal*(*Divide*(*Divide*(*Z*::'*a*,*Y*),*Divide*(*Z*::'*a*,*X*)),*Zero*))
&
(~*equal*(*Divide*(*identity*::'*a*,*a*),*Divide*(*identity*::'*a*,*Divide*(*identity*::'*a*,*Divide*(*identity*::'*a*,*a*)))))
&
(*equal*(*Divide*(*identity*::'*a*,*a*),*b*)) &
(*equal*(*Divide*(*identity*::'*a*,*b*),*c*)) &
(*equal*(*Divide*(*identity*::'*a*,*c*),*d*)) &
(~*equal*(*b*::'*a*,*d*)) −−> *False*
⟨*proof*⟩

lemma *HEN012-3*:
EQU001-0-ax equal &
HEN002-0-ax identity Zero Divide equal mless-equal &
HEN002-0-eq mless-equal Divide equal &
(~*mless-equal*(*a*::'*a*,*a*)) −−> *False*
⟨*proof*⟩

lemma *LCL010-1*:
(∀ *X Y*. *is-a-theorem*(*equivalent*(*X*::'*a*,*Y*)) & *is-a-theorem*(*X*) −−> *is-a-theorem*(*Y*))
&
(∀ *X Z Y*. *is-a-theorem*(*equivalent*(*equivalent*(*X*::'*a*,*Y*),*equivalent*(*equivalent*(*X*::'*a*,*Z*),*equivalent*(*Z*::'*a*,*Y*))))
&
(~*is-a-theorem*(*equivalent*(*equivalent*(*a*::'*a*,*b*),*equivalent*(*equivalent*(*c*::'*a*,*b*),*equivalent*(*a*::'*a*,*c*)))))
−−> *False*
⟨*proof*⟩

lemma *LCL077-2*:
(∀ *X Y*. *is-a-theorem*(*implies*(*X*,*Y*)) & *is-a-theorem*(*X*) −−> *is-a-theorem*(*Y*))
&
(∀ *Y X*. *is-a-theorem*(*implies*(*X*,*implies*(*Y*,*X*)))) &
(∀ *Y X Z*. *is-a-theorem*(*implies*(*implies*(*X*,*implies*(*Y*,*Z*)),*implies*(*implies*(*X*,*Y*),*implies*(*X*,*Z*)))))

&
(∀ Y X. is-a-theorem(implies(implies(not(X),not(Y)),implies(Y,X)))) &
(∀ X2 X1 X3. is-a-theorem(implies(X1,X2)) & is-a-theorem(implies(X2,X3))
−−> is-a-theorem(implies(X1,X3))) &
(~is-a-theorem(implies(not(not(a)),a))) −−> False
⟨proof⟩

lemma *LCL082-1*:
(∀ X Y. is-a-theorem(implies(X::'a,Y)) & is-a-theorem(X) −−> is-a-theorem(Y))
&
(∀ Y Z U X. is-a-theorem(implies(implies(implies(X::'a,Y),Z),implies(implies(Z::'a,X),implies(U::'a,X)))))
&
(~is-a-theorem(implies(a::'a,implies(b::'a,a)))) −−> False
⟨proof⟩

lemma *LCL111-1*:
(∀ X Y. is-a-theorem(implies(X,Y)) & is-a-theorem(X) −−> is-a-theorem(Y))
&
(∀ Y X. is-a-theorem(implies(X,implies(Y,X)))) &
(∀ Y X Z. is-a-theorem(implies(implies(X,Y),implies(implies(Y,Z),implies(X,Z)))))
&
(∀ Y X. is-a-theorem(implies(implies(implies(X,Y),Y),implies(implies(Y,X),X))))
&
(∀ Y X. is-a-theorem(implies(implies(not(X),not(Y)),implies(Y,X)))) &
(~is-a-theorem(implies(implies(a,b),implies(implies(c,a),implies(c,b))))) −−> False
⟨proof⟩

lemma *LCL143-1*:
(∀ X. equal(X,X)) &
(∀ Y X. equal(X,Y) −−> equal(Y,X)) &
(∀ Y X Z. equal(X,Y) & equal(Y,Z) −−> equal(X,Z)) &
(∀ X. equal(implies(true,X),X)) &
(∀ Y X Z. equal(implies(implies(X,Y),implies(implies(Y,Z),implies(X,Z))),true))
&
(∀ Y X. equal(implies(implies(X,Y),Y),implies(implies(Y,X),X))) &
(∀ Y X. equal(implies(implies(not(X),not(Y)),implies(Y,X)),true)) &
(∀ A B C. equal(A,B) −−> equal(implies(A,C),implies(B,C))) &
(∀ D F' E. equal(D,E) −−> equal(implies(F',D),implies(F',E))) &
(∀ G H. equal(G,H) −−> equal(not(G),not(H))) &
(∀ X Y. equal(big-V(X,Y),implies(implies(X,Y),Y))) &
(∀ X Y. equal(big-hat(X,Y),not(big-V(not(X),not(Y))))) &
(∀ X Y. ordered(X,Y) −−> equal(implies(X,Y),true)) &
(∀ X Y. equal(implies(X,Y),true) −−> ordered(X,Y)) &
(∀ A B C. equal(A,B) −−> equal(big-V(A,C),big-V(B,C))) &
(∀ D F' E. equal(D,E) −−> equal(big-V(F',D),big-V(F',E))) &
(∀ G H I'. equal(G,H) −−> equal(big-hat(G,I'),big-hat(H,I'))) &

$(\forall\ J\ L\ K'.\ equal(J,K') \longrightarrow equal(big\text{-}hat(L,J),big\text{-}hat(L,K')))\ \&$
$(\forall\ M\ N\ O'.\ equal(M,N)\ \&\ ordered(M,O') \longrightarrow ordered(N,O'))\ \&$
$(\forall\ P\ R\ Q.\ equal(P,Q)\ \&\ ordered(R,P) \longrightarrow ordered(R,Q))\ \&$
$(ordered(x,y))\ \&$
$(\sim ordered(implies(z,x),implies(z,y))) \longrightarrow False$
$\langle proof \rangle$

lemma *LCL182-1*:
$(\forall\ A.\ axiom(or(not(or(A,A)),A)))\ \&$
$(\forall\ B\ A.\ axiom(or(not(A),or(B,A))))\ \&$
$(\forall\ B\ A.\ axiom(or(not(or(A,B)),or(B,A))))\ \&$
$(\forall\ B\ A\ C.\ axiom(or(not(or(A,or(B,C))),or(B,or(A,C)))))\ \&$
$(\forall\ A\ C\ B.\ axiom(or(not(or(not(A),B)),or(not(or(C,A)),or(C,B)))))\ \&$
$(\forall\ X.\ axiom(X) \longrightarrow theorem(X))\ \&$
$(\forall\ X\ Y.\ axiom(or(not(Y),X))\ \&\ theorem(Y) \longrightarrow theorem(X))\ \&$
$(\forall\ X\ Y\ Z.\ axiom(or(not(X),Y))\ \&\ theorem(or(not(Y),Z)) \longrightarrow theorem(or(not(X),Z)))$
$\&$
$(\sim theorem(or(not(or(not(p),q)),or(not(not(q)),not(p))))) \longrightarrow False$
$\langle proof \rangle$

lemma *LCL200-1*:
$(\forall\ A.\ axiom(or(not(or(A,A)),A)))\ \&$
$(\forall\ B\ A.\ axiom(or(not(A),or(B,A))))\ \&$
$(\forall\ B\ A.\ axiom(or(not(or(A,B)),or(B,A))))\ \&$
$(\forall\ B\ A\ C.\ axiom(or(not(or(A,or(B,C))),or(B,or(A,C)))))\ \&$
$(\forall\ A\ C\ B.\ axiom(or(not(or(not(A),B)),or(not(or(C,A)),or(C,B)))))\ \&$
$(\forall\ X.\ axiom(X) \longrightarrow theorem(X))\ \&$
$(\forall\ X\ Y.\ axiom(or(not(Y),X))\ \&\ theorem(Y) \longrightarrow theorem(X))\ \&$
$(\forall\ X\ Y\ Z.\ axiom(or(not(X),Y))\ \&\ theorem(or(not(Y),Z)) \longrightarrow theorem(or(not(X),Z)))$
$\&$
$(\sim theorem(or(not(not(or(p,q))),not(q)))) \longrightarrow False$
$\langle proof \rangle$

lemma *LCL215-1*:
$(\forall\ A.\ axiom(or(not(or(A,A)),A)))\ \&$
$(\forall\ B\ A.\ axiom(or(not(A),or(B,A))))\ \&$
$(\forall\ B\ A.\ axiom(or(not(or(A,B)),or(B,A))))\ \&$
$(\forall\ B\ A\ C.\ axiom(or(not(or(A,or(B,C))),or(B,or(A,C)))))\ \&$
$(\forall\ A\ C\ B.\ axiom(or(not(or(not(A),B)),or(not(or(C,A)),or(C,B)))))\ \&$
$(\forall\ X.\ axiom(X) \longrightarrow theorem(X))\ \&$
$(\forall\ X\ Y.\ axiom(or(not(Y),X))\ \&\ theorem(Y) \longrightarrow theorem(X))\ \&$
$(\forall\ X\ Y\ Z.\ axiom(or(not(X),Y))\ \&\ theorem(or(not(Y),Z)) \longrightarrow theorem(or(not(X),Z)))$
$\&$
$(\sim theorem(or(not(or(not(p),q)),or(not(or(p,q)),q)))) \longrightarrow False$
$\langle proof \rangle$

lemma *LCL230-2*:
$(q \dashrightarrow p \mid r)$ &
$(\sim p)$ &
(q) &
$(\sim r) \dashrightarrow \mathit{False}$
⟨*proof*⟩

lemma *LDA003-1*:
EQU001-0-ax equal &
$(\forall\, Y\ X\ Z.\ \mathit{equal}(f(X::'a,f(Y::'a,Z)),f(f(X::'a,Y),f(X::'a,Z))))$ &
$(\forall\, X\ Y.\ \mathit{left}(X::'a,f(X::'a,Y)))$ &
$(\forall\, Y\ X\ Z.\ \mathit{left}(X::'a,Y)$ & $\mathit{left}(Y::'a,Z) \dashrightarrow \mathit{left}(X::'a,Z))$ &
$(\mathit{equal}(\mathit{num2}::'a,f(\mathit{num1}::'a,\mathit{num1})))$ &
$(\mathit{equal}(\mathit{num3}::'a,f(\mathit{num2}::'a,\mathit{num1})))$ &
$(\mathit{equal}(u::'a,f(\mathit{num2}::'a,\mathit{num2})))$ &
$(\forall\, A\ B\ C.\ \mathit{equal}(A::'a,B) \dashrightarrow \mathit{equal}(f(A::'a,C),f(B::'a,C)))$ &
$(\forall\, D\ F'\ E.\ \mathit{equal}(D::'a,E) \dashrightarrow \mathit{equal}(f(F'::'a,D),f(F'::'a,E)))$ &
$(\forall\, G\ H\ I'.\ \mathit{equal}(G::'a,H)$ & $\mathit{left}(G::'a,I') \dashrightarrow \mathit{left}(H::'a,I'))$ &
$(\forall\, J\ L\ K'.\ \mathit{equal}(J::'a,K')$ & $\mathit{left}(L::'a,J) \dashrightarrow \mathit{left}(L::'a,K'))$ &
$(\sim \mathit{left}(\mathit{num3}::'a,u)) \dashrightarrow \mathit{False}$
⟨*proof*⟩

lemma *MSC002-1*:
$(\mathit{at}(\mathit{something}::'a,\mathit{here},\mathit{now}))$ &
$(\forall\, \mathit{Place}\ \mathit{Situation}.\ \mathit{hand\text{-}at}(\mathit{Place}::'a,\mathit{Situation}) \dashrightarrow \mathit{hand\text{-}at}(\mathit{Place}::'a,\mathit{let\text{-}go}(\mathit{Situation})))$
&
$(\forall\, \mathit{Place}\ \mathit{Another\text{-}place}\ \mathit{Situation}.\ \mathit{hand\text{-}at}(\mathit{Place}::'a,\mathit{Situation}) \dashrightarrow \mathit{hand\text{-}at}(\mathit{Another\text{-}place}::'a,\mathit{go}(\mathit{Another\text{-}pl$
&
$(\forall\, \mathit{Thing}\ \mathit{Situation}.\ \sim \mathit{held}(\mathit{Thing}::'a,\mathit{let\text{-}go}(\mathit{Situation})))$ &
$(\forall\, \mathit{Situation}\ \mathit{Thing}.\ \mathit{at}(\mathit{Thing}::'a,\mathit{here},\mathit{Situation}) \dashrightarrow \mathit{red}(\mathit{Thing}))$ &
$(\forall\, \mathit{Thing}\ \mathit{Place}\ \mathit{Situation}.\ \mathit{at}(\mathit{Thing}::'a,\mathit{Place},\mathit{Situation}) \dashrightarrow \mathit{at}(\mathit{Thing}::'a,\mathit{Place},\mathit{let\text{-}go}(\mathit{Situation})))$
&
$(\forall\, \mathit{Thing}\ \mathit{Place}\ \mathit{Situation}.\ \mathit{at}(\mathit{Thing}::'a,\mathit{Place},\mathit{Situation}) \dashrightarrow \mathit{at}(\mathit{Thing}::'a,\mathit{Place},\mathit{pick\text{-}up}(\mathit{Situation})))$
&
$(\forall\, \mathit{Thing}\ \mathit{Place}\ \mathit{Situation}.\ \mathit{at}(\mathit{Thing}::'a,\mathit{Place},\mathit{Situation}) \dashrightarrow \mathit{grabbed}(\mathit{Thing}::'a,\mathit{pick\text{-}up}(\mathit{go}(\mathit{Place}::'a,\mathit{let\text{-}go}(\mathit{Si$
&
$(\forall\, \mathit{Thing}\ \mathit{Situation}.\ \mathit{red}(\mathit{Thing})$ & $\mathit{put}(\mathit{Thing}::'a,\mathit{there},\mathit{Situation}) \dashrightarrow \mathit{answer}(\mathit{Situation}))$
&
$(\forall\, \mathit{Place}\ \mathit{Thing}\ \mathit{Another\text{-}place}\ \mathit{Situation}.\ \mathit{at}(\mathit{Thing}::'a,\mathit{Place},\mathit{Situation})$ & $\mathit{grabbed}(\mathit{Thing}::'a,\mathit{Situation})$
$\dashrightarrow \mathit{put}(\mathit{Thing}::'a,\mathit{Another\text{-}place},\mathit{go}(\mathit{Another\text{-}place}::'a,\mathit{Situation})))$ &
$(\forall\, \mathit{Thing}\ \mathit{Place}\ \mathit{Another\text{-}place}\ \mathit{Situation}.\ \mathit{at}(\mathit{Thing}::'a,\mathit{Place},\mathit{Situation}) \dashrightarrow \mathit{held}(\mathit{Thing}::'a,\mathit{Situation})$
$\mid \mathit{at}(\mathit{Thing}::'a,\mathit{Place},\mathit{go}(\mathit{Another\text{-}place}::'a,\mathit{Situation})))$ &
$(\forall\, \mathit{One\text{-}place}\ \mathit{Thing}\ \mathit{Place}\ \mathit{Situation}.\ \mathit{hand\text{-}at}(\mathit{One\text{-}place}::'a,\mathit{Situation})$ & $\mathit{held}(\mathit{Thing}::'a,\mathit{Situation})$
$\dashrightarrow \mathit{at}(\mathit{Thing}::'a,\mathit{Place},\mathit{go}(\mathit{Place}::'a,\mathit{Situation})))$ &
$(\forall\, \mathit{Place}\ \mathit{Thing}\ \mathit{Situation}.\ \mathit{hand\text{-}at}(\mathit{Place}::'a,\mathit{Situation})$ & $\mathit{at}(\mathit{Thing}::'a,\mathit{Place},\mathit{Situation})$

$--> held(Thing::'a,pick\text{-}up(Situation)))$ &
$(\forall Situation.\ \sim answer(Situation)) --> False$
$\langle proof \rangle$

lemma *MSC003-1*:
$(\forall Number\text{-}of\text{-}small\text{-}parts\ Small\text{-}part\ Big\text{-}part\ Number\text{-}of\text{-}mid\text{-}parts\ Mid\text{-}part.\ has\text{-}parts(Big\text{-}part::'a,Number\text{-}o$
$--> in'(object\text{-}in(Big\text{-}part::'a,Mid\text{-}part,Small\text{-}part,Number\text{-}of\text{-}mid\text{-}parts,Number\text{-}of\text{-}small\text{-}parts),Mid\text{-}part)$
$| has\text{-}parts(Big\text{-}part::'a,mtimes(Number\text{-}of\text{-}mid\text{-}parts::'a,Number\text{-}of\text{-}small\text{-}parts),Small\text{-}part))$
&
$(\forall Big\text{-}part\ Mid\text{-}part\ Number\text{-}of\text{-}mid\text{-}parts\ Number\text{-}of\text{-}small\text{-}parts\ Small\text{-}part.\ has\text{-}parts(Big\text{-}part::'a,Number\text{-}o$
$\& has\text{-}parts(object\text{-}in(Big\text{-}part::'a,Mid\text{-}part,Small\text{-}part,Number\text{-}of\text{-}mid\text{-}parts,Number\text{-}of\text{-}small\text{-}parts),Number\text{-}$
$--> has\text{-}parts(Big\text{-}part::'a,mtimes(Number\text{-}of\text{-}mid\text{-}parts::'a,Number\text{-}of\text{-}small\text{-}parts),Small\text{-}part))$
&
$(in'(john::'a,boy))$ &
$(\forall X.\ in'(X::'a,boy) --> in'(X::'a,human))$ &
$(\forall X.\ in'(X::'a,hand) --> has\text{-}parts(X::'a,num5,fingers))$ &
$(\forall X.\ in'(X::'a,human) --> has\text{-}parts(X::'a,num2,arm))$ &
$(\forall X.\ in'(X::'a,arm) --> has\text{-}parts(X::'a,num1,hand))$ &
$(\sim has\text{-}parts(john::'a,mtimes(num2::'a,num1),hand)) --> False$
$\langle proof \rangle$

lemma *MSC004-1*:
$(\forall Number\text{-}of\text{-}small\text{-}parts\ Small\text{-}part\ Big\text{-}part\ Number\text{-}of\text{-}mid\text{-}parts\ Mid\text{-}part.\ has\text{-}parts(Big\text{-}part::'a,Number\text{-}of$
$--> in'(object\text{-}in(Big\text{-}part::'a,Mid\text{-}part,Small\text{-}part,Number\text{-}of\text{-}mid\text{-}parts,Number\text{-}of\text{-}small\text{-}parts),Mid\text{-}part)$
$| has\text{-}parts(Big\text{-}part::'a,mtimes(Number\text{-}of\text{-}mid\text{-}parts::'a,Number\text{-}of\text{-}small\text{-}parts),Small\text{-}part))$
&
$(\forall Big\text{-}part\ Mid\text{-}part\ Number\text{-}of\text{-}mid\text{-}parts\ Number\text{-}of\text{-}small\text{-}parts\ Small\text{-}part.\ has\text{-}parts(Big\text{-}part::'a,Number\text{-}o$
$\& has\text{-}parts(object\text{-}in(Big\text{-}part::'a,Mid\text{-}part,Small\text{-}part,Number\text{-}of\text{-}mid\text{-}parts,Number\text{-}of\text{-}small\text{-}parts),Number\text{-}$
$--> has\text{-}parts(Big\text{-}part::'a,mtimes(Number\text{-}of\text{-}mid\text{-}parts::'a,Number\text{-}of\text{-}small\text{-}parts),Small\text{-}part))$
&
$(in'(john::'a,boy))$ &
$(\forall X.\ in'(X::'a,boy) --> in'(X::'a,human))$ &
$(\forall X.\ in'(X::'a,hand) --> has\text{-}parts(X::'a,num5,fingers))$ &
$(\forall X.\ in'(X::'a,human) --> has\text{-}parts(X::'a,num2,arm))$ &
$(\forall X.\ in'(X::'a,arm) --> has\text{-}parts(X::'a,num1,hand))$ &
$(\sim has\text{-}parts(john::'a,mtimes(mtimes(num2::'a,num1),num5),fingers)) --> False$
$\langle proof \rangle$

lemma *MSC005-1*:
$(value(truth::'a,truth))$ &
$(value(falsity::'a,falsity))$ &
$(\forall X\ Y.\ value(X::'a,truth)\ \&\ value(Y::'a,truth) --> value(xor(X::'a,Y),falsity))$
&
$(\forall X\ Y.\ value(X::'a,truth)\ \&\ value(Y::'a,falsity) --> value(xor(X::'a,Y),truth))$
&
$(\forall X\ Y.\ value(X::'a,falsity)\ \&\ value(Y::'a,truth) --> value(xor(X::'a,Y),truth))$

&
(∀ X Y. value(X::'a,falsity) & value(Y::'a,falsity) −−> value(xor(X::'a,Y),falsity))
&
(∀ Value. ~value(xor(xor(xor(xor(truth::'a,falsity),falsity),truth),falsity),Value))
−−> False
⟨proof⟩

lemma *MSC006-1*:
(∀ Y X Z. p(X::'a,Y) & p(Y::'a,Z) −−> p(X::'a,Z)) &
(∀ Y X Z. q(X::'a,Y) & q(Y::'a,Z) −−> q(X::'a,Z)) &
(∀ Y X. q(X::'a,Y) −−> q(Y::'a,X)) &
(∀ X Y. p(X::'a,Y) | q(X::'a,Y)) &
(~p(a::'a,b)) &
(~q(c::'a,d)) −−> False
⟨proof⟩

lemma *NUM001-1*:
(∀ A. equal(A::'a,A)) &
(∀ B A C. equal(A::'a,B) & equal(B::'a,C) −−> equal(A::'a,C)) &
(∀ B A. equal(add(A::'a,B),add(B::'a,A))) &
(∀ A B C. equal(add(A::'a,add(B::'a,C)),add(add(A::'a,B),C))) &
(∀ B A. equal(subtract(add(A::'a,B),B),A)) &
(∀ A B. equal(A::'a,subtract(add(A::'a,B),B))) &
(∀ A C B. equal(add(subtract(A::'a,B),C),subtract(add(A::'a,C),B))) &
(∀ A C B. equal(subtract(add(A::'a,B),C),add(subtract(A::'a,C),B))) &
(∀ A C B D. equal(A::'a,B) & equal(C::'a,add(A::'a,D)) −−> equal(C::'a,add(B::'a,D)))
&
(∀ A C D B. equal(A::'a,B) & equal(C::'a,add(D::'a,A)) −−> equal(C::'a,add(D::'a,B)))
&
(∀ A C B D. equal(A::'a,B) & equal(C::'a,subtract(A::'a,D)) −−> equal(C::'a,subtract(B::'a,D)))
&
(∀ A C D B. equal(A::'a,B) & equal(C::'a,subtract(D::'a,A)) −−> equal(C::'a,subtract(D::'a,B)))
&
(~equal(add(add(a::'a,b),c),add(a::'a,add(b::'a,c)))) −−> False
⟨proof⟩

abbreviation *NUM001-0-ax multiply successor num0 add equal* ≡
(∀ A. equal(add(A::'a,num0),A)) &
(∀ A B. equal(add(A::'a,successor(B)),successor(add(A::'a,B)))) &
(∀ A. equal(multiply(A::'a,num0),num0)) &
(∀ B A. equal(multiply(A::'a,successor(B)),add(multiply(A::'a,B),A))) &
(∀ A B. equal(successor(A),successor(B)) −−> equal(A::'a,B)) &
(∀ A B. equal(A::'a,B) −−> equal(successor(A),successor(B)))

abbreviation *NUM001-1-ax predecessor-of-1st-minus-2nd successor add equal mless*
≡
(∀ A C B. mless(A::'a,B) & mless(C::'a,A) −−> mless(C::'a,B)) &

$(\forall\ A\ B\ C.\ equal(add(successor(A),B),C)\ --{>}\ mless(B::'a,C))\ \&$
$(\forall\ A\ B.\ mless(A::'a,B)\ --{>}\ equal(add(successor(predecessor\text{-}of\text{-}1st\text{-}minus\text{-}2nd(B::'a,A)),A),B))$

abbreviation *NUM001-2-ax equal mless divides* $\equiv$
$(\forall\ A\ B.\ divides(A::'a,B)\ --{>}\ mless(A::'a,B)\ |\ equal(A::'a,B))\ \&$
$(\forall\ A\ B.\ mless(A::'a,B)\ --{>}\ divides(A::'a,B))\ \&$
$(\forall\ A\ B.\ equal(A::'a,B)\ --{>}\ divides(A::'a,B))$

lemma *NUM021-1*:
EQU001-0-ax equal &
NUM001-0-ax multiply successor num0 add equal &
NUM001-1-ax predecessor-of-1st-minus-2nd successor add equal mless &
NUM001-2-ax equal mless divides &
$(mless(b::'a,c))\ \&$
$(\sim mless(b::'a,a))\ \&$
$(divides(c::'a,a))\ \&$
$(\forall\ A.\ \sim equal(successor(A),num0))\ --{>}\ False$
$\langle proof\rangle$

lemma *NUM024-1*:
EQU001-0-ax equal &
NUM001-0-ax multiply successor num0 add equal &
NUM001-1-ax predecessor-of-1st-minus-2nd successor add equal mless &
$(\forall\ B\ A.\ equal(add(A::'a,B),add(B::'a,A)))\ \&$
$(\forall\ B\ A\ C.\ equal(add(A::'a,B),add(C::'a,B))\ --{>}\ equal(A::'a,C))\ \&$
$(mless(a::'a,a))\ \&$
$(\forall\ A.\ \sim equal(successor(A),num0))\ --{>}\ False$
$\langle proof\rangle$

abbreviation *SET004-0-ax not-homomorphism2 not-homomorphism1 homomorphism compatible operation cantor diagonalise subset-relation one-to-one choice apply regular function identity-relation single-valued-class compos powerClass sum-class omega inductive successor-relation successor image' rng domain range-of INVERSE flip rot domain-of null-class restrct difference union complement intersection element-relation second first cross-product ordered-pair singleton unordered-pair equal universal-class not-subclass-element member subclass* $\equiv$
$(\forall\ X\ U\ Y.\ subclass(X::'a,Y)\ \&\ member(U::'a,X)\ --{>}\ member(U::'a,Y))\ \&$
$(\forall\ X\ Y.\ member(not\text{-}subclass\text{-}element(X::'a,Y),X)\ |\ subclass(X::'a,Y))\ \&$
$(\forall\ X\ Y.\ member(not\text{-}subclass\text{-}element(X::'a,Y),Y)\ --{>}\ subclass(X::'a,Y))\ \&$
$(\forall\ X.\ subclass(X::'a,universal\text{-}class))\ \&$
$(\forall\ X\ Y.\ equal(X::'a,Y)\ --{>}\ subclass(X::'a,Y))\ \&$
$(\forall\ Y\ X.\ equal(X::'a,Y)\ --{>}\ subclass(Y::'a,X))\ \&$
$(\forall\ X\ Y.\ subclass(X::'a,Y)\ \&\ subclass(Y::'a,X)\ --{>}\ equal(X::'a,Y))\ \&$
$(\forall\ X\ U\ Y.\ member(U::'a,unordered\text{-}pair(X::'a,Y))\ --{>}\ equal(U::'a,X)\ |\ equal(U::'a,Y))$
&

$(\forall\, X\ Y.\ member(X::'a,universal\text{-}class) \longrightarrow member(X::'a,unordered\text{-}pair(X::'a,Y)))$
&
$(\forall\, X\ Y.\ member(Y::'a,universal\text{-}class) \longrightarrow member(Y::'a,unordered\text{-}pair(X::'a,Y)))$
&
$(\forall\, X\ Y.\ member(unordered\text{-}pair(X::'a,Y),universal\text{-}class))$ &
$(\forall\, X.\ equal(unordered\text{-}pair(X::'a,X),singleton(X)))$ &
$(\forall\, X\ Y.\ equal(unordered\text{-}pair(singleton(X),unordered\text{-}pair(X::'a,singleton(Y))),ordered\text{-}pair(X::'a,Y)))$
&
$(\forall\, V\ Y\ U\ X.\ member(ordered\text{-}pair(U::'a,V),cross\text{-}product(X::'a,Y)) \longrightarrow mem-$
$ber(U::'a,X))$ &
$(\forall\, U\ X\ V\ Y.\ member(ordered\text{-}pair(U::'a,V),cross\text{-}product(X::'a,Y)) \longrightarrow mem-$
$ber(V::'a,Y))$ &
$(\forall\, U\ V\ X\ Y.\ member(U::'a,X)$ & $member(V::'a,Y) \longrightarrow member(ordered\text{-}pair(U::'a,V),cross\text{-}product(X::'a$
&
$(\forall\, X\ Y\ Z.\ member(Z::'a,cross\text{-}product(X::'a,Y)) \longrightarrow equal(ordered\text{-}pair(first(Z),second(Z)),Z))$
&
$(subclass(element\text{-}relation::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))$
&
$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),element\text{-}relation) \longrightarrow member(X::'a,Y))$
&
$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),cross\text{-}product(universal\text{-}class::'a,universal\text{-}class))$
& $member(X::'a,Y) \longrightarrow member(ordered\text{-}pair(X::'a,Y),element\text{-}relation))$ &
$(\forall\, Y\ Z\ X.\ member(Z::'a,intersection(X::'a,Y)) \longrightarrow member(Z::'a,X))$ &
$(\forall\, X\ Z\ Y.\ member(Z::'a,intersection(X::'a,Y)) \longrightarrow member(Z::'a,Y))$ &
$(\forall\, Z\ X\ Y.\ member(Z::'a,X)$ & $member(Z::'a,Y) \longrightarrow member(Z::'a,intersection(X::'a,Y)))$
&
$(\forall\, Z\ X.\ \sim(member(Z::'a,complement(X))$ & $member(Z::'a,X)))$ &
$(\forall\, Z\ X.\ member(Z::'a,universal\text{-}class) \longrightarrow member(Z::'a,complement(X)) \mid$
$member(Z::'a,X))$ &
$(\forall\, X\ Y.\ equal(complement(intersection(complement(X),complement(Y))),union(X::'a,Y)))$
&
$(\forall\, X\ Y.\ equal(intersection(complement(intersection(X::'a,Y)),complement(intersection(complement(X),com$
&
$(\forall\, Xr\ X\ Y.\ equal(intersection(Xr::'a,cross\text{-}product(X::'a,Y)),restrct(Xr::'a,X,Y)))$
&
$(\forall\, Xr\ X\ Y.\ equal(intersection(cross\text{-}product(X::'a,Y),Xr),restrct(Xr::'a,X,Y)))$
&
$(\forall\, Z\ X.\ \sim(equal(restrct(X::'a,singleton(Z),universal\text{-}class),null\text{-}class)$ & $mem-$
$ber(Z::'a,domain\text{-}of(X))))$ &
$(\forall\, Z\ X.\ member(Z::'a,universal\text{-}class) \longrightarrow equal(restrct(X::'a,singleton(Z),universal\text{-}class),null\text{-}class)$
$\mid member(Z::'a,domain\text{-}of(X)))$ &
$(\forall\, X.\ subclass(rot(X),cross\text{-}product(cross\text{-}product(universal\text{-}class::'a,universal\text{-}class),universal\text{-}class)))$
&
$(\forall\, V\ W\ U\ X.\ member(ordered\text{-}pair(ordered\text{-}pair(U::'a,V),W),rot(X)) \longrightarrow mem-$
$ber(ordered\text{-}pair(ordered\text{-}pair(V::'a,W),U),X))$ &
$(\forall\, U\ V\ W\ X.\ member(ordered\text{-}pair(ordered\text{-}pair(V::'a,W),U),X)$ & $member(ordered\text{-}pair(ordered\text{-}pair(U::'a$
$\longrightarrow member(ordered\text{-}pair(ordered\text{-}pair(U::'a,V),W),rot(X)))$ &
$(\forall\, X.\ subclass(flip(X),cross\text{-}product(cross\text{-}product(universal\text{-}class::'a,universal\text{-}class),universal\text{-}class)))$
&

$(\forall\ V\ U\ W\ X.\ member(ordered\text{-}pair(ordered\text{-}pair(U::'a,V),W),flip(X)) \longrightarrow member(ordered\text{-}pair(ordered\text{-}pair(V::'a,U),W),X))\ \&$
$(\forall\ U\ V\ W\ X.\ member(ordered\text{-}pair(ordered\text{-}pair(V::'a,U),W),X)\ \&\ member(ordered\text{-}pair(ordered\text{-}pair(U::'a \longrightarrow member(ordered\text{-}pair(ordered\text{-}pair(U::'a,V),W),flip(X)))\ \&$
$(\forall\ Y.\ equal(domain\text{-}of(flip(cross\text{-}product(Y::'a,universal\text{-}class))),INVERSE(Y)))\ \&$
$(\forall\ Z.\ equal(domain\text{-}of(INVERSE(Z)),range\text{-}of(Z)))\ \&$
$(\forall\ Z\ X\ Y.\ equal(first(not\text{-}subclass\text{-}element(restrct(Z::'a,X,singleton(Y)),null\text{-}class)),domain(Z::'a,X,Y)))\ \&$
$(\forall\ Z\ X\ Y.\ equal(second(not\text{-}subclass\text{-}element(restrct(Z::'a,singleton(X),Y),null\text{-}class)),rng(Z::'a,X,Y)))\ \&$
$(\forall\ Xr\ X.\ equal(range\text{-}of(restrct(Xr::'a,X,universal\text{-}class)),image'(Xr::'a,X)))\ \&$
$(\forall\ X.\ equal(union(X::'a,singleton(X)),successor(X)))\ \&$
$(subclass(successor\text{-}relation::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))\ \&$
$(\forall\ X\ Y.\ member(ordered\text{-}pair(X::'a,Y),successor\text{-}relation) \longrightarrow equal(successor(X),Y))\ \&$
$(\forall\ X\ Y.\ equal(successor(X),Y)\ \&\ member(ordered\text{-}pair(X::'a,Y),cross\text{-}product(universal\text{-}class::'a,universal- \longrightarrow member(ordered\text{-}pair(X::'a,Y),successor\text{-}relation))\ \&$
$(\forall\ X.\ inductive(X) \longrightarrow member(null\text{-}class::'a,X))\ \&$
$(\forall\ X.\ inductive(X) \longrightarrow subclass(image'(successor\text{-}relation::'a,X),X))\ \&$
$(\forall\ X.\ member(null\text{-}class::'a,X)\ \&\ subclass(image'(successor\text{-}relation::'a,X),X) \longrightarrow inductive(X))\ \&$
$(inductive(omega))\ \&$
$(\forall\ Y.\ inductive(Y) \longrightarrow subclass(omega::'a,Y))\ \&$
$(member(omega::'a,universal\text{-}class))\ \&$
$(\forall\ X.\ equal(domain\text{-}of(restrct(element\text{-}relation::'a,universal\text{-}class,X)),sum\text{-}class(X)))\ \&$
$(\forall\ X.\ member(X::'a,universal\text{-}class) \longrightarrow member(sum\text{-}class(X),universal\text{-}class))\ \&$
$(\forall\ X.\ equal(complement(image'(element\text{-}relation::'a,complement(X))),powerClass(X)))\ \&$
$(\forall\ U.\ member(U::'a,universal\text{-}class) \longrightarrow member(powerClass(U),universal\text{-}class))\ \&$
$(\forall\ Yr\ Xr.\ subclass(compos(Yr::'a,Xr),cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))\ \&$
$(\forall\ Z\ Yr\ Xr\ Y.\ member(ordered\text{-}pair(Y::'a,Z),compos(Yr::'a,Xr)) \longrightarrow member(Z::'a,image'(Yr::'a,image'(Xr::'a,singleton(Y))))\ \&$
$(\forall\ Y\ Z\ Yr\ Xr.\ member(Z::'a,image'(Yr::'a,image'(Xr::'a,singleton(Y))))\ \&\ member(ordered\text{-}pair(Y::'a,Z),cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)) \longrightarrow member(ordered\text{-}pair(Y::'a,Z),compos(Yr::'a,Xr)))\ \&$
$(\forall\ X.\ single\text{-}valued\text{-}class(X) \longrightarrow subclass(compos(X::'a,INVERSE(X)),identity\text{-}relation))\ \&$
$(\forall\ X.\ subclass(compos(X::'a,INVERSE(X)),identity\text{-}relation) \longrightarrow single\text{-}valued\text{-}class(X))\ \&$
$(\forall\ Xf.\ function(Xf) \longrightarrow subclass(Xf::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))\ \&$
$(\forall\ Xf.\ function(Xf) \longrightarrow subclass(compos(Xf::'a,INVERSE(Xf)),identity\text{-}relation))\ \&$

$(\forall\, Xf.\ subclass(Xf::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class))\ \&\ sub\text{-}$
$class(compos(Xf::'a,INVERSE(Xf)),identity\text{-}relation)\ \text{--}\!>\ function(Xf))\ \&$
$(\forall\, Xf\ X.\ function(Xf)\ \&\ member(X::'a,universal\text{-}class)\ \text{--}\!>\ member(image'(Xf::'a,X),universal\text{-}class))$
$\&$
$(\forall\, X.\ equal(X::'a,null\text{-}class)\ |\ member(regular(X),X))\ \&$
$(\forall\, X.\ equal(X::'a,null\text{-}class)\ |\ equal(intersection(X::'a,regular(X)),null\text{-}class))\ \&$
$(\forall\, Xf\ Y.\ equal(sum\text{-}class(image'(Xf::'a,singleton(Y))),apply(Xf::'a,Y)))\ \&$
$(function(choice))\ \&$
$(\forall\, Y.\ member(Y::'a,universal\text{-}class)\ \text{--}\!>\ equal(Y::'a,null\text{-}class)\ |\ member(apply(choice::'a,Y),Y))$
$\&$
$(\forall\, Xf.\ one\text{-}to\text{-}one(Xf)\ \text{--}\!>\ function(Xf))\ \&$
$(\forall\, Xf.\ one\text{-}to\text{-}one(Xf)\ \text{--}\!>\ function(INVERSE(Xf)))\ \&$
$(\forall\, Xf.\ function(INVERSE(Xf))\ \&\ function(Xf)\ \text{--}\!>\ one\text{-}to\text{-}one(Xf))\ \&$
$(equal(intersection(cross\text{-}product(universal\text{-}class::'a,universal\text{-}class),intersection(cross\text{-}product(universal\text{-}clas$
$\&$
$(equal(intersection(INVERSE(subset\text{-}relation),subset\text{-}relation),identity\text{-}relation))$
$\&$
$(\forall\, Xr.\ equal(complement(domain\text{-}of(intersection(Xr::'a,identity\text{-}relation))),diagonalise(Xr)))$
$\&$
$(\forall\, X.\ equal(intersection(domain\text{-}of(X),diagonalise(compos(INVERSE(element\text{-}relation),X))),cantor(X)))$
$\&$
$(\forall\, Xf.\ operation(Xf)\ \text{--}\!>\ function(Xf))\ \&$
$(\forall\, Xf.\ operation(Xf)\ \text{--}\!>\ equal(cross\text{-}product(domain\text{-}of(domain\text{-}of(Xf)),domain\text{-}of(domain\text{-}of(Xf))),dom$
$\&$
$(\forall\, Xf.\ operation(Xf)\ \text{--}\!>\ subclass(range\text{-}of(Xf),domain\text{-}of(domain\text{-}of(Xf))))$
$\&$
$(\forall\, Xf.\ function(Xf)\ \&\ equal(cross\text{-}product(domain\text{-}of(domain\text{-}of(Xf)),domain\text{-}of(domain\text{-}of(Xf))),domain\text{-}$
$\&\ subclass(range\text{-}of(Xf),domain\text{-}of(domain\text{-}of(Xf)))\ \text{--}\!>\ operation(Xf))\ \&$
$(\forall\, Xf1\ Xf2\ Xh.\ compatible(Xh::'a,Xf1,Xf2)\ \text{--}\!>\ function(Xh))\ \&$
$(\forall\, Xf2\ Xf1\ Xh.\ compatible(Xh::'a,Xf1,Xf2)\ \text{--}\!>\ equal(domain\text{-}of(domain\text{-}of(Xf1)),domain\text{-}of(Xh)))$
$\&$
$(\forall\, Xf1\ Xh\ Xf2.\ compatible(Xh::'a,Xf1,Xf2)\ \text{--}\!>\ subclass(range\text{-}of(Xh),domain\text{-}of(domain\text{-}of(Xf2))))$
$\&$
$(\forall\, Xh\ Xh1\ Xf1\ Xf2.\ function(Xh)\ \&\ equal(domain\text{-}of(domain\text{-}of(Xf1)),domain\text{-}of(Xh))$
$\&\ subclass(range\text{-}of(Xh),domain\text{-}of(domain\text{-}of(Xf2)))\ \text{--}\!>\ compatible(Xh1::'a,Xf1,Xf2))$
$\&$
$(\forall\, Xh\ Xf2\ Xf1.\ homomorphism(Xh::'a,Xf1,Xf2)\ \text{--}\!>\ operation(Xf1))\ \&$
$(\forall\, Xh\ Xf1\ Xf2.\ homomorphism(Xh::'a,Xf1,Xf2)\ \text{--}\!>\ operation(Xf2))\ \&$
$(\forall\, Xh\ Xf1\ Xf2.\ homomorphism(Xh::'a,Xf1,Xf2)\ \text{--}\!>\ compatible(Xh::'a,Xf1,Xf2))$
$\&$
$(\forall\, Xf2\ Xh\ Xf1\ X\ Y.\ homomorphism(Xh::'a,Xf1,Xf2)\ \&\ member(ordered\text{-}pair(X::'a,Y),domain\text{-}of(Xf1))$
$\text{--}\!>\ equal(apply(Xf2::'a,ordered\text{-}pair(apply(Xh::'a,X),apply(Xh::'a,Y))),apply(Xh::'a,apply(Xf1::'a,ordered\text{-}$
$\&$
$(\forall\, Xh\ Xf1\ Xf2.\ operation(Xf1)\ \&\ operation(Xf2)\ \&\ compatible(Xh::'a,Xf1,Xf2)$
$\text{--}\!>\ member(ordered\text{-}pair(not\text{-}homomorphism1(Xh::'a,Xf1,Xf2),not\text{-}homomorphism2(Xh::'a,Xf1,Xf2)),dom$
$|\ homomorphism(Xh::'a,Xf1,Xf2))\ \&$
$(\forall\, Xh\ Xf1\ Xf2.\ operation(Xf1)\ \&\ operation(Xf2)\ \&\ compatible(Xh::'a,Xf1,Xf2)$
$\&\ equal(apply(Xf2::'a,ordered\text{-}pair(apply(Xh::'a,not\text{-}homomorphism1(Xh::'a,Xf1,Xf2)),apply(Xh::'a,not\text{-}hom$
$\text{--}\!>\ homomorphism(Xh::'a,Xf1,Xf2))$

abbreviation *SET004-0-eq subclass single-valued-class operation one-to-one member inductive homomorphism function compatible unordered-pair union sum-class successor singleton second rot restrct regular range-of rng powerClass ordered-pair not-subclass-element not-homomorphism2 not-homomorphism1 INVERSE intersection image′ flip first domain-of domain difference diagonalise cross-product compos complement cantor apply equal* ≡

$(\forall\, D\ E\ F'.\ equal(D::'a,E) \dashrightarrow equal(apply(D::'a,F'),apply(E::'a,F'))) \&$
$(\forall\, G\ I'\ H.\ equal(G::'a,H) \dashrightarrow equal(apply(I'::'a,G),apply(I'::'a,H))) \&$
$(\forall\, J\ K'.\ equal(J::'a,K') \dashrightarrow equal(cantor(J),cantor(K'))) \&$
$(\forall\, L\ M.\ equal(L::'a,M) \dashrightarrow equal(complement(L),complement(M))) \&$
$(\forall\, N\ O'\ P.\ equal(N::'a,O') \dashrightarrow equal(compos(N::'a,P),compos(O'::'a,P))) \&$
$(\forall\, Q\ S'\ R.\ equal(Q::'a,R) \dashrightarrow equal(compos(S'::'a,Q),compos(S'::'a,R))) \&$
$(\forall\, T'\ U\ V.\ equal(T'::'a,U) \dashrightarrow equal(cross\text{-}product(T'::'a,V),cross\text{-}product(U::'a,V)))$
$\&$
$(\forall\, W\ Y\ X.\ equal(W::'a,X) \dashrightarrow equal(cross\text{-}product(Y::'a,W),cross\text{-}product(Y::'a,X)))$
$\&$
$(\forall\, Z\ A1.\ equal(Z::'a,A1) \dashrightarrow equal(diagonalise(Z),diagonalise(A1))) \&$
$(\forall\, B1\ C1\ D1.\ equal(B1::'a,C1) \dashrightarrow equal(difference(B1::'a,D1),difference(C1::'a,D1)))$
$\&$
$(\forall\, E1\ G1\ F1.\ equal(E1::'a,F1) \dashrightarrow equal(difference(G1::'a,E1),difference(G1::'a,F1)))$
$\&$
$(\forall\, H1\ I1\ J1\ K1.\ equal(H1::'a,I1) \dashrightarrow equal(domain(H1::'a,J1,K1),domain(I1::'a,J1,K1)))$
$\&$
$(\forall\, L1\ N1\ M1\ O1.\ equal(L1::'a,M1) \dashrightarrow equal(domain(N1::'a,L1,O1),domain(N1::'a,M1,O1)))$
$\&$
$(\forall\, P1\ R1\ S1\ Q1.\ equal(P1::'a,Q1) \dashrightarrow equal(domain(R1::'a,S1,P1),domain(R1::'a,S1,Q1)))$
$\&$
$(\forall\, T1\ U1.\ equal(T1::'a,U1) \dashrightarrow equal(domain\text{-}of(T1),domain\text{-}of(U1))) \&$
$(\forall\, V1\ W1.\ equal(V1::'a,W1) \dashrightarrow equal(first(V1),first(W1))) \&$
$(\forall\, X1\ Y1.\ equal(X1::'a,Y1) \dashrightarrow equal(flip(X1),flip(Y1))) \&$
$(\forall\, Z1\ A2\ B2.\ equal(Z1::'a,A2) \dashrightarrow equal(image'(Z1::'a,B2),image'(A2::'a,B2)))$
$\&$
$(\forall\, C2\ E2\ D2.\ equal(C2::'a,D2) \dashrightarrow equal(image'(E2::'a,C2),image'(E2::'a,D2)))$
$\&$
$(\forall\, F2\ G2\ H2.\ equal(F2::'a,G2) \dashrightarrow equal(intersection(F2::'a,H2),intersection(G2::'a,H2)))$
$\&$
$(\forall\, I2\ K2\ J2.\ equal(I2::'a,J2) \dashrightarrow equal(intersection(K2::'a,I2),intersection(K2::'a,J2)))$
$\&$
$(\forall\, L2\ M2.\ equal(L2::'a,M2) \dashrightarrow equal(INVERSE(L2),INVERSE(M2))) \&$
$(\forall\, N2\ O2\ P2\ Q2.\ equal(N2::'a,O2) \dashrightarrow equal(not\text{-}homomorphism1(N2::'a,P2,Q2),not\text{-}homomorphism1(O2$
$\&$
$(\forall\, R2\ T2\ S2\ U2.\ equal(R2::'a,S2) \dashrightarrow equal(not\text{-}homomorphism1(T2::'a,R2,U2),not\text{-}homomorphism1(T2:$
$\&$
$(\forall\, V2\ X2\ Y2\ W2.\ equal(V2::'a,W2) \dashrightarrow equal(not\text{-}homomorphism1(X2::'a,Y2,V2),not\text{-}homomorphism1(X$
$\&$
$(\forall\, Z2\ A3\ B3\ C3.\ equal(Z2::'a,A3) \dashrightarrow equal(not\text{-}homomorphism2(Z2::'a,B3,C3),not\text{-}homomorphism2(A3:$
$\&$

(∀ D3 F3 E3 G3. equal(D3::'a,E3) −−> equal(not-homomorphism2(F3::'a,D3,G3),not-homomorphism2(F3
&
(∀ H3 J3 K3 I3. equal(H3::'a,I3) −−> equal(not-homomorphism2(J3::'a,K3,H3),not-homomorphism2(J3::'
&
(∀ L3 M3 N3. equal(L3::'a,M3) −−> equal(not-subclass-element(L3::'a,N3),not-subclass-element(M3::'a,N3
&
(∀ O3 Q3 P3. equal(O3::'a,P3) −−> equal(not-subclass-element(Q3::'a,O3),not-subclass-element(Q3::'a,P3
&
(∀ R3 S3 T3. equal(R3::'a,S3) −−> equal(ordered-pair(R3::'a,T3),ordered-pair(S3::'a,T3)))
&
(∀ U3 W3 V3. equal(U3::'a,V3) −−> equal(ordered-pair(W3::'a,U3),ordered-pair(W3::'a,V3)))
&
(∀ X3 Y3. equal(X3::'a,Y3) −−> equal(powerClass(X3),powerClass(Y3))) &
(∀ Z3 A4 B4 C4. equal(Z3::'a,A4) −−> equal(rng(Z3::'a,B4,C4),rng(A4::'a,B4,C4)))
&
(∀ D4 F4 E4 G4. equal(D4::'a,E4) −−> equal(rng(F4::'a,D4,G4),rng(F4::'a,E4,G4)))
&
(∀ H4 J4 K4 I4. equal(H4::'a,I4) −−> equal(rng(J4::'a,K4,H4),rng(J4::'a,K4,I4)))
&
(∀ L4 M4. equal(L4::'a,M4) −−> equal(range-of(L4),range-of(M4))) &
(∀ N4 O4. equal(N4::'a,O4) −−> equal(regular(N4),regular(O4))) &
(∀ P4 Q4 R4 S4. equal(P4::'a,Q4) −−> equal(restrct(P4::'a,R4,S4),restrct(Q4::'a,R4,S4)))
&
(∀ T4 V4 U4 W4. equal(T4::'a,U4) −−> equal(restrct(V4::'a,T4,W4),restrct(V4::'a,U4,W4)))
&
(∀ X4 Z4 A5 Y4. equal(X4::'a,Y4) −−> equal(restrct(Z4::'a,A5,X4),restrct(Z4::'a,A5,Y4)))
&
(∀ B5 C5. equal(B5::'a,C5) −−> equal(rot(B5),rot(C5))) &
(∀ D5 E5. equal(D5::'a,E5) −−> equal(second(D5),second(E5))) &
(∀ F5 G5. equal(F5::'a,G5) −−> equal(singleton(F5),singleton(G5))) &
(∀ H5 I5. equal(H5::'a,I5) −−> equal(successor(H5),successor(I5))) &
(∀ J5 K5. equal(J5::'a,K5) −−> equal(sum-class(J5),sum-class(K5))) &
(∀ L5 M5 N5. equal(L5::'a,M5) −−> equal(union(L5::'a,N5),union(M5::'a,N5)))
&
(∀ O5 Q5 P5. equal(O5::'a,P5) −−> equal(union(Q5::'a,O5),union(Q5::'a,P5)))
&
(∀ R5 S5 T5. equal(R5::'a,S5) −−> equal(unordered-pair(R5::'a,T5),unordered-pair(S5::'a,T5)))
&
(∀ U5 W5 V5. equal(U5::'a,V5) −−> equal(unordered-pair(W5::'a,U5),unordered-pair(W5::'a,V5)))
&
(∀ X5 Y5 Z5 A6. equal(X5::'a,Y5) & compatible(X5::'a,Z5,A6) −−> compati-
ble(Y5::'a,Z5,A6)) &
(∀ B6 D6 C6 E6. equal(B6::'a,C6) & compatible(D6::'a,B6,E6) −−> compati-
ble(D6::'a,C6,E6)) &
(∀ F6 H6 I6 G6. equal(F6::'a,G6) & compatible(H6::'a,I6,F6) −−> compati-
ble(H6::'a,I6,G6)) &
(∀ J6 K6. equal(J6::'a,K6) & function(J6) −−> function(K6)) &
(∀ L6 M6 N6 O6. equal(L6::'a,M6) & homomorphism(L6::'a,N6,O6) −−> ho-
momorphism(M6::'a,N6,O6)) &

(∀ P6 R6 Q6 S6. equal(P6::'a,Q6) & homomorphism(R6::'a,P6,S6) −−> homomorphism(R6::'a,Q6,S6)) &
(∀ T6 V6 W6 U6. equal(T6::'a,U6) & homomorphism(V6::'a,W6,T6) −−> homomorphism(V6::'a,W6,U6)) &
(∀ X6 Y6. equal(X6::'a,Y6) & inductive(X6) −−> inductive(Y6)) &
(∀ Z6 A7 B7. equal(Z6::'a,A7) & member(Z6::'a,B7) −−> member(A7::'a,B7))
&
(∀ C7 E7 D7. equal(C7::'a,D7) & member(E7::'a,C7) −−> member(E7::'a,D7))
&
(∀ F7 G7. equal(F7::'a,G7) & one-to-one(F7) −−> one-to-one(G7)) &
(∀ H7 I7. equal(H7::'a,I7) & operation(H7) −−> operation(I7)) &
(∀ J7 K7. equal(J7::'a,K7) & single-valued-class(J7) −−> single-valued-class(K7))
&
(∀ L7 M7 N7. equal(L7::'a,M7) & subclass(L7::'a,N7) −−> subclass(M7::'a,N7))
&
(∀ O7 Q7 P7. equal(O7::'a,P7) & subclass(Q7::'a,O7) −−> subclass(Q7::'a,P7))

abbreviation *SET004-1-ax range-of function maps apply*
application-function singleton-relation element-relation complement
intersection single-valued3 singleton image' domain single-valued2
second single-valued1 identity-relation INVERSE not-subclass-element
first domain-of domain-relation composition-function compos equal
ordered-pair member universal-class cross-product compose-class
subclass ≡
(∀ X. subclass(compose-class(X),cross-product(universal-class::'a,universal-class)))
&
(∀ X Y Z. member(ordered-pair(Y::'a,Z),compose-class(X)) −−> equal(compos(X::'a,Y),Z))
&
(∀ Y Z X. member(ordered-pair(Y::'a,Z),cross-product(universal-class::'a,universal-class))
& equal(compos(X::'a,Y),Z) −−> member(ordered-pair(Y::'a,Z),compose-class(X)))
&
(subclass(composition-function::'a,cross-product(universal-class::'a,cross-product(universal-class::'a,universa
&
(∀ X Y Z. member(ordered-pair(X::'a,ordered-pair(Y::'a,Z)),composition-function)
−−> equal(compos(X::'a,Y),Z)) &
(∀ X Y. member(ordered-pair(X::'a,Y),cross-product(universal-class::'a,universal-class))
−−> member(ordered-pair(X::'a,ordered-pair(Y::'a,compos(X::'a,Y))),composition-function))
&
(subclass(domain-relation::'a,cross-product(universal-class::'a,universal-class))) &
(∀ X Y. member(ordered-pair(X::'a,Y),domain-relation) −−> equal(domain-of(X),Y))
&
(∀ X. member(X::'a,universal-class) −−> member(ordered-pair(X::'a,domain-of(X)),domain-relation))
&
(∀ X. equal(first(not-subclass-element(compos(X::'a,INVERSE(X)),identity-relation)),single-valued1(X)))
&
(∀ X. equal(second(not-subclass-element(compos(X::'a,INVERSE(X)),identity-relation)),single-valued2(X))
&
(∀ X. equal(domain(X::'a,image'(INVERSE(X),singleton(single-valued1(X))),single-valued2(X)),single-valu
&

(*equal*(*intersection*(*complement*(*compos*(*element-relation*::'*a*,*complement*(*identity-relation*))),*element-relation*
&
(*subclass*(*application-function*::'*a*,*cross-product*(*universal-class*::'*a*,*cross-product*(*universal-class*::'*a*,*universal-*
&
(∀ *Z Y X*. *member*(*ordered-pair*(*X*::'*a*,*ordered-pair*(*Y*::'*a*,*Z*)),*application-function*)
−−> *member*(*Y*::'*a*,*domain-of*(*X*))) &
(∀ *X Y Z*. *member*(*ordered-pair*(*X*::'*a*,*ordered-pair*(*Y*::'*a*,*Z*)),*application-function*)
−−> *equal*(*apply*(*X*::'*a*,*Y*),*Z*)) &
(∀ *Z X Y*. *member*(*ordered-pair*(*X*::'*a*,*ordered-pair*(*Y*::'*a*,*Z*)),*cross-product*(*universal-class*::'*a*,*cross-product*(
& *member*(*Y*::'*a*,*domain-of*(*X*)) −−> *member*(*ordered-pair*(*X*::'*a*,*ordered-pair*(*Y*::'*a*,*apply*(*X*::'*a*,*Y*))),*applic*
&
(∀ *X Y Xf*. *maps*(*Xf*::'*a*,*X*,*Y*) −−> *function*(*Xf*)) &
(∀ *Y Xf X*. *maps*(*Xf*::'*a*,*X*,*Y*) −−> *equal*(*domain-of*(*Xf*),*X*)) &
(∀ *X Xf Y*. *maps*(*Xf*::'*a*,*X*,*Y*) −−> *subclass*(*range-of*(*Xf*),*Y*)) &
(∀ *Xf Y*. *function*(*Xf*) & *subclass*(*range-of*(*Xf*),*Y*) −−> *maps*(*Xf*::'*a*,*domain-of*(*Xf*),*Y*))

abbreviation *SET004-1-eq maps single-valued3 single-valued2 single-valued1 compose-class*
equal ≡
(∀ *L M*. *equal*(*L*::'*a*,*M*) −−> *equal*(*compose-class*(*L*),*compose-class*(*M*))) &
(∀ *N2 O2*. *equal*(*N2*::'*a*,*O2*) −−> *equal*(*single-valued1*(*N2*),*single-valued1*(*O2*)))
&
(∀ *P2 Q2*. *equal*(*P2*::'*a*,*Q2*) −−> *equal*(*single-valued2*(*P2*),*single-valued2*(*Q2*)))
&
(∀ *R2 S2*. *equal*(*R2*::'*a*,*S2*) −−> *equal*(*single-valued3*(*R2*),*single-valued3*(*S2*)))
&
(∀ *X2 Y2 Z2 A3*. *equal*(*X2*::'*a*,*Y2*) & *maps*(*X2*::'*a*,*Z2*,*A3*) −−> *maps*(*Y2*::'*a*,*Z2*,*A3*))
&
(∀ *B3 D3 C3 E3*. *equal*(*B3*::'*a*,*C3*) & *maps*(*D3*::'*a*,*B3*,*E3*) −−> *maps*(*D3*::'*a*,*C3*,*E3*))
&
(∀ *F3 H3 I3 G3*. *equal*(*F3*::'*a*,*G3*) & *maps*(*H3*::'*a*,*I3*,*F3*) −−> *maps*(*H3*::'*a*,*I3*,*G3*))

abbreviation *NUM004-0-ax integer-of omega ordinal-multiply*
add-relation ordinal-add recursion apply range-of union-of-range-map
function recursion-equation-functions rest-relation rest-of
limit-ordinals kind-1-ordinals successor-relation image'
universal-class sum-class element-relation ordinal-numbers section
not-well-ordering ordered-pair least member well-ordering singleton
domain-of segment null-class intersection asymmetric compos transitive
cross-product connected identity-relation complement restrct subclass
irreflexive symmetrization-of INVERSE union equal ≡
(∀ *X*. *equal*(*union*(*X*::'*a*,*INVERSE*(*X*)),*symmetrization-of*(*X*))) &
(∀ *X Y*. *irreflexive*(*X*::'*a*,*Y*) −−> *subclass*(*restrct*(*X*::'*a*,*Y*,*Y*),*complement*(*identity-relation*)))
&
(∀ *X Y*. *subclass*(*restrct*(*X*::'*a*,*Y*,*Y*),*complement*(*identity-relation*)) −−> *irreflex-*
ive(*X*::'*a*,*Y*)) &
(∀ *Y X*. *connected*(*X*::'*a*,*Y*) −−> *subclass*(*cross-product*(*Y*::'*a*,*Y*),*union*(*identity-relation*::'*a*,*symmetrizatio*
&
(∀ *X Y*. *subclass*(*cross-product*(*Y*::'*a*,*Y*),*union*(*identity-relation*::'*a*,*symmetrization-of*(*X*)))
−−> *connected*(*X*::'*a*,*Y*)) &

$(\forall\, Xr\ Y.\ transitive(Xr::'a,Y) \longrightarrow subclass(compos(restrct(Xr::'a,Y,Y),restrct(Xr::'a,Y,Y)),restrct(Xr::'a$
$\&$
$(\forall\, Xr\ Y.\ subclass(compos(restrct(Xr::'a,Y,Y),restrct(Xr::'a,Y,Y)),restrct(Xr::'a,Y,Y))$
$\longrightarrow transitive(Xr::'a,Y))\ \&$
$(\forall\, Xr\ Y.\ asymmetric(Xr::'a,Y) \longrightarrow equal(restrct(intersection(Xr::'a,INVERSE(Xr)),Y,Y),null\text{-}class))$
$\&$
$(\forall\, Xr\ Y.\ equal(restrct(intersection(Xr::'a,INVERSE(Xr)),Y,Y),null\text{-}class) \longrightarrow$
$asymmetric(Xr::'a,Y))\ \&$
$(\forall\, Xr\ Y\ Z.\ equal(segment(Xr::'a,Y,Z),domain\text{-}of(restrct(Xr::'a,Y,singleton(Z)))))$
$\&$
$(\forall\, X\ Y.\ well\text{-}ordering(X::'a,Y) \longrightarrow connected(X::'a,Y))\ \&$
$(\forall\, Y\ Xr\ U.\ well\text{-}ordering(Xr::'a,Y)\ \&\ subclass(U::'a,Y) \longrightarrow equal(U::'a,null\text{-}class)$
$\mid member(least(Xr::'a,U),U))\ \&$
$(\forall\, Y\ V\ Xr\ U.\ well\text{-}ordering(Xr::'a,Y)\ \&\ subclass(U::'a,Y)\ \&\ member(V::'a,U)$
$\longrightarrow member(least(Xr::'a,U),U))\ \&$
$(\forall\, Y\ Xr\ U.\ well\text{-}ordering(Xr::'a,Y)\ \&\ subclass(U::'a,Y) \longrightarrow equal(segment(Xr::'a,U,least(Xr::'a,U)),null\text{-}$
$\&$
$(\forall\, Y\ V\ U\ Xr.\ \sim(well\text{-}ordering(Xr::'a,Y)\ \&\ subclass(U::'a,Y)\ \&\ member(V::'a,U)$
$\&\ member(ordered\text{-}pair(V::'a,least(Xr::'a,U)),Xr)))\ \&$
$(\forall\, Xr\ Y.\ connected(Xr::'a,Y)\ \&\ equal(not\text{-}well\text{-}ordering(Xr::'a,Y),null\text{-}class)$
$\longrightarrow well\text{-}ordering(Xr::'a,Y))\ \&$
$(\forall\, Xr\ Y.\ connected(Xr::'a,Y) \longrightarrow subclass(not\text{-}well\text{-}ordering(Xr::'a,Y),Y)\ \mid$
$well\text{-}ordering(Xr::'a,Y))\ \&$
$(\forall\, V\ Xr\ Y.\ member(V::'a,not\text{-}well\text{-}ordering(Xr::'a,Y))\ \&\ equal(segment(Xr::'a,not\text{-}well\text{-}ordering(Xr::'a,Y),$
$\&\ connected(Xr::'a,Y) \longrightarrow well\text{-}ordering(Xr::'a,Y))\ \&$
$(\forall\, Xr\ Y\ Z.\ section(Xr::'a,Y,Z) \longrightarrow subclass(Y::'a,Z))\ \&$
$(\forall\, Xr\ Z\ Y.\ section(Xr::'a,Y,Z) \longrightarrow subclass(domain\text{-}of(restrct(Xr::'a,Z,Y)),Y))$
$\&$
$(\forall\, Xr\ Y\ Z.\ subclass(Y::'a,Z)\ \&\ subclass(domain\text{-}of(restrct(Xr::'a,Z,Y)),Y) \longrightarrow$
$section(Xr::'a,Y,Z))\ \&$
$(\forall\, X.\ member(X::'a,ordinal\text{-}numbers) \longrightarrow well\text{-}ordering(element\text{-}relation::'a,X))$
$\&$
$(\forall\, X.\ member(X::'a,ordinal\text{-}numbers) \longrightarrow subclass(sum\text{-}class(X),X))\ \&$
$(\forall\, X.\ well\text{-}ordering(element\text{-}relation::'a,X)\ \&\ subclass(sum\text{-}class(X),X)\ \&\ mem\text{-}$
$ber(X::'a,universal\text{-}class) \longrightarrow member(X::'a,ordinal\text{-}numbers))\ \&$
$(\forall\, X.\ well\text{-}ordering(element\text{-}relation::'a,X)\ \&\ subclass(sum\text{-}class(X),X) \longrightarrow$
$member(X::'a,ordinal\text{-}numbers)\ \mid\ equal(X::'a,ordinal\text{-}numbers))\ \&$
$(equal(union(singleton(null\text{-}class),image'(successor\text{-}relation::'a,ordinal\text{-}numbers)),kind\text{-}1\text{-}ordinals))$
$\&$
$(equal(intersection(complement(kind\text{-}1\text{-}ordinals),ordinal\text{-}numbers),limit\text{-}ordinals))$
$\&$
$(\forall\, X.\ subclass(rest\text{-}of(X),cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))\ \&$
$(\forall\, V\ U\ X.\ member(ordered\text{-}pair(U::'a,V),rest\text{-}of(X)) \longrightarrow member(U::'a,domain\text{-}of(X)))$
$\&$
$(\forall\, X\ U\ V.\ member(ordered\text{-}pair(U::'a,V),rest\text{-}of(X)) \longrightarrow equal(restrct(X::'a,U,universal\text{-}class),V))$
$\&$
$(\forall\, U\ V\ X.\ member(U::'a,domain\text{-}of(X))\ \&\ equal(restrct(X::'a,U,universal\text{-}class),V)$
$\longrightarrow member(ordered\text{-}pair(U::'a,V),rest\text{-}of(X)))\ \&$
$(subclass(rest\text{-}relation::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))\ \&$

$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),rest\text{-}relation) \dashrightarrow equal(rest\text{-}of(X),Y))$
&
$(\forall\, X.\ member(X::'a,universal\text{-}class) \dashrightarrow member(ordered\text{-}pair(X::'a,rest\text{-}of(X)),rest\text{-}relation))$
&
$(\forall\, X\ Z.\ member(X::'a,recursion\text{-}equation\text{-}functions(Z)) \dashrightarrow function(Z))$ &
$(\forall\, Z\ X.\ member(X::'a,recursion\text{-}equation\text{-}functions(Z)) \dashrightarrow function(X))$ &
$(\forall\, Z\ X.\ member(X::'a,recursion\text{-}equation\text{-}functions(Z)) \dashrightarrow member(domain\text{-}of(X),ordinal\text{-}numbers))$
&
$(\forall\, Z\ X.\ member(X::'a,recursion\text{-}equation\text{-}functions(Z)) \dashrightarrow equal(compos(Z::'a,rest\text{-}of(X)),X))$
&
$(\forall\, X\ Z.\ function(Z)$ & $function(X)$ & $member(domain\text{-}of(X),ordinal\text{-}numbers)$ &
$equal(compos(Z::'a,rest\text{-}of(X)),X) \dashrightarrow member(X::'a,recursion\text{-}equation\text{-}functions(Z)))$
&
$(subclass(union\text{-}of\text{-}range\text{-}map::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))$
&
$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),union\text{-}of\text{-}range\text{-}map) \dashrightarrow equal(sum\text{-}class(range\text{-}of(X)),Y))$
&
$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),cross\text{-}product(universal\text{-}class::'a,universal\text{-}class))$
& $equal(sum\text{-}class(range\text{-}of(X)),Y) \dashrightarrow member(ordered\text{-}pair(X::'a,Y),union\text{-}of\text{-}range\text{-}map))$
&
$(\forall\, X\ Y.\ equal(apply(recursion(X::'a,successor\text{-}relation,union\text{-}of\text{-}range\text{-}map),Y),ordinal\text{-}add(X::'a,Y)))$
&
$(\forall\, X\ Y.\ equal(recursion(null\text{-}class::'a,apply(add\text{-}relation::'a,X),union\text{-}of\text{-}range\text{-}map),ordinal\text{-}multiply(X::'a,$
&
$(\forall\, X.\ member(X::'a,omega) \dashrightarrow equal(integer\text{-}of(X),X))$ &
$(\forall\, X.\ member(X::'a,omega) \mid equal(integer\text{-}of(X),null\text{-}class))$

abbreviation *NUM004-0-eq well-ordering transitive section irreflexive connected asymmetric symmetrization-of segment rest-of recursion-equation-functions recursion ordinal-multiply ordinal-add not-well-ordering least integer-of equal* $\equiv$
$(\forall\, D\ E.\ equal(D::'a,E) \dashrightarrow equal(integer\text{-}of(D),integer\text{-}of(E)))$ &
$(\forall\, F'\ G\ H.\ equal(F'::'a,G) \dashrightarrow equal(least(F'::'a,H),least(G::'a,H)))$ &
$(\forall\, I'\ K'\ J.\ equal(I'::'a,J) \dashrightarrow equal(least(K'::'a,I'),least(K'::'a,J)))$ &
$(\forall\, L\ M\ N.\ equal(L::'a,M) \dashrightarrow equal(not\text{-}well\text{-}ordering(L::'a,N),not\text{-}well\text{-}ordering(M::'a,N)))$
&
$(\forall\, O'\ Q\ P.\ equal(O'::'a,P) \dashrightarrow equal(not\text{-}well\text{-}ordering(Q::'a,O'),not\text{-}well\text{-}ordering(Q::'a,P)))$
&
$(\forall\, R\ S'\ T'.\ equal(R::'a,S') \dashrightarrow equal(ordinal\text{-}add(R::'a,T'),ordinal\text{-}add(S'::'a,T')))$
&
$(\forall\, U\ W\ V.\ equal(U::'a,V) \dashrightarrow equal(ordinal\text{-}add(W::'a,U),ordinal\text{-}add(W::'a,V)))$
&
$(\forall\, X\ Y\ Z.\ equal(X::'a,Y) \dashrightarrow equal(ordinal\text{-}multiply(X::'a,Z),ordinal\text{-}multiply(Y::'a,Z)))$
&
$(\forall\, A1\ C1\ B1.\ equal(A1::'a,B1) \dashrightarrow equal(ordinal\text{-}multiply(C1::'a,A1),ordinal\text{-}multiply(C1::'a,B1)))$
&
$(\forall\, F1\ G1\ H1\ I1.\ equal(F1::'a,G1) \dashrightarrow equal(recursion(F1::'a,H1,I1),recursion(G1::'a,H1,I1)))$
&
$(\forall\, J1\ L1\ K1\ M1.\ equal(J1::'a,K1) \dashrightarrow equal(recursion(L1::'a,J1,M1),recursion(L1::'a,K1,M1)))$

$$
\begin{array}{l}
\& \\
\quad (\forall\, N1\ P1\ Q1\ O1.\ equal(N1::'a,O1) \longrightarrow equal(recursion(P1::'a,Q1,N1),recursion(P1::'a,Q1,O1))) \\
\& \\
\quad (\forall\, R1\ S1.\ equal(R1::'a,S1) \longrightarrow equal(recursion\text{-}equation\text{-}functions(R1),recursion\text{-}equation\text{-}functions(S1))) \\
\& \\
\quad (\forall\, T1\ U1.\ equal(T1::'a,U1) \longrightarrow equal(rest\text{-}of(T1),rest\text{-}of(U1))) \ \& \\
\quad (\forall\, V1\ W1\ X1\ Y1.\ equal(V1::'a,W1) \longrightarrow equal(segment(V1::'a,X1,Y1),segment(W1::'a,X1,Y1))) \\
\& \\
\quad (\forall\, Z1\ B2\ A2\ C2.\ equal(Z1::'a,A2) \longrightarrow equal(segment(B2::'a,Z1,C2),segment(B2::'a,A2,C2))) \\
\& \\
\quad (\forall\, D2\ F2\ G2\ E2.\ equal(D2::'a,E2) \longrightarrow equal(segment(F2::'a,G2,D2),segment(F2::'a,G2,E2))) \\
\& \\
\quad (\forall\, H2\ I2.\ equal(H2::'a,I2) \longrightarrow equal(symmetrization\text{-}of(H2),symmetrization\text{-}of(I2))) \\
\& \\
\quad (\forall\, J2\ K2\ L2.\ equal(J2::'a,K2)\ \&\ asymmetric(J2::'a,L2) \longrightarrow asymmetric(K2::'a,L2)) \\
\& \\
\quad (\forall\, M2\ O2\ N2.\ equal(M2::'a,N2)\ \&\ asymmetric(O2::'a,M2) \longrightarrow asymmetric(O2::'a,N2)) \\
\& \\
\quad (\forall\, P2\ Q2\ R2.\ equal(P2::'a,Q2)\ \&\ connected(P2::'a,R2) \longrightarrow connected(Q2::'a,R2)) \\
\& \\
\quad (\forall\, S2\ U2\ T2.\ equal(S2::'a,T2)\ \&\ connected(U2::'a,S2) \longrightarrow connected(U2::'a,T2)) \\
\& \\
\quad (\forall\, V2\ W2\ X2.\ equal(V2::'a,W2)\ \&\ irreflexive(V2::'a,X2) \longrightarrow irreflexive(W2::'a,X2)) \\
\& \\
\quad (\forall\, Y2\ A3\ Z2.\ equal(Y2::'a,Z2)\ \&\ irreflexive(A3::'a,Y2) \longrightarrow irreflexive(A3::'a,Z2)) \\
\& \\
\quad (\forall\, B3\ C3\ D3\ E3.\ equal(B3::'a,C3)\ \&\ section(B3::'a,D3,E3) \longrightarrow section(C3::'a,D3,E3)) \\
\& \\
\quad (\forall\, F3\ H3\ G3\ I3.\ equal(F3::'a,G3)\ \&\ section(H3::'a,F3,I3) \longrightarrow section(H3::'a,G3,I3)) \\
\& \\
\quad (\forall\, J3\ L3\ M3\ K3.\ equal(J3::'a,K3)\ \&\ section(L3::'a,M3,J3) \longrightarrow section(L3::'a,M3,K3)) \\
\& \\
\quad (\forall\, N3\ O3\ P3.\ equal(N3::'a,O3)\ \&\ transitive(N3::'a,P3) \longrightarrow transitive(O3::'a,P3)) \\
\& \\
\quad (\forall\, Q3\ S3\ R3.\ equal(Q3::'a,R3)\ \&\ transitive(S3::'a,Q3) \longrightarrow transitive(S3::'a,R3)) \\
\& \\
\quad (\forall\, T3\ U3\ V3.\ equal(T3::'a,U3)\ \&\ well\text{-}ordering(T3::'a,V3) \longrightarrow well\text{-}ordering(U3::'a,V3)) \\
\& \\
\quad (\forall\, W3\ Y3\ X3.\ equal(W3::'a,X3)\ \&\ well\text{-}ordering(Y3::'a,W3) \longrightarrow well\text{-}ordering(Y3::'a,X3))
\end{array}
$$

lemma *NUM180-1*:
EQU001-0-ax equal &
SET004-0-ax not-homomorphism2 not-homomorphism1
homomorphism compatible operation cantor diagonalise subset-relation
one-to-one choice apply regular function identity-relation
single-valued-class compos powerClass sum-class omega inductive
successor-relation successor image' rng domain range-of INVERSE flip
rot domain-of null-class restrct difference union complement

intersection element-relation second first cross-product ordered-pair
singleton unordered-pair equal universal-class not-subclass-element
member subclass &
SET004-0-eq subclass single-valued-class operation
one-to-one member inductive homomorphism function compatible
unordered-pair union sum-class successor singleton second rot restrct
regular range-of rng powerClass ordered-pair not-subclass-element
not-homomorphism2 not-homomorphism1 INVERSE intersection image' flip
first domain-of domain difference diagonalise cross-product compos
complement cantor apply equal &
SET004-1-ax range-of function maps apply
application-function singleton-relation element-relation complement
intersection single-valued3 singleton image' domain single-valued2
second single-valued1 identity-relation INVERSE not-subclass-element
first domain-of domain-relation composition-function compos equal
ordered-pair member universal-class cross-product compose-class
subclass &
SET004-1-eq maps single-valued3 single-valued2 single-valued1 compose-class equal &
NUM004-0-ax integer-of omega ordinal-multiply
add-relation ordinal-add recursion apply range-of union-of-range-map
function recursion-equation-functions rest-relation rest-of
limit-ordinals kind-1-ordinals successor-relation image'
universal-class sum-class element-relation ordinal-numbers section
not-well-ordering ordered-pair least member well-ordering singleton
domain-of segment null-class intersection asymmetric compos transitive
cross-product connected identity-relation complement restrct subclass
irreflexive symmetrization-of INVERSE union equal &
NUM004-0-eq well-ordering transitive section irreflexive
connected asymmetric symmetrization-of segment rest-of
recursion-equation-functions recursion ordinal-multiply ordinal-add
not-well-ordering least integer-of equal &
$(\sim subclass(limit\text{-}ordinals::'a, ordinal\text{-}numbers)) \longrightarrow False$
⟨*proof*⟩

lemma *NUM228-1*:
EQU001-0-ax equal &
SET004-0-ax not-homomorphism2 not-homomorphism1
homomorphism compatible operation cantor diagonalise subset-relation
one-to-one choice apply regular function identity-relation
single-valued-class compos powerClass sum-class omega inductive
successor-relation successor image' rng domain range-of INVERSE flip
rot domain-of null-class restrct difference union complement
intersection element-relation second first cross-product ordered-pair
singleton unordered-pair equal universal-class not-subclass-element
member subclass &
SET004-0-eq subclass single-valued-class operation

one-to-one member inductive homomorphism function compatible
unordered-pair union sum-class successor singleton second rot restrct
regular range-of rng powerClass ordered-pair not-subclass-element
not-homomorphism2 not-homomorphism1 INVERSE intersection image' flip
first domain-of domain difference diagonalise cross-product compos
complement cantor apply equal &
SET004-1-ax range-of function maps apply
application-function singleton-relation element-relation complement
intersection single-valued3 singleton image' domain single-valued2
second single-valued1 identity-relation INVERSE not-subclass-element
first domain-of domain-relation composition-function compos equal
ordered-pair member universal-class cross-product compose-class
subclass &
SET004-1-eq maps single-valued3 single-valued2 single-valued1 compose-class equal
&
NUM004-0-ax integer-of omega ordinal-multiply
add-relation ordinal-add recursion apply range-of union-of-range-map
function recursion-equation-functions rest-relation rest-of
limit-ordinals kind-1-ordinals successor-relation image'
universal-class sum-class element-relation ordinal-numbers section
not-well-ordering ordered-pair least member well-ordering singleton
domain-of segment null-class intersection asymmetric compos transitive
cross-product connected identity-relation complement restrct subclass
irreflexive symmetrization-of INVERSE union equal &
NUM004-0-eq well-ordering transitive section irreflexive
connected asymmetric symmetrization-of segment rest-of
recursion-equation-functions recursion ordinal-multiply ordinal-add
not-well-ordering least integer-of equal &
$(\sim function(z))$ &
$(\sim equal(recursion\text{-}equation\text{-}functions(z), null\text{-}class))$ $--> False$
⟨*proof*⟩

lemma *PLA002-1*:
$(\forall Situation1\ Situation2.\ warm(Situation1) \mid cold(Situation2))$ &
$(\forall Situation.\ at(a::'a, Situation) --> at(b::'a, walk(b::'a, Situation)))$ &
$(\forall Situation.\ at(a::'a, Situation) --> at(b::'a, drive(b::'a, Situation)))$ &
$(\forall Situation.\ at(b::'a, Situation) --> at(a::'a, walk(a::'a, Situation)))$ &
$(\forall Situation.\ at(b::'a, Situation) --> at(a::'a, drive(a::'a, Situation)))$ &
$(\forall Situation.\ cold(Situation)\ \&\ at(b::'a, Situation) --> at(c::'a, skate(c::'a, Situation)))$
&
$(\forall Situation.\ cold(Situation)\ \&\ at(c::'a, Situation) --> at(b::'a, skate(b::'a, Situation)))$
&
$(\forall Situation.\ warm(Situation)\ \&\ at(b::'a, Situation) --> at(d::'a, climb(d::'a, Situation)))$
&
$(\forall Situation.\ warm(Situation)\ \&\ at(d::'a, Situation) --> at(b::'a, climb(b::'a, Situation)))$
&
$(\forall Situation.\ at(c::'a, Situation) --> at(d::'a, go(d::'a, Situation)))$ &

$(\forall\, Situation.\ at(d::'a,Situation) \longrightarrow at(c::'a,go(c::'a,Situation)))$ &
$(\forall\, Situation.\ at(c::'a,Situation) \longrightarrow at(e::'a,go(e::'a,Situation)))$ &
$(\forall\, Situation.\ at(e::'a,Situation) \longrightarrow at(c::'a,go(c::'a,Situation)))$ &
$(\forall\, Situation.\ at(d::'a,Situation) \longrightarrow at(f::'a,go(f::'a,Situation)))$ &
$(\forall\, Situation.\ at(f::'a,Situation) \longrightarrow at(d::'a,go(d::'a,Situation)))$ &
$(at(f::'a,s0))$ &
$(\forall\, S'.\ \sim at(a::'a,S')) \longrightarrow False$
$\langle proof \rangle$

abbreviation *PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds* $\equiv$
$(\forall\, X\ Y\ State.\ holds(X::'a,State)\ \&\ holds(Y::'a,State) \longrightarrow holds(and'(X::'a,Y),State))$
&
$(\forall\, State\ X.\ holds(EMPTY::'a,State)\ \&\ holds(clear(X),State)\ \&\ differ(X::'a,table)$
$\longrightarrow holds(holding(X),do(pickup(X),State)))$ &
$(\forall\, Y\ X\ State.\ holds(on(X::'a,Y),State)\ \&\ holds(clear(X),State)\ \&\ holds(EMPTY::'a,State)$
$\longrightarrow holds(clear(Y),do(pickup(X),State)))$ &
$(\forall\, Y\ State\ X\ Z.\ holds(on(X::'a,Y),State)\ \&\ differ(X::'a,Z) \longrightarrow holds(on(X::'a,Y),do(pickup(Z),State)))$
&
$(\forall\, State\ X\ Z.\ holds(clear(X),State)\ \&\ differ(X::'a,Z) \longrightarrow holds(clear(X),do(pickup(Z),State)))$
&
$(\forall\, X\ Y\ State.\ holds(holding(X),State)\ \&\ holds(clear(Y),State) \longrightarrow holds(EMPTY::'a,do(putdown(X::'a,Y$
&
$(\forall\, X\ Y\ State.\ holds(holding(X),State)\ \&\ holds(clear(Y),State) \longrightarrow holds(on(X::'a,Y),do(putdown(X::'a,Y$
&
$(\forall\, X\ Y\ State.\ holds(holding(X),State)\ \&\ holds(clear(Y),State) \longrightarrow holds(clear(X),do(putdown(X::'a,Y),St$
&
$(\forall\, Z\ W\ X\ Y\ State.\ holds(on(X::'a,Y),State) \longrightarrow holds(on(X::'a,Y),do(putdown(Z::'a,W),State)))$
&
$(\forall\, X\ State\ Z\ Y.\ holds(clear(Z),State)\ \&\ differ(Z::'a,Y) \longrightarrow holds(clear(Z),do(putdown(X::'a,Y),State)))$

abbreviation *PLA001-1-ax EMPTY clear s0 on holds table d c b a differ* $\equiv$
$(\forall\, Y\ X.\ differ(Y::'a,X) \longrightarrow differ(X::'a,Y))$ &
$(differ(a::'a,b))$ &
$(differ(a::'a,c))$ &
$(differ(a::'a,d))$ &
$(differ(a::'a,table))$ &
$(differ(b::'a,c))$ &
$(differ(b::'a,d))$ &
$(differ(b::'a,table))$ &
$(differ(c::'a,d))$ &
$(differ(c::'a,table))$ &
$(differ(d::'a,table))$ &
$(holds(on(a::'a,table),s0))$ &
$(holds(on(b::'a,table),s0))$ &
$(holds(on(c::'a,d),s0))$ &
$(holds(on(d::'a,table),s0))$ &
$(holds(clear(a),s0))$ &
$(holds(clear(b),s0))$ &

(*holds*(*clear*(*c*),*s0*)) &
(*holds*(*EMPTY*::'*a*,*s0*)) &
(∀ *State*. *holds*(*clear*(*table*),*State*))

lemma *PLA006-1*:
PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
&
PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
(∀ *State*. ~*holds*(*on*(*c*::'*a*,*table*),*State*)) −−> *False*
⟨*proof*⟩

lemma *PLA017-1*:
PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
&
PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
(∀ *State*. ~*holds*(*on*(*a*::'*a*,*c*),*State*)) −−> *False*
⟨*proof*⟩

lemma *PLA022-1*:
PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
&
PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
(∀ *State*. ~*holds*(*and'*(*on*(*c*::'*a*,*d*),*on*(*a*::'*a*,*c*)),*State*)) −−> *False*
⟨*proof*⟩

lemma *PLA022-2*:
PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
&
PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
(∀ *State*. ~*holds*(*and'*(*on*(*a*::'*a*,*c*),*on*(*c*::'*a*,*d*)),*State*)) −−> *False*
⟨*proof*⟩

lemma *PRV001-1*:
(∀ *X Y Z*. *q1*(*X*::'*a*,*Y*,*Z*) & *mless-or-equal*(*X*::'*a*,*Y*) −−> *q2*(*X*::'*a*,*Y*,*Z*)) &
(∀ *X Y Z*. *q1*(*X*::'*a*,*Y*,*Z*) −−> *mless-or-equal*(*X*::'*a*,*Y*) | *q3*(*X*::'*a*,*Y*,*Z*)) &
(∀ *Z X Y*. *q2*(*X*::'*a*,*Y*,*Z*) −−> *q4*(*X*::'*a*,*Y*,*Y*)) &
(∀ *Z Y X*. *q3*(*X*::'*a*,*Y*,*Z*) −−> *q4*(*X*::'*a*,*Y*,*X*)) &
(∀ *X*. *mless-or-equal*(*X*::'*a*,*X*)) &
(∀ *X Y*. *mless-or-equal*(*X*::'*a*,*Y*) & *mless-or-equal*(*Y*::'*a*,*X*) −−> *equal*(*X*::'*a*,*Y*))
&
(∀ *Y X Z*. *mless-or-equal*(*X*::'*a*,*Y*) & *mless-or-equal*(*Y*::'*a*,*Z*) −−> *mless-or-equal*(*X*::'*a*,*Z*))
&
(∀ *Y X*. *mless-or-equal*(*X*::'*a*,*Y*) | *mless-or-equal*(*Y*::'*a*,*X*)) &
(∀ *X Y*. *equal*(*X*::'*a*,*Y*) −−> *mless-or-equal*(*X*::'*a*,*Y*)) &

(∀ X Y Z. equal(X::'a,Y) & mless-or-equal(X::'a,Z) −−> mless-or-equal(Y::'a,Z))
&
(∀ X Z Y. equal(X::'a,Y) & mless-or-equal(Z::'a,X) −−> mless-or-equal(Z::'a,Y))
&
(q1(a::'a,b,c)) &
(∀ W. ~(q4(a::'a,b,W) & mless-or-equal(a::'a,W) & mless-or-equal(b::'a,W) &
mless-or-equal(W::'a,a))) &
(∀ W. ~(q4(a::'a,b,W) & mless-or-equal(a::'a,W) & mless-or-equal(b::'a,W) &
mless-or-equal(W::'a,b))) −−> False
⟨proof⟩

abbreviation SWV001-1-ax mless-THAN successor predecessor equal ≡
(∀ X. equal(predecessor(successor(X)),X)) &
(∀ X. equal(successor(predecessor(X)),X)) &
(∀ X Y. equal(predecessor(X),predecessor(Y)) −−> equal(X::'a,Y)) &
(∀ X Y. equal(successor(X),successor(Y)) −−> equal(X::'a,Y)) &
(∀ X. mless-THAN(predecessor(X),X)) &
(∀ X. mless-THAN(X::'a,successor(X))) &
(∀ X Y Z. mless-THAN(X::'a,Y) & mless-THAN(Y::'a,Z) −−> mless-THAN(X::'a,Z))
&
(∀ X Y. mless-THAN(X::'a,Y) | mless-THAN(Y::'a,X) | equal(X::'a,Y)) &
(∀ X. ~mless-THAN(X::'a,X)) &
(∀ Y X. ~(mless-THAN(X::'a,Y) & mless-THAN(Y::'a,X))) &
(∀ Y X Z. equal(X::'a,Y) & mless-THAN(X::'a,Z) −−> mless-THAN(Y::'a,Z))
&
(∀ Y Z X. equal(X::'a,Y) & mless-THAN(Z::'a,X) −−> mless-THAN(Z::'a,Y))

abbreviation SWV001-0-eq a successor predecessor equal ≡
(∀ X Y. equal(X::'a,Y) −−> equal(predecessor(X),predecessor(Y))) &
(∀ X Y. equal(X::'a,Y) −−> equal(successor(X),successor(Y))) &
(∀ X Y. equal(X::'a,Y) −−> equal(a(X),a(Y)))

lemma PRV003-1:
EQU001-0-ax equal &
SWV001-1-ax mless-THAN successor predecessor equal &
SWV001-0-eq a successor predecessor equal &
(~mless-THAN(n::'a,j)) &
(mless-THAN(k::'a,j)) &
(~mless-THAN(k::'a,i)) &
(mless-THAN(i::'a,n)) &
(mless-THAN(a(j),a(k))) &
(∀ X. mless-THAN(X::'a,j) & mless-THAN(a(X),a(k)) −−> mless-THAN(X::'a,i))
&
(∀ X. mless-THAN(One::'a,i) & mless-THAN(a(X),a(predecessor(i))) −−> mless-THAN(X::'a,i)
| mless-THAN(n::'a,X)) &
(∀ X. ~(mless-THAN(One::'a,X) & mless-THAN(X::'a,i) & mless-THAN(a(X),a(predecessor(X)))))
&

(*mless-THAN*(*j*::'*a*,*i*)) −−> *False*
⟨*proof*⟩

lemma *PRV005-1*:
EQU001-0-ax equal &
SWV001-1-ax mless-THAN successor predecessor equal &
SWV001-0-eq a successor predecessor equal &
(~*mless-THAN*(*n*::'*a*,*k*)) &
(~*mless-THAN*(*k*::'*a*,*l*)) &
(~*mless-THAN*(*k*::'*a*,*i*)) &
(*mless-THAN*(*l*::'*a*,*n*)) &
(*mless-THAN*(*One*::'*a*,*l*)) &
(*mless-THAN*(*a*(*k*),*a*(*predecessor*(*l*)))) &
(∀ *X*. *mless-THAN*(*X*::'*a*,*successor*(*n*)) & *mless-THAN*(*a*(*X*),*a*(*k*)) −−> *mless-THAN*(*X*::'*a*,*l*))
&
(∀ *X*. *mless-THAN*(*One*::'*a*,*l*) & *mless-THAN*(*a*(*X*),*a*(*predecessor*(*l*))) −−> *mless-THAN*(*X*::'*a*,*l*)
| *mless-THAN*(*n*::'*a*,*X*)) &
(∀ *X*. ~(*mless-THAN*(*One*::'*a*,*X*) & *mless-THAN*(*X*::'*a*,*l*) & *mless-THAN*(*a*(*X*),*a*(*predecessor*(*X*)))))
−−> *False*
⟨*proof*⟩

lemma *PRV006-1*:
EQU001-0-ax equal &
SWV001-1-ax mless-THAN successor predecessor equal &
SWV001-0-eq a successor predecessor equal &
(~*mless-THAN*(*n*::'*a*,*m*)) &
(*mless-THAN*(*i*::'*a*,*m*)) &
(*mless-THAN*(*i*::'*a*,*n*)) &
(~*mless-THAN*(*i*::'*a*,*One*)) &
(*mless-THAN*(*a*(*i*),*a*(*m*))) &
(∀ *X*. *mless-THAN*(*X*::'*a*,*successor*(*n*)) & *mless-THAN*(*a*(*X*),*a*(*m*)) −−> *mless-THAN*(*X*::'*a*,*i*))
&
(∀ *X*. *mless-THAN*(*One*::'*a*,*i*) & *mless-THAN*(*a*(*X*),*a*(*predecessor*(*i*))) −−> *mless-THAN*(*X*::'*a*,*i*)
| *mless-THAN*(*n*::'*a*,*X*)) &
(∀ *X*. ~(*mless-THAN*(*One*::'*a*,*X*) & *mless-THAN*(*X*::'*a*,*i*) & *mless-THAN*(*a*(*X*),*a*(*predecessor*(*X*)))))
−−> *False*
⟨*proof*⟩

lemma *PRV009-1*:
(∀ *Y X*. *mless-or-equal*(*X*::'*a*, *Y*) | *mless*(*Y*::'*a*,*X*)) &
(*mless*(*j*::'*a*,*i*)) &
(*mless-or-equal*(*m*::'*a*,*p*)) &
(*mless-or-equal*(*p*::'*a*,*q*)) &
(*mless-or-equal*(*q*::'*a*,*n*)) &
(∀ *X Y*. *mless-or-equal*(*m*::'*a*,*X*) & *mless*(*X*::'*a*,*i*) & *mless*(*j*::'*a*, *Y*) & *mless-or-equal*(*Y*::'*a*,*n*)
−−> *mless-or-equal*(*a*(*X*),*a*(*Y*))) &

$(\forall X\ Y.\ mless\text{-}or\text{-}equal(m::'a,X)\ \&\ mless\text{-}or\text{-}equal(X::'a,Y)\ \&\ mless\text{-}or\text{-}equal(Y::'a,j)$
$--> mless\text{-}or\text{-}equal(a(X),a(Y)))\ \&$
$(\forall X\ Y.\ mless\text{-}or\text{-}equal(i::'a,X)\ \&\ mless\text{-}or\text{-}equal(X::'a,Y)\ \&\ mless\text{-}or\text{-}equal(Y::'a,n)$
$--> mless\text{-}or\text{-}equal(a(X),a(Y)))\ \&$
$(\sim mless\text{-}or\text{-}equal(a(p),a(q)))\ --> False$
$\langle proof \rangle$

lemma *PUZ012-1*:
$(\forall X.\ equal\text{-}fruits(X::'a,X))\ \&$
$(\forall X.\ equal\text{-}boxes(X::'a,X))\ \&$
$(\forall X\ Y.\ \sim(label(X::'a,Y)\ \&\ contains(X::'a,Y)))\ \&$
$(\forall X.\ contains(boxa::'a,X) \mid contains(boxb::'a,X) \mid contains(boxc::'a,X))\ \&$
$(\forall X.\ contains(X::'a,apples) \mid contains(X::'a,bananas) \mid contains(X::'a,oranges))$
$\&$
$(\forall X\ Y\ Z.\ contains(X::'a,Y)\ \&\ contains(X::'a,Z) --> equal\text{-}fruits(Y::'a,Z))\ \&$
$(\forall Y\ X\ Z.\ contains(X::'a,Y)\ \&\ contains(Z::'a,Y) --> equal\text{-}boxes(X::'a,Z))\ \&$
$(\sim equal\text{-}boxes(boxa::'a,boxb))\ \&$
$(\sim equal\text{-}boxes(boxb::'a,boxc))\ \&$
$(\sim equal\text{-}boxes(boxa::'a,boxc))\ \&$
$(\sim equal\text{-}fruits(apples::'a,bananas))\ \&$
$(\sim equal\text{-}fruits(bananas::'a,oranges))\ \&$
$(\sim equal\text{-}fruits(apples::'a,oranges))\ \&$
$(label(boxa::'a,apples))\ \&$
$(label(boxb::'a,oranges))\ \&$
$(label(boxc::'a,bananas))\ \&$
$(contains(boxb::'a,apples))\ \&$
$(\sim(contains(boxa::'a,bananas)\ \&\ contains(boxc::'a,oranges)))\ --> False$
$\langle proof \rangle$

lemma *PUZ020-1*:
$EQU001\text{-}0\text{-}ax\ equal\ \&$
$(\forall A\ B.\ equal(A::'a,B) --> equal(statement\text{-}by(A),statement\text{-}by(B)))\ \&$
$(\forall X.\ person(X) --> knight(X) \mid knave(X))\ \&$
$(\forall X.\ \sim(person(X)\ \&\ knight(X)\ \&\ knave(X)))\ \&$
$(\forall X\ Y.\ says(X::'a,Y)\ \&\ a\text{-}truth(Y) --> a\text{-}truth(Y))\ \&$
$(\forall X\ Y.\ \sim(says(X::'a,Y)\ \&\ equal(X::'a,Y)))\ \&$
$(\forall Y\ X.\ says(X::'a,Y) --> equal(Y::'a,statement\text{-}by(X)))\ \&$
$(\forall X\ Y.\ \sim(person(X)\ \&\ equal(X::'a,statement\text{-}by(Y))))\ \&$
$(\forall X.\ person(X)\ \&\ a\text{-}truth(statement\text{-}by(X)) --> knight(X))\ \&$
$(\forall X.\ person(X) --> a\text{-}truth(statement\text{-}by(X)) \mid knave(X))\ \&$
$(\forall X\ Y.\ equal(X::'a,Y)\ \&\ knight(X) --> knight(Y))\ \&$
$(\forall X\ Y.\ equal(X::'a,Y)\ \&\ knave(X) --> knave(Y))\ \&$
$(\forall X\ Y.\ equal(X::'a,Y)\ \&\ person(X) --> person(Y))\ \&$
$(\forall X\ Y\ Z.\ equal(X::'a,Y)\ \&\ says(X::'a,Z) --> says(Y::'a,Z))\ \&$
$(\forall X\ Z\ Y.\ equal(X::'a,Y)\ \&\ says(Z::'a,X) --> says(Z::'a,Y))\ \&$
$(\forall X\ Y.\ equal(X::'a,Y)\ \&\ a\text{-}truth(X) --> a\text{-}truth(Y))\ \&$
$(\forall X\ Y.\ knight(X)\ \&\ says(X::'a,Y) --> a\text{-}truth(Y))\ \&$

$(\forall\, X\ Y.\ \sim(knave(X)\ \&\ says(X::'a,Y)\ \&\ a\text{-}truth(Y)))\ \&$
$(person(husband))\ \&$
$(person(wife))\ \&$
$(\sim equal(husband::'a,wife))\ \&$
$(says(husband::'a,statement\text{-}by(husband)))\ \&$
$(a\text{-}truth(statement\text{-}by(husband))\ \&\ knight(husband)\ \text{--}\!>\ knight(wife))\ \&$
$(knight(husband)\ \text{--}\!>\ a\text{-}truth(statement\text{-}by(husband)))\ \&$
$(a\text{-}truth(statement\text{-}by(husband))\ |\ knight(wife))\ \&$
$(knight(wife)\ \text{--}\!>\ a\text{-}truth(statement\text{-}by(husband)))\ \&$
$(\sim knight(husband))\ \text{--}\!>\ False$
$\langle proof \rangle$

lemma *PUZ025-1*:
$(\forall\, X.\ a\text{-}truth(truthteller(X))\ |\ a\text{-}truth(liar(X)))\ \&$
$(\forall\, X.\ \sim(a\text{-}truth(truthteller(X))\ \&\ a\text{-}truth(liar(X))))\ \&$
$(\forall\, Truthteller\ Statement.\ a\text{-}truth(truthteller(Truthteller))\ \&\ a\text{-}truth(says(Truthteller::'a,Statement))$
$\text{--}\!>\ a\text{-}truth(Statement))\ \&$
$(\forall\, Liar\ Statement.\ \sim(a\text{-}truth(liar(Liar))\ \&\ a\text{-}truth(says(Liar::'a,Statement))\ \&$
$a\text{-}truth(Statement)))\ \&$
$(\forall\, Statement\ Truthteller.\ a\text{-}truth(Statement)\ \&\ a\text{-}truth(says(Truthteller::'a,Statement))$
$\text{--}\!>\ a\text{-}truth(truthteller(Truthteller)))\ \&$
$(\forall\, Statement\ Liar.\ a\text{-}truth(says(Liar::'a,Statement))\ \text{--}\!>\ a\text{-}truth(Statement)\ |$
$a\text{-}truth(liar(Liar)))\ \&$
$(\forall\, Z\ X\ Y.\ people(X::'a,Y,Z)\ \&\ a\text{-}truth(liar(X))\ \&\ a\text{-}truth(liar(Y))\ \text{--}\!>\ a\text{-}truth(equal\text{-}type(X::'a,Y)))$
$\&$
$(\forall\, Z\ X\ Y.\ people(X::'a,Y,Z)\ \&\ a\text{-}truth(truthteller(X))\ \&\ a\text{-}truth(truthteller(Y))$
$\text{--}\!>\ a\text{-}truth(equal\text{-}type(X::'a,Y)))\ \&$
$(\forall\, X\ Y.\ a\text{-}truth(equal\text{-}type(X::'a,Y))\ \&\ a\text{-}truth(truthteller(X))\ \text{--}\!>\ a\text{-}truth(truthteller(Y)))$
$\&$
$(\forall\, X\ Y.\ a\text{-}truth(equal\text{-}type(X::'a,Y))\ \&\ a\text{-}truth(liar(X))\ \text{--}\!>\ a\text{-}truth(liar(Y)))$
$\&$
$(\forall\, X\ Y.\ a\text{-}truth(truthteller(X))\ \text{--}\!>\ a\text{-}truth(equal\text{-}type(X::'a,Y))\ |\ a\text{-}truth(liar(Y)))$
$\&$
$(\forall\, X\ Y.\ a\text{-}truth(liar(X))\ \text{--}\!>\ a\text{-}truth(equal\text{-}type(X::'a,Y))\ |\ a\text{-}truth(truthteller(Y)))$
$\&$
$(\forall\, Y\ X.\ a\text{-}truth(equal\text{-}type(X::'a,Y))\ \text{--}\!>\ a\text{-}truth(equal\text{-}type(Y::'a,X)))\ \&$
$(\forall\, X\ Y.\ ask\text{-}1\text{-}if\text{-}2(X::'a,Y)\ \&\ a\text{-}truth(truthteller(X))\ \&\ a\text{-}truth(Y)\ \text{--}\!>\ an\text{-}$
$swer(yes))\ \&$
$(\forall\, X\ Y.\ ask\text{-}1\text{-}if\text{-}2(X::'a,Y)\ \&\ a\text{-}truth(truthteller(X))\ \text{--}\!>\ a\text{-}truth(Y)\ |\ an\text{-}$
$swer(no))\ \&$
$(\forall\, X\ Y.\ ask\text{-}1\text{-}if\text{-}2(X::'a,Y)\ \&\ a\text{-}truth(liar(X))\ \&\ a\text{-}truth(Y)\ \text{--}\!>\ answer(no))$
$\&$
$(\forall\, X\ Y.\ ask\text{-}1\text{-}if\text{-}2(X::'a,Y)\ \&\ a\text{-}truth(liar(X))\ \text{--}\!>\ a\text{-}truth(Y)\ |\ answer(yes))$
$\&$
$(people(b::'a,c,a))\ \&$
$(people(a::'a,b,a))\ \&$
$(people(a::'a,c,b))\ \&$
$(people(c::'a,b,a))\ \&$

(*a-truth*(*says*(*a*::'*a*,*equal-type*(*b*::'*a*,*c*)))) &
(*ask-1-if-2*(*c*::'*a*,*equal-type*(*a*::'*a*,*b*))) &
(∀ *Answer*. ~*answer*(*Answer*)) −−> *False*
⟨*proof*⟩

lemma *PUZ029-1*:
(∀ *X*. *dances-on-tightropes*(*X*) | *eats-pennybuns*(*X*) | *old*(*X*)) &
(∀ *X*. *pig*(*X*) & *liable-to-giddiness*(*X*) −−> *treated-with-respect*(*X*)) &
(∀ *X*. *wise*(*X*) & *balloonist*(*X*) −−> *has-umbrella*(*X*)) &
(∀ *X*. ~(*looks-ridiculous*(*X*) & *eats-pennybuns*(*X*) & *eats-lunch-in-public*(*X*))) &
(∀ *X*. *balloonist*(*X*) & *young*(*X*) −−> *liable-to-giddiness*(*X*)) &
(∀ *X*. *fat*(*X*) & *looks-ridiculous*(*X*) −−> *dances-on-tightropes*(*X*) | *eats-lunch-in-public*(*X*))
&
(∀ *X*. ~(*liable-to-giddiness*(*X*) & *wise*(*X*) & *dances-on-tightropes*(*X*))) &
(∀ *X*. *pig*(*X*) & *has-umbrella*(*X*) −−> *looks-ridiculous*(*X*)) &
(∀ *X*. *treated-with-respect*(*X*) −−> *dances-on-tightropes*(*X*) | *fat*(*X*)) &
(∀ *X*. *young*(*X*) | *old*(*X*)) &
(∀ *X*. ~(*young*(*X*) & *old*(*X*))) &
(*wise*(*piggy*)) &
(*young*(*piggy*)) &
(*pig*(*piggy*)) &
(*balloonist*(*piggy*)) −−> *False*
⟨*proof*⟩

abbreviation *RNG001-0-ax equal additive-inverse add multiply product additive-identity sum* ≡
(∀ *X*. *sum*(*additive-identity*::'*a*,*X*,*X*)) &
(∀ *X*. *sum*(*X*::'*a*,*additive-identity*,*X*)) &
(∀ *X Y*. *product*(*X*::'*a*,*Y*,*multiply*(*X*::'*a*,*Y*))) &
(∀ *X Y*. *sum*(*X*::'*a*,*Y*,*add*(*X*::'*a*,*Y*))) &
(∀ *X*. *sum*(*additive-inverse*(*X*),*X*,*additive-identity*)) &
(∀ *X*. *sum*(*X*::'*a*,*additive-inverse*(*X*),*additive-identity*)) &
(∀ *Y U Z X V W*. *sum*(*X*::'*a*,*Y*,*U*) & *sum*(*Y*::'*a*,*Z*,*V*) & *sum*(*U*::'*a*,*Z*,*W*) −−>
sum(*X*::'*a*,*V*,*W*)) &
(∀ *Y X V U Z W*. *sum*(*X*::'*a*,*Y*,*U*) & *sum*(*Y*::'*a*,*Z*,*V*) & *sum*(*X*::'*a*,*V*,*W*) −−>
sum(*U*::'*a*,*Z*,*W*)) &
(∀ *Y X Z*. *sum*(*X*::'*a*,*Y*,*Z*) −−> *sum*(*Y*::'*a*,*X*,*Z*)) &
(∀ *Y U Z X V W*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*Y*::'*a*,*Z*,*V*) & *product*(*U*::'*a*,*Z*,*W*)
−−> *product*(*X*::'*a*,*V*,*W*)) &
(∀ *Y X V U Z W*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*Y*::'*a*,*Z*,*V*) & *product*(*X*::'*a*,*V*,*W*)
−−> *product*(*U*::'*a*,*Z*,*W*)) &
(∀ *Y Z X V3 V1 V2 V4*. *product*(*X*::'*a*,*Y*,*V1*) & *product*(*X*::'*a*,*Z*,*V2*) & *sum*(*Y*::'*a*,*Z*,*V3*)
& *product*(*X*::'*a*,*V3*,*V4*) −−> *sum*(*V1*::'*a*,*V2*,*V4*)) &
(∀ *Y Z V1 V2 X V3 V4*. *product*(*X*::'*a*,*Y*,*V1*) & *product*(*X*::'*a*,*Z*,*V2*) & *sum*(*Y*::'*a*,*Z*,*V3*)
& *sum*(*V1*::'*a*,*V2*,*V4*) −−> *product*(*X*::'*a*,*V3*,*V4*)) &
(∀ *Y Z V3 X V1 V2 V4*. *product*(*Y*::'*a*,*X*,*V1*) & *product*(*Z*::'*a*,*X*,*V2*) & *sum*(*Y*::'*a*,*Z*,*V3*)
& *product*(*V3*::'*a*,*X*,*V4*) −−> *sum*(*V1*::'*a*,*V2*,*V4*)) &

(∀ Y Z V1 V2 V3 X V4. product(Y::'a,X,V1) & product(Z::'a,X,V2) & sum(Y::'a,Z,V3) & sum(V1::'a,V2,V4) −−> product(V3::'a,X,V4)) &
(∀ X Y U V. sum(X::'a,Y,U) & sum(X::'a,Y,V) −−> equal(U::'a,V)) &
(∀ X Y U V. product(X::'a,Y,U) & product(X::'a,Y,V) −−> equal(U::'a,V))

abbreviation *RNG001-0-eq product multiply sum add additive-inverse equal* ≡
(∀ X Y. equal(X::'a,Y) −−> equal(additive-inverse(X),additive-inverse(Y))) &
(∀ X Y W. equal(X::'a,Y) −−> equal(add(X::'a,W),add(Y::'a,W))) &
(∀ X W Y. equal(X::'a,Y) −−> equal(add(W::'a,X),add(W::'a,Y))) &
(∀ X Y W Z. equal(X::'a,Y) & sum(X::'a,W,Z) −−> sum(Y::'a,W,Z)) &
(∀ X W Y Z. equal(X::'a,Y) & sum(W::'a,X,Z) −−> sum(W::'a,Y,Z)) &
(∀ X W Z Y. equal(X::'a,Y) & sum(W::'a,Z,X) −−> sum(W::'a,Z,Y)) &
(∀ X Y W. equal(X::'a,Y) −−> equal(multiply(X::'a,W),multiply(Y::'a,W)))
&
(∀ X W Y. equal(X::'a,Y) −−> equal(multiply(W::'a,X),multiply(W::'a,Y)))
&
(∀ X Y W Z. equal(X::'a,Y) & product(X::'a,W,Z) −−> product(Y::'a,W,Z))
&
(∀ X W Y Z. equal(X::'a,Y) & product(W::'a,X,Z) −−> product(W::'a,Y,Z))
&
(∀ X W Z Y. equal(X::'a,Y) & product(W::'a,Z,X) −−> product(W::'a,Z,Y))

lemma *RNG001-3*:
(∀ X. sum(additive-identity::'a,X,X)) &
(∀ X. sum(additive-inverse(X),X,additive-identity)) &
(∀ Y U Z X V W. sum(X::'a,Y,U) & sum(Y::'a,Z,V) & sum(U::'a,Z,W) −−>
sum(X::'a,V,W)) &
(∀ Y X V U Z W. sum(X::'a,Y,U) & sum(Y::'a,Z,V) & sum(X::'a,V,W) −−>
sum(U::'a,Z,W)) &
(∀ X Y. product(X::'a,Y,multiply(X::'a,Y))) &
(∀ Y Z X V3 V1 V2 V4. product(X::'a,Y,V1) & product(X::'a,Z,V2) & sum(Y::'a,Z,V3)
& product(X::'a,V3,V4) −−> sum(V1::'a,V2,V4)) &
(∀ Y Z V1 V2 X V3 V4. product(X::'a,Y,V1) & product(X::'a,Z,V2) & sum(Y::'a,Z,V3)
& sum(V1::'a,V2,V4) −−> product(X::'a,V3,V4)) &
(~product(a::'a,additive-identity,additive-identity)) −−> False
⟨*proof*⟩

abbreviation *RNG-other-ax multiply add equal product additive-identity additive-inverse sum* ≡
(∀ X. sum(X::'a,additive-inverse(X),additive-identity)) &
(∀ Y U Z X V W. sum(X::'a,Y,U) & sum(Y::'a,Z,V) & sum(U::'a,Z,W) −−>
sum(X::'a,V,W)) &
(∀ Y X V U Z W. sum(X::'a,Y,U) & sum(Y::'a,Z,V) & sum(X::'a,V,W) −−>
sum(U::'a,Z,W)) &
(∀ Y X Z. sum(X::'a,Y,Z) −−> sum(Y::'a,X,Z)) &
(∀ Y U Z X V W. product(X::'a,Y,U) & product(Y::'a,Z,V) & product(U::'a,Z,W)
−−> product(X::'a,V,W)) &
(∀ Y X V U Z W. product(X::'a,Y,U) & product(Y::'a,Z,V) & product(X::'a,V,W)

$--> product(U::'a,Z,W))$ &
$(\forall\ Y\ Z\ X\ V3\ V1\ V2\ V4.\ product(X::'a,Y,V1)$ & $product(X::'a,Z,V2)$ & $sum(Y::'a,Z,V3)$
& $product(X::'a,V3,V4)$ $--> sum(V1::'a,V2,V4))$ &
$(\forall\ Y\ Z\ V1\ V2\ X\ V3\ V4.\ product(X::'a,Y,V1)$ & $product(X::'a,Z,V2)$ & $sum(Y::'a,Z,V3)$
& $sum(V1::'a,V2,V4)$ $--> product(X::'a,V3,V4))$ &
$(\forall\ Y\ Z\ V3\ X\ V1\ V2\ V4.\ product(Y::'a,X,V1)$ & $product(Z::'a,X,V2)$ & $sum(Y::'a,Z,V3)$
& $product(V3::'a,X,V4)$ $--> sum(V1::'a,V2,V4))$ &
$(\forall\ Y\ Z\ V1\ V2\ V3\ X\ V4.\ product(Y::'a,X,V1)$ & $product(Z::'a,X,V2)$ & $sum(Y::'a,Z,V3)$
& $sum(V1::'a,V2,V4)$ $--> product(V3::'a,X,V4))$ &
$(\forall\ X\ Y\ U\ V.\ sum(X::'a,Y,U)$ & $sum(X::'a,Y,V)$ $--> equal(U::'a,V))$ &
$(\forall\ X\ Y\ U\ V.\ product(X::'a,Y,U)$ & $product(X::'a,Y,V)$ $--> equal(U::'a,V))$
&
$(\forall\ X\ Y.\ equal(X::'a,Y)$ $--> equal(additive\text{-}inverse(X),additive\text{-}inverse(Y)))$ &
$(\forall\ X\ Y\ W.\ equal(X::'a,Y)$ $--> equal(add(X::'a,W),add(Y::'a,W)))$ &
$(\forall\ X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $sum(X::'a,W,Z)$ $--> sum(Y::'a,W,Z))$ &
$(\forall\ X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $sum(W::'a,X,Z)$ $--> sum(W::'a,Y,Z))$ &
$(\forall\ X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $sum(W::'a,Z,X)$ $--> sum(W::'a,Z,Y))$ &
$(\forall\ X\ Y\ W.\ equal(X::'a,Y)$ $--> equal(multiply(X::'a,W),multiply(Y::'a,W)))$
&
$(\forall\ X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $product(X::'a,W,Z)$ $--> product(Y::'a,W,Z))$
&
$(\forall\ X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $product(W::'a,X,Z)$ $--> product(W::'a,Y,Z))$
&
$(\forall\ X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $product(W::'a,Z,X)$ $--> product(W::'a,Z,Y))$

lemma *RNG001-5*:
EQU001-0-ax equal &
$(\forall\ X.\ sum(additive\text{-}identity::'a,X,X))$ &
$(\forall\ X.\ sum(X::'a,additive\text{-}identity,X))$ &
$(\forall\ X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall\ X\ Y.\ sum(X::'a,Y,add(X::'a,Y)))$ &
$(\forall\ X.\ sum(additive\text{-}inverse(X),X,additive\text{-}identity))$ &
RNG-other-ax multiply add equal product additive-identity additive-inverse sum
&
$(\sim product(a::'a,additive\text{-}identity,additive\text{-}identity))$ $--> False$
⟨*proof*⟩

lemma *RNG011-5*:
EQU001-0-ax equal &
$(\forall\ A\ B\ C.\ equal(A::'a,B)$ $--> equal(add(A::'a,C),add(B::'a,C)))$ &
$(\forall\ D\ F'\ E.\ equal(D::'a,E)$ $--> equal(add(F'::'a,D),add(F'::'a,E)))$ &
$(\forall\ G\ H.\ equal(G::'a,H)$ $--> equal(additive\text{-}inverse(G),additive\text{-}inverse(H)))$ &
$(\forall\ I'\ J\ K'.\ equal(I'::'a,J)$ $--> equal(multiply(I'::'a,K'),multiply(J::'a,K')))$ &
$(\forall\ L\ N\ M.\ equal(L::'a,M)$ $--> equal(multiply(N::'a,L),multiply(N::'a,M)))$ &
$(\forall\ A\ B\ C\ D.\ equal(A::'a,B)$ $--> equal(associator(A::'a,C,D),associator(B::'a,C,D)))$
&

$(\forall\, E\ G\ F'\ H.\ equal(E::'a,F') \longrightarrow equal(associator(G::'a,E,H),associator(G::'a,F',H)))$
&
$(\forall\, I'\ K'\ L\ J.\ equal(I'::'a,J) \longrightarrow equal(associator(K'::'a,L,I'),associator(K'::'a,L,J)))$
&
$(\forall\, M\ N\ O'.\ equal(M::'a,N) \longrightarrow equal(commutator(M::'a,O'),commutator(N::'a,O')))$
&
$(\forall\, P\ R\ Q.\ equal(P::'a,Q) \longrightarrow equal(commutator(R::'a,P),commutator(R::'a,Q)))$
&
$(\forall\, Y\ X.\ equal(add(X::'a,Y),add(Y::'a,X)))$ &
$(\forall\, X\ Y\ Z.\ equal(add(add(X::'a,Y),Z),add(X::'a,add(Y::'a,Z))))$ &
$(\forall\, X.\ equal(add(X::'a,additive\text{-}identity),X))$ &
$(\forall\, X.\ equal(add(additive\text{-}identity::'a,X),X))$ &
$(\forall\, X.\ equal(add(X::'a,additive\text{-}inverse(X)),additive\text{-}identity))$ &
$(\forall\, X.\ equal(add(additive\text{-}inverse(X),X),additive\text{-}identity))$ &
$(equal(additive\text{-}inverse(additive\text{-}identity),additive\text{-}identity))$ &
$(\forall\, X\ Y.\ equal(add(X::'a,add(additive\text{-}inverse(X),Y)),Y))$ &
$(\forall\, X\ Y.\ equal(additive\text{-}inverse(add(X::'a,Y)),add(additive\text{-}inverse(X),additive\text{-}inverse(Y))))$
&
$(\forall\, X.\ equal(additive\text{-}inverse(additive\text{-}inverse(X)),X))$ &
$(\forall\, X.\ equal(multiply(X::'a,additive\text{-}identity),additive\text{-}identity))$ &
$(\forall\, X.\ equal(multiply(additive\text{-}identity::'a,X),additive\text{-}identity))$ &
$(\forall\, X\ Y.\ equal(multiply(additive\text{-}inverse(X),additive\text{-}inverse(Y)),multiply(X::'a,Y)))$
&
$(\forall\, X\ Y.\ equal(multiply(X::'a,additive\text{-}inverse(Y)),additive\text{-}inverse(multiply(X::'a,Y))))$
&
$(\forall\, X\ Y.\ equal(multiply(additive\text{-}inverse(X),Y),additive\text{-}inverse(multiply(X::'a,Y))))$
&
$(\forall\, Y\ X\ Z.\ equal(multiply(X::'a,add(Y::'a,Z)),add(multiply(X::'a,Y),multiply(X::'a,Z))))$
&
$(\forall\, X\ Y\ Z.\ equal(multiply(add(X::'a,Y),Z),add(multiply(X::'a,Z),multiply(Y::'a,Z))))$
&
$(\forall\, X\ Y.\ equal(multiply(multiply(X::'a,Y),Y),multiply(X::'a,multiply(Y::'a,Y))))$
&
$(\forall\, X\ Y\ Z.\ equal(associator(X::'a,Y,Z),add(multiply(multiply(X::'a,Y),Z),additive\text{-}inverse(multiply(X::'a,m$
&
$(\forall\, X\ Y.\ equal(commutator(X::'a,Y),add(multiply(Y::'a,X),additive\text{-}inverse(multiply(X::'a,Y)))))$
&
$(\forall\, X\ Y.\ equal(multiply(multiply(associator(X::'a,X,Y),X),associator(X::'a,X,Y)),additive\text{-}identity))$
&
$(\sim equal(multiply(multiply(associator(a::'a,a,b),a),associator(a::'a,a,b)),additive\text{-}identity))$
$\longrightarrow False$
$\langle proof \rangle$

lemma *RNG023-6*:
EQU001-0-ax equal &
$(\forall\, Y\ X.\ equal(add(X::'a,Y),add(Y::'a,X)))$ &
$(\forall\, X\ Y\ Z.\ equal(add(X::'a,add(Y::'a,Z)),add(add(X::'a,Y),Z)))$ &
$(\forall\, X.\ equal(add(additive\text{-}identity::'a,X),X))$ &

$(\forall X.\ equal(add(X::'a,additive\text{-}identity),X))\ \&$
$(\forall X.\ equal(multiply(additive\text{-}identity::'a,X),additive\text{-}identity))\ \&$
$(\forall X.\ equal(multiply(X::'a,additive\text{-}identity),additive\text{-}identity))\ \&$
$(\forall X.\ equal(add(additive\text{-}inverse(X),X),additive\text{-}identity))\ \&$
$(\forall X.\ equal(add(X::'a,additive\text{-}inverse(X)),additive\text{-}identity))\ \&$
$(\forall Y\ X\ Z.\ equal(multiply(X::'a,add(Y::'a,Z)),add(multiply(X::'a,Y),multiply(X::'a,Z))))$
$\&$
$(\forall X\ Y\ Z.\ equal(multiply(add(X::'a,Y),Z),add(multiply(X::'a,Z),multiply(Y::'a,Z))))$
$\&$
$(\forall X.\ equal(additive\text{-}inverse(additive\text{-}inverse(X)),X))\ \&$
$(\forall X\ Y.\ equal(multiply(multiply(X::'a,Y),Y),multiply(X::'a,multiply(Y::'a,Y))))$
$\&$
$(\forall X\ Y.\ equal(multiply(multiply(X::'a,X),Y),multiply(X::'a,multiply(X::'a,Y))))$
$\&$
$(\forall X\ Y\ Z.\ equal(associator(X::'a,Y,Z),add(multiply(multiply(X::'a,Y),Z),additive\text{-}inverse(multiply(X::'a,m$
$\&$
$(\forall X\ Y.\ equal(commutator(X::'a,Y),add(multiply(Y::'a,X),additive\text{-}inverse(multiply(X::'a,Y)))))$
$\&$
$(\forall D\ E\ F'.\ equal(D::'a,E) \longrightarrow equal(add(D::'a,F'),add(E::'a,F')))\ \&$
$(\forall G\ I'\ H.\ equal(G::'a,H) \longrightarrow equal(add(I'::'a,G),add(I'::'a,H)))\ \&$
$(\forall J\ K'.\ equal(J::'a,K') \longrightarrow equal(additive\text{-}inverse(J),additive\text{-}inverse(K')))\ \&$
$(\forall L\ M\ N\ O'.\ equal(L::'a,M) \longrightarrow equal(associator(L::'a,N,O'),associator(M::'a,N,O')))$
$\&$
$(\forall P\ R\ Q\ S'.\ equal(P::'a,Q) \longrightarrow equal(associator(R::'a,P,S'),associator(R::'a,Q,S')))$
$\&$
$(\forall T'\ V\ W\ U.\ equal(T'::'a,U) \longrightarrow equal(associator(V::'a,W,T'),associator(V::'a,W,U)))$
$\&$
$(\forall X\ Y\ Z.\ equal(X::'a,Y) \longrightarrow equal(commutator(X::'a,Z),commutator(Y::'a,Z)))$
$\&$
$(\forall A1\ C1\ B1.\ equal(A1::'a,B1) \longrightarrow equal(commutator(C1::'a,A1),commutator(C1::'a,B1)))$
$\&$
$(\forall D1\ E1\ F1.\ equal(D1::'a,E1) \longrightarrow equal(multiply(D1::'a,F1),multiply(E1::'a,F1)))$
$\&$
$(\forall G1\ I1\ H1.\ equal(G1::'a,H1) \longrightarrow equal(multiply(I1::'a,G1),multiply(I1::'a,H1)))$
$\&$
$(\sim equal(associator(x::'a,x,y),additive\text{-}identity)) \longrightarrow False$
$\langle proof \rangle$

lemma *RNG028-2*:
$EQU001\text{-}0\text{-}ax\ equal\ \&$
$(\forall X.\ equal(add(additive\text{-}identity::'a,X),X))\ \&$
$(\forall X.\ equal(multiply(additive\text{-}identity::'a,X),additive\text{-}identity))\ \&$
$(\forall X.\ equal(multiply(X::'a,additive\text{-}identity),additive\text{-}identity))\ \&$
$(\forall X.\ equal(add(additive\text{-}inverse(X),X),additive\text{-}identity))\ \&$
$(\forall X\ Y.\ equal(additive\text{-}inverse(add(X::'a,Y)),add(additive\text{-}inverse(X),additive\text{-}inverse(Y))))$
$\&$
$(\forall X.\ equal(additive\text{-}inverse(additive\text{-}inverse(X)),X))\ \&$
$(\forall Y\ X\ Z.\ equal(multiply(X::'a,add(Y::'a,Z)),add(multiply(X::'a,Y),multiply(X::'a,Z))))$

&
(∀ X Y Z. equal(multiply(add(X::'a,Y),Z),add(multiply(X::'a,Z),multiply(Y::'a,Z))))
&
(∀ X Y. equal(multiply(multiply(X::'a,Y),Y),multiply(X::'a,multiply(Y::'a,Y))))
&
(∀ X Y. equal(multiply(multiply(X::'a,X),Y),multiply(X::'a,multiply(X::'a,Y))))
&
(∀ X Y. equal(multiply(additive-inverse(X),Y),additive-inverse(multiply(X::'a,Y))))
&
(∀ X Y. equal(multiply(X::'a,additive-inverse(Y)),additive-inverse(multiply(X::'a,Y))))
&
(equal(additive-inverse(additive-identity),additive-identity)) &
(∀ Y X. equal(add(X::'a,Y),add(Y::'a,X))) &
(∀ X Y Z. equal(add(X::'a,add(Y::'a,Z)),add(add(X::'a,Y),Z))) &
(∀ Z X Y. equal(add(X::'a,Z),add(Y::'a,Z)) −−> equal(X::'a,Y)) &
(∀ Z X Y. equal(add(Z::'a,X),add(Z::'a,Y)) −−> equal(X::'a,Y)) &
(∀ D E F'. equal(D::'a,E) −−> equal(add(D::'a,F'),add(E::'a,F'))) &
(∀ G I' H. equal(G::'a,H) −−> equal(add(I'::'a,G),add(I'::'a,H))) &
(∀ J K'. equal(J::'a,K') −−> equal(additive-inverse(J),additive-inverse(K'))) &
(∀ D1 E1 F1. equal(D1::'a,E1) −−> equal(multiply(D1::'a,F1),multiply(E1::'a,F1)))
&
(∀ G1 I1 H1. equal(G1::'a,H1) −−> equal(multiply(I1::'a,G1),multiply(I1::'a,H1)))
&
(∀ X Y Z. equal(associator(X::'a,Y,Z),add(multiply(multiply(X::'a,Y),Z),additive-inverse(multiply(X::'a,m
&
(∀ L M N O'. equal(L::'a,M) −−> equal(associator(L::'a,N,O'),associator(M::'a,N,O')))
&
(∀ P R Q S'. equal(P::'a,Q) −−> equal(associator(R::'a,P,S'),associator(R::'a,Q,S')))
&
(∀ T' V W U. equal(T'::'a,U) −−> equal(associator(V::'a,W,T'),associator(V::'a,W,U)))
&
(∀ X Y. ~equal(multiply(multiply(Y::'a,X),Y),multiply(Y::'a,multiply(X::'a,Y))))
&
(∀ X Y Z. ~equal(associator(Y::'a,X,Z),additive-inverse(associator(X::'a,Y,Z))))
&
(∀ X Y Z. ~equal(associator(Z::'a,Y,X),additive-inverse(associator(X::'a,Y,Z))))
&
(~equal(multiply(multiply(cx::'a,multiply(cy::'a,cx)),cz),multiply(cx::'a,multiply(cy::'a,multiply(cx::'a,cz))))
−−> False
⟨proof⟩

lemma *RNG038-2*:
(∀ X. sum(X::'a,additive-identity,X)) &
(∀ X Y. product(X::'a,Y,multiply(X::'a,Y))) &
(∀ X Y. sum(X::'a,Y,add(X::'a,Y))) &
RNG-other-ax multiply add equal product additive-identity additive-inverse sum
&
(∀ X. product(additive-identity::'a,X,additive-identity)) &

$(\forall X.\ product(X::'a,additive\text{-}identity,additive\text{-}identity))$ &
$(\forall X\ Y.\ equal(X::'a,additive\text{-}identity) \longrightarrow product(X::'a,h(X::'a,Y),Y))$ &
$(product(a::'a,b,additive\text{-}identity))$ &
$(\sim equal(a::'a,additive\text{-}identity))$ &
$(\sim equal(b::'a,additive\text{-}identity)) \longrightarrow False$
$\langle proof \rangle$

lemma *RNG040-2*:
EQU001-0-ax equal &
RNG001-0-eq product multiply sum add additive-inverse equal &
$(\forall X.\ sum(additive\text{-}identity::'a,X,X))$ &
$(\forall X.\ sum(X::'a,additive\text{-}identity,X))$ &
$(\forall X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall X\ Y.\ sum(X::'a,Y,add(X::'a,Y)))$ &
$(\forall X.\ sum(additive\text{-}inverse(X),X,additive\text{-}identity))$ &
$(\forall X.\ sum(X::'a,additive\text{-}inverse(X),additive\text{-}identity))$ &
$(\forall Y\ U\ Z\ X\ V\ W.\ sum(X::'a,Y,U)\ \&\ sum(Y::'a,Z,V)\ \&\ sum(U::'a,Z,W) \longrightarrow sum(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ sum(X::'a,Y,U)\ \&\ sum(Y::'a,Z,V)\ \&\ sum(X::'a,V,W) \longrightarrow sum(U::'a,Z,W))$ &
$(\forall Y\ X\ Z.\ sum(X::'a,Y,Z) \longrightarrow sum(Y::'a,X,Z))$ &
$(\forall Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)\ \&\ product(Y::'a,Z,V)\ \&\ product(U::'a,Z,W) \longrightarrow product(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)\ \&\ product(Y::'a,Z,V)\ \&\ product(X::'a,V,W) \longrightarrow product(U::'a,Z,W))$ &
$(\forall Y\ Z\ X\ V3\ V1\ V2\ V4.\ product(X::'a,Y,V1)\ \&\ product(X::'a,Z,V2)\ \&\ sum(Y::'a,Z,V3)\ \&\ product(X::'a,V3,V4) \longrightarrow sum(V1::'a,V2,V4))$ &
$(\forall Y\ Z\ V1\ V2\ X\ V3\ V4.\ product(X::'a,Y,V1)\ \&\ product(X::'a,Z,V2)\ \&\ sum(Y::'a,Z,V3)\ \&\ sum(V1::'a,V2,V4) \longrightarrow product(X::'a,V3,V4))$ &
$(\forall X\ Y\ U\ V.\ sum(X::'a,Y,U)\ \&\ sum(X::'a,Y,V) \longrightarrow equal(U::'a,V))$ &
$(\forall X\ Y\ U\ V.\ product(X::'a,Y,U)\ \&\ product(X::'a,Y,V) \longrightarrow equal(U::'a,V))$
&
$(\forall A.\ product(A::'a,multiplicative\text{-}identity,A))$ &
$(\forall A.\ product(multiplicative\text{-}identity::'a,A,A))$ &
$(\forall A.\ product(A::'a,h(A),multiplicative\text{-}identity) \mid equal(A::'a,additive\text{-}identity))$
&
$(\forall A.\ product(h(A),A,multiplicative\text{-}identity) \mid equal(A::'a,additive\text{-}identity))$ &
$(\forall B\ A\ C.\ product(A::'a,B,C) \longrightarrow product(B::'a,A,C))$ &
$(\forall A\ B.\ equal(A::'a,B) \longrightarrow equal(h(A),h(B)))$ &
$(sum(b::'a,c,d))$ &
$(product(d::'a,a,additive\text{-}identity))$ &
$(product(b::'a,a,l))$ &
$(product(c::'a,a,n))$ &
$(\sim sum(l::'a,n,additive\text{-}identity)) \longrightarrow False$
$\langle proof \rangle$

lemma *RNG041-1*:

EQU001-0-ax equal &
RNG001-0-ax equal additive-inverse add multiply product additive-identity sum &
RNG001-0-eq product multiply sum add additive-inverse equal &
$(\forall\ A\ B.\ equal(A::'a,B)\ \text{--}\!>\ equal(h(A),h(B)))$ &
$(\forall\ A.\ product(additive\text{-}identity::'a,A,additive\text{-}identity))$ &
$(\forall\ A.\ product(A::'a,additive\text{-}identity,additive\text{-}identity))$ &
$(\forall\ A.\ product(A::'a,multiplicative\text{-}identity,A))$ &
$(\forall\ A.\ product(multiplicative\text{-}identity::'a,A,A))$ &
$(\forall\ A.\ product(A::'a,h(A),multiplicative\text{-}identity)\ |\ equal(A::'a,additive\text{-}identity))$
&
$(\forall\ A.\ product(h(A),A,multiplicative\text{-}identity)\ |\ equal(A::'a,additive\text{-}identity))$ &
$(product(a::'a,b,additive\text{-}identity))$ &
$(\sim equal(a::'a,additive\text{-}identity))$ &
$(\sim equal(b::'a,additive\text{-}identity))\ \text{--}\!>\ False$
⟨*proof*⟩

lemma *ROB010-1*:
EQU001-0-ax equal &
$(\forall\ Y\ X.\ equal(add(X::'a,Y),add(Y::'a,X)))$ &
$(\forall\ X\ Y\ Z.\ equal(add(add(X::'a,Y),Z),add(X::'a,add(Y::'a,Z))))$ &
$(\forall\ Y\ X.\ equal(negate(add(negate(add(X::'a,Y)),negate(add(X::'a,negate(Y))))),X))$
&
$(\forall\ A\ B\ C.\ equal(A::'a,B)\ \text{--}\!>\ equal(add(A::'a,C),add(B::'a,C)))$ &
$(\forall\ D\ F'\ E.\ equal(D::'a,E)\ \text{--}\!>\ equal(add(F'::'a,D),add(F'::'a,E)))$ &
$(\forall\ G\ H.\ equal(G::'a,H)\ \text{--}\!>\ equal(negate(G),negate(H)))$ &
$(equal(negate(add(a::'a,negate(b))),c))$ &
$(\sim equal(negate(add(c::'a,negate(add(b::'a,a))))),a))\ \text{--}\!>\ False$
⟨*proof*⟩

lemma *ROB013-1*:
EQU001-0-ax equal &
$(\forall\ Y\ X.\ equal(add(X::'a,Y),add(Y::'a,X)))$ &
$(\forall\ X\ Y\ Z.\ equal(add(add(X::'a,Y),Z),add(X::'a,add(Y::'a,Z))))$ &
$(\forall\ Y\ X.\ equal(negate(add(negate(add(X::'a,Y)),negate(add(X::'a,negate(Y))))),X))$
&
$(\forall\ A\ B\ C.\ equal(A::'a,B)\ \text{--}\!>\ equal(add(A::'a,C),add(B::'a,C)))$ &
$(\forall\ D\ F'\ E.\ equal(D::'a,E)\ \text{--}\!>\ equal(add(F'::'a,D),add(F'::'a,E)))$ &
$(\forall\ G\ H.\ equal(G::'a,H)\ \text{--}\!>\ equal(negate(G),negate(H)))$ &
$(equal(negate(add(a::'a,b)),c))$ &
$(\sim equal(negate(add(c::'a,negate(add(negate(b),a))))),a))\ \text{--}\!>\ False$
⟨*proof*⟩

lemma *ROB016-1*:
EQU001-0-ax equal &
$(\forall\ Y\ X.\ equal(add(X::'a,Y),add(Y::'a,X)))$ &

$(\forall\ X\ Y\ Z.\ equal(add(add(X::'a,Y),Z),add(X::'a,add(Y::'a,Z)))) \&$
$(\forall\ Y\ X.\ equal(negate(add(negate(add(X::'a,Y)),negate(add(X::'a,negate(Y))))),X))$
$\&$
$(\forall\ A\ B\ C.\ equal(A::'a,B) \longrightarrow equal(add(A::'a,C),add(B::'a,C))) \&$
$(\forall\ D\ F'\ E.\ equal(D::'a,E) \longrightarrow equal(add(F'::'a,D),add(F'::'a,E))) \&$
$(\forall\ G\ H.\ equal(G::'a,H) \longrightarrow equal(negate(G),negate(H))) \&$
$(\forall\ J\ K'\ L.\ equal(J::'a,K') \longrightarrow equal(multiply(J::'a,L),multiply(K'::'a,L))) \&$
$(\forall\ M\ O'\ N.\ equal(M::'a,N) \longrightarrow equal(multiply(O'::'a,M),multiply(O'::'a,N)))$
$\&$
$(\forall\ P\ Q.\ equal(P::'a,Q) \longrightarrow equal(successor(P),successor(Q))) \&$
$(\forall\ R\ S'.\ equal(R::'a,S') \&\ positive\text{-}integer(R) \longrightarrow positive\text{-}integer(S')) \&$
$(\forall\ X.\ equal(multiply(One::'a,X),X)) \&$
$(\forall\ V\ X.\ positive\text{-}integer(X) \longrightarrow equal(multiply(successor(V),X),add(X::'a,multiply(V::'a,X))))$
$\&$
$(positive\text{-}integer(One)) \&$
$(\forall\ X.\ positive\text{-}integer(X) \longrightarrow positive\text{-}integer(successor(X))) \&$
$(equal(negate(add(d::'a,e)),negate(e))) \&$
$(positive\text{-}integer(k)) \&$
$(\forall\ Vk\ X\ Y.\ equal(negate(add(negate(Y),negate(add(X::'a,negate(Y))))),X) \&$
$positive\text{-}integer(Vk) \longrightarrow equal(negate(add(Y::'a,multiply(Vk::'a,add(X::'a,negate(add(X::'a,negate(Y))))))$
$\&$
$(\sim equal(negate(add(e::'a,multiply(k::'a,add(d::'a,negate(add(d::'a,negate(e)))))))),negate(e)))$
$\longrightarrow False$
$\langle proof \rangle$

lemma *ROB021-1*:
$EQU001\text{-}0\text{-}ax\ equal \&$
$(\forall\ Y\ X.\ equal(add(X::'a,Y),add(Y::'a,X))) \&$
$(\forall\ X\ Y\ Z.\ equal(add(add(X::'a,Y),Z),add(X::'a,add(Y::'a,Z)))) \&$
$(\forall\ Y\ X.\ equal(negate(add(negate(add(X::'a,Y)),negate(add(X::'a,negate(Y))))),X))$
$\&$
$(\forall\ A\ B\ C.\ equal(A::'a,B) \longrightarrow equal(add(A::'a,C),add(B::'a,C))) \&$
$(\forall\ D\ F'\ E.\ equal(D::'a,E) \longrightarrow equal(add(F'::'a,D),add(F'::'a,E))) \&$
$(\forall\ G\ H.\ equal(G::'a,H) \longrightarrow equal(negate(G),negate(H))) \&$
$(\forall\ X\ Y.\ equal(negate(X),negate(Y)) \longrightarrow equal(X::'a,Y)) \&$
$(\sim equal(add(negate(add(a::'a,negate(b))),negate(add(negate(a),negate(b)))),b))$
$\longrightarrow False$
$\langle proof \rangle$

lemma *SET005-1*:
$(\forall\ Subset\ Element\ Superset.\ member(Element::'a,Subset) \&\ subset(Subset::'a,Superset)$
$\longrightarrow member(Element::'a,Superset)) \&$
$(\forall\ Superset\ Subset.\ subset(Subset::'a,Superset) \mid member(member\text{-}of\text{-}1\text{-}not\text{-}of\text{-}2(Subset::'a,Superset),Subset))$
$\&$
$(\forall\ Subset\ Superset.\ member(member\text{-}of\text{-}1\text{-}not\text{-}of\text{-}2(Subset::'a,Superset),Superset)$
$\longrightarrow subset(Subset::'a,Superset)) \&$
$(\forall\ Subset\ Superset.\ equal\text{-}sets(Subset::'a,Superset) \longrightarrow subset(Subset::'a,Superset))$

&
(∀ *Subset Superset*. *equal-sets*(*Superset*::'*a*,*Subset*) −−> *subset*(*Subset*::'*a*,*Superset*))
&
(∀ *Set2 Set1*. *subset*(*Set1*::'*a*,*Set2*) & *subset*(*Set2*::'*a*,*Set1*) −−> *equal-sets*(*Set2*::'*a*,*Set1*))
&
(∀ *Set2 Intersection Element Set1*. *intersection*(*Set1*::'*a*,*Set2*,*Intersection*) & *member*(*Element*::'*a*,*Intersection*) −−> *member*(*Element*::'*a*,*Set1*)) &
(∀ *Set1 Intersection Element Set2*. *intersection*(*Set1*::'*a*,*Set2*,*Intersection*) & *member*(*Element*::'*a*,*Intersection*) −−> *member*(*Element*::'*a*,*Set2*)) &
(∀ *Set2 Set1 Element Intersection*. *intersection*(*Set1*::'*a*,*Set2*,*Intersection*) & *member*(*Element*::'*a*,*Set2*) & *member*(*Element*::'*a*,*Set1*) −−> *member*(*Element*::'*a*,*Intersection*))
&
(∀ *Set2 Intersection Set1*. *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Intersection*) | *intersection*(*Set1*::'*a*,*Set2*,*Intersection*) | *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Set1*))
&
(∀ *Set1 Intersection Set2*. *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Intersection*) | *intersection*(*Set1*::'*a*,*Set2*,*Intersection*) | *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Set2*))
&
(∀ *Set1 Set2 Intersection*. *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Intersection*) & *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Set2*) & *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Set1*) −−> *intersection*(*Set1*::'*a*,*Set2*,*Intersection*)) &
(*intersection*(*a*::'*a*,*b*,*aIb*)) &
(*intersection*(*b*::'*a*,*c*,*bIc*)) &
(*intersection*(*a*::'*a*,*bIc*,*aIbIc*)) &
(~*intersection*(*aIb*::'*a*,*c*,*aIbIc*)) −−> *False*
⟨*proof*⟩

lemma *SET009-1*:
(∀ *Subset Element Superset*. *member*(*Element*::'*a*,*Subset*) & *ssubset*(*Subset*::'*a*,*Superset*) −−> *member*(*Element*::'*a*,*Superset*)) &
(∀ *Superset Subset*. *ssubset*(*Subset*::'*a*,*Superset*) | *member*(*member-of-1-not-of-2*(*Subset*::'*a*,*Superset*),*Subset*))
&
(∀ *Subset Superset*. *member*(*member-of-1-not-of-2*(*Subset*::'*a*,*Superset*),*Superset*) −−> *ssubset*(*Subset*::'*a*,*Superset*)) &
(∀ *Subset Superset*. *equal-sets*(*Subset*::'*a*,*Superset*) −−> *ssubset*(*Subset*::'*a*,*Superset*))
&
(∀ *Subset Superset*. *equal-sets*(*Superset*::'*a*,*Subset*) −−> *ssubset*(*Subset*::'*a*,*Superset*))
&
(∀ *Set2 Set1*. *ssubset*(*Set1*::'*a*,*Set2*) & *ssubset*(*Set2*::'*a*,*Set1*) −−> *equal-sets*(*Set2*::'*a*,*Set1*))
&
(∀ *Set2 Difference Element Set1*. *difference*(*Set1*::'*a*,*Set2*,*Difference*) & *member*(*Element*::'*a*,*Difference*) −−> *member*(*Element*::'*a*,*Set1*)) &
(∀ *Element A-set Set1 Set2*. ~(*member*(*Element*::'*a*,*Set1*) & *member*(*Element*::'*a*,*Set2*) & *difference*(*A-set*::'*a*,*Set1*,*Set2*))) &
(∀ *Set1 Difference Element Set2*. *member*(*Element*::'*a*,*Set1*) & *difference*(*Set1*::'*a*,*Set2*,*Difference*) −−> *member*(*Element*::'*a*,*Difference*) | *member*(*Element*::'*a*,*Set2*)) &
(∀ *Set1 Set2 Difference*. *difference*(*Set1*::'*a*,*Set2*,*Difference*) | *member*(*k*(*Set1*::'*a*,*Set2*,*Difference*),*Set1*)

| $member(k(Set1::'a,Set2,Difference),Difference))$ &
$(\forall\ Set1\ Set2\ Difference.\ member(k(Set1::'a,Set2,Difference),Set2)\ --> member(k(Set1::'a,Set2,Difference),Difference)\ |\ difference(Set1::'a,Set2,Difference))$ &
$(\forall\ Set1\ Set2\ Difference.\ member(k(Set1::'a,Set2,Difference),Difference)\ \&\ member(k(Set1::'a,Set2,Difference),Set1)\ --> member(k(Set1::'a,Set2,Difference),Set2)\ |\ difference(Set1::'a,Set2,Difference))$ &
$(ssubset(d::'a,a))$ &
$(difference(b::'a,a,bDa))$ &
$(difference(b::'a,d,bDd))$ &
$(\sim ssubset(bDa::'a,bDd))\ --> False$
$\langle proof \rangle$

lemma *SET025-4*:
EQU001-0-ax equal &
$(\forall\ Y\ X.\ member(X::'a,Y)\ --> little\text{-}set(X))$ &
$(\forall\ X\ Y.\ little\text{-}set(f1(X::'a,Y))\ |\ equal(X::'a,Y))$ &
$(\forall\ X\ Y.\ member(f1(X::'a,Y),X)\ |\ member(f1(X::'a,Y),Y)\ |\ equal(X::'a,Y))$ &
$(\forall\ X\ Y.\ member(f1(X::'a,Y),X)\ \&\ member(f1(X::'a,Y),Y)\ --> equal(X::'a,Y))$ &
$(\forall\ X\ U\ Y.\ member(U::'a,non\text{-}ordered\text{-}pair(X::'a,Y))\ --> equal(U::'a,X)\ |\ equal(U::'a,Y))$ &
$(\forall\ Y\ U\ X.\ little\text{-}set(U)\ \&\ equal(U::'a,X)\ --> member(U::'a,non\text{-}ordered\text{-}pair(X::'a,Y)))$ &
$(\forall\ X\ U\ Y.\ little\text{-}set(U)\ \&\ equal(U::'a,Y)\ --> member(U::'a,non\text{-}ordered\text{-}pair(X::'a,Y)))$ &
$(\forall\ X\ Y.\ little\text{-}set(non\text{-}ordered\text{-}pair(X::'a,Y)))$ &
$(\forall\ X.\ equal(singleton\text{-}set(X),non\text{-}ordered\text{-}pair(X::'a,X)))$ &
$(\forall\ X\ Y.\ equal(ordered\text{-}pair(X::'a,Y),non\text{-}ordered\text{-}pair(singleton\text{-}set(X),non\text{-}ordered\text{-}pair(X::'a,Y))))$ &
$(\forall\ X.\ ordered\text{-}pair\text{-}predicate(X)\ --> little\text{-}set(f2(X)))$ &
$(\forall\ X.\ ordered\text{-}pair\text{-}predicate(X)\ --> little\text{-}set(f3(X)))$ &
$(\forall\ X.\ ordered\text{-}pair\text{-}predicate(X)\ --> equal(X::'a,ordered\text{-}pair(f2(X),f3(X))))$ &
$(\forall\ X\ Y\ Z.\ little\text{-}set(Y)\ \&\ little\text{-}set(Z)\ \&\ equal(X::'a,ordered\text{-}pair(Y::'a,Z))\ --> ordered\text{-}pair\text{-}predicate(X))$ &
$(\forall\ Z\ X.\ member(Z::'a,first(X))\ --> little\text{-}set(f4(Z::'a,X)))$ &
$(\forall\ Z\ X.\ member(Z::'a,first(X))\ --> little\text{-}set(f5(Z::'a,X)))$ &
$(\forall\ Z\ X.\ member(Z::'a,first(X))\ --> equal(X::'a,ordered\text{-}pair(f4(Z::'a,X),f5(Z::'a,X))))$ &
$(\forall\ Z\ X.\ member(Z::'a,first(X))\ --> member(Z::'a,f4(Z::'a,X)))$ &
$(\forall\ X\ V\ Z\ U.\ little\text{-}set(U)\ \&\ little\text{-}set(V)\ \&\ equal(X::'a,ordered\text{-}pair(U::'a,V))\ \&\ member(Z::'a,U)\ --> member(Z::'a,first(X)))$ &
$(\forall\ Z\ X.\ member(Z::'a,second(X))\ --> little\text{-}set(f6(Z::'a,X)))$ &
$(\forall\ Z\ X.\ member(Z::'a,second(X))\ --> little\text{-}set(f7(Z::'a,X)))$ &
$(\forall\ Z\ X.\ member(Z::'a,second(X))\ --> equal(X::'a,ordered\text{-}pair(f6(Z::'a,X),f7(Z::'a,X))))$ &
$(\forall\ Z\ X.\ member(Z::'a,second(X))\ --> member(Z::'a,f7(Z::'a,X)))$ &
$(\forall\ X\ U\ Z\ V.\ little\text{-}set(U)\ \&\ little\text{-}set(V)\ \&\ equal(X::'a,ordered\text{-}pair(U::'a,V))\ \&\ member(Z::'a,V)\ --> member(Z::'a,second(X)))$ &

$(\forall\, Z.\ member(Z::'a,estin)\ -->\ ordered\text{-}pair\text{-}predicate(Z))\ \&$
$(\forall\, Z.\ member(Z::'a,estin)\ -->\ member(first(Z),second(Z)))\ \&$
$(\forall\, Z.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Z)\ \&\ member(first(Z),second(Z))$
$-->\ member(Z::'a,estin))\ \&$
$(\forall\, Y\ Z\ X.\ member(Z::'a,intersection(X::'a,Y))\ -->\ member(Z::'a,X))\ \&$
$(\forall\, X\ Z\ Y.\ member(Z::'a,intersection(X::'a,Y))\ -->\ member(Z::'a,Y))\ \&$
$(\forall\, X\ Z\ Y.\ member(Z::'a,X)\ \&\ member(Z::'a,Y)\ -->\ member(Z::'a,intersection(X::'a,Y)))$
$\&$
$(\forall\, Z\ X.\ \sim(member(Z::'a,complement(X))\ \&\ member(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ little\text{-}set(Z)\ -->\ member(Z::'a,complement(X))\ |\ member(Z::'a,X))\ \&$
$(\forall\, X\ Y.\ equal(union(X::'a,Y),complement(intersection(complement(X),complement(Y)))))$
$\&$
$(\forall\, Z\ X.\ member(Z::'a,domain\text{-}of(X))\ -->\ ordered\text{-}pair\text{-}predicate(f8(Z::'a,X)))$
$\&$
$(\forall\, Z\ X.\ member(Z::'a,domain\text{-}of(X))\ -->\ member(f8(Z::'a,X),X))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,domain\text{-}of(X))\ -->\ equal(Z::'a,first(f8(Z::'a,X))))\ \&$
$(\forall\, X\ Z\ Xp.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Xp)\ \&\ member(Xp::'a,X)\ \&$
$equal(Z::'a,first(Xp))\ -->\ member(Z::'a,domain\text{-}of(X)))\ \&$
$(\forall\, X\ Y\ Z.\ member(Z::'a,cross\text{-}product(X::'a,Y))\ -->\ ordered\text{-}pair\text{-}predicate(Z))$
$\&$
$(\forall\, Y\ Z\ X.\ member(Z::'a,cross\text{-}product(X::'a,Y))\ -->\ member(first(Z),X))\ \&$
$(\forall\, X\ Z\ Y.\ member(Z::'a,cross\text{-}product(X::'a,Y))\ -->\ member(second(Z),Y))$
$\&$
$(\forall\, X\ Z\ Y.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Z)\ \&\ member(first(Z),X)\ \&$
$member(second(Z),Y)\ -->\ member(Z::'a,cross\text{-}product(X::'a,Y)))\ \&$
$(\forall\, X\ Z.\ member(Z::'a,inv1\ X)\ -->\ ordered\text{-}pair\text{-}predicate(Z))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,inv1\ X)\ -->\ member(ordered\text{-}pair(second(Z),first(Z)),X))$
$\&$
$(\forall\, Z\ X.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Z)\ \&\ member(ordered\text{-}pair(second(Z),first(Z)),X)$
$-->\ member(Z::'a,inv1\ X))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,rot\text{-}right(X))\ -->\ little\text{-}set(f9(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,rot\text{-}right(X))\ -->\ little\text{-}set(f10(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,rot\text{-}right(X))\ -->\ little\text{-}set(f11(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,rot\text{-}right(X))\ -->\ equal(Z::'a,ordered\text{-}pair(f9(Z::'a,X),ordered\text{-}pair(f10(Z::'a,X),f11$
$\&$
$(\forall\, Z\ X.\ member(Z::'a,rot\text{-}right(X))\ -->\ member(ordered\text{-}pair(f10(Z::'a,X),ordered\text{-}pair(f11(Z::'a,X),f9(Z$
$\&$
$(\forall\, Z\ V\ W\ U\ X.\ little\text{-}set(Z)\ \&\ little\text{-}set(U)\ \&\ little\text{-}set(V)\ \&\ little\text{-}set(W)\ \&$
$equal(Z::'a,ordered\text{-}pair(U::'a,ordered\text{-}pair(V::'a,W)))\ \&\ member(ordered\text{-}pair(V::'a,ordered\text{-}pair(W::'a,U))$
$-->\ member(Z::'a,rot\text{-}right(X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X))\ -->\ little\text{-}set(f12(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X))\ -->\ little\text{-}set(f13(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X))\ -->\ little\text{-}set(f14(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X))\ -->\ equal(Z::'a,ordered\text{-}pair(f12(Z::'a,X),ordered\text{-}pair(f13(Z::'a,X$
$\&$
$(\forall\, Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X))\ -->\ member(ordered\text{-}pair(f12(Z::'a,X),ordered\text{-}pair(f14(Z::'a,X),f$
$\&$
$(\forall\, Z\ U\ W\ V\ X.\ little\text{-}set(Z)\ \&\ little\text{-}set(U)\ \&\ little\text{-}set(V)\ \&\ little\text{-}set(W)\ \&$
$equal(Z::'a,ordered\text{-}pair(U::'a,ordered\text{-}pair(V::'a,W)))\ \&\ member(ordered\text{-}pair(U::'a,ordered\text{-}pair(W::'a,V))$

$$
\begin{array}{l}
--> member(Z::'a,flip\text{-}range\text{-}of(X))) \;\&\\
\quad(\forall X.\ equal(successor(X),union(X::'a,singleton\text{-}set(X)))) \;\&\\
\quad(\forall Z.\ \sim member(Z::'a,empty\text{-}set)) \;\&\\
\quad(\forall Z.\ little\text{-}set(Z) --> member(Z::'a,universal\text{-}set)) \;\&\\
\quad(little\text{-}set(infinity)) \;\&\\
\quad(member(empty\text{-}set::'a,infinity)) \;\&\\
\quad(\forall X.\ member(X::'a,infinity) --> member(successor(X),infinity)) \;\&\\
\quad(\forall Z\ X.\ member(Z::'a,sigma(X)) --> member(f16(Z::'a,X),X)) \;\&\\
\quad(\forall Z\ X.\ member(Z::'a,sigma(X)) --> member(Z::'a,f16(Z::'a,X))) \;\&\\
\quad(\forall X\ Z\ Y.\ member(Y::'a,X)\ \&\ member(Z::'a,Y) --> member(Z::'a,sigma(X)))\\
\&\\
\quad(\forall U.\ little\text{-}set(U) --> little\text{-}set(sigma(U))) \;\&\\
\quad(\forall X\ U\ Y.\ ssubset(X::'a,Y)\ \&\ member(U::'a,X) --> member(U::'a,Y)) \;\&\\
\quad(\forall Y\ X.\ ssubset(X::'a,Y) \mid member(f17(X::'a,Y),X)) \;\&\\
\quad(\forall X\ Y.\ member(f17(X::'a,Y),Y) --> ssubset(X::'a,Y)) \;\&\\
\quad(\forall X\ Y.\ proper\text{-}subset(X::'a,Y) --> ssubset(X::'a,Y)) \;\&\\
\quad(\forall X\ Y.\ \sim(proper\text{-}subset(X::'a,Y)\ \&\ equal(X::'a,Y))) \;\&\\
\quad(\forall X\ Y.\ ssubset(X::'a,Y) --> proper\text{-}subset(X::'a,Y) \mid equal(X::'a,Y)) \;\&\\
\quad(\forall Z\ X.\ member(Z::'a,powerset(X)) --> ssubset(Z::'a,X)) \;\&\\
\quad(\forall Z\ X.\ little\text{-}set(Z)\ \&\ ssubset(Z::'a,X) --> member(Z::'a,powerset(X))) \;\&\\
\quad(\forall U.\ little\text{-}set(U) --> little\text{-}set(powerset(U))) \;\&\\
\quad(\forall Z\ X.\ relation(Z)\ \&\ member(X::'a,Z) --> ordered\text{-}pair\text{-}predicate(X)) \;\&\\
\quad(\forall Z.\ relation(Z) \mid member(f18(Z),Z)) \;\&\\
\quad(\forall Z.\ ordered\text{-}pair\text{-}predicate(f18(Z)) --> relation(Z)) \;\&\\
\quad(\forall U\ X\ V\ W.\ single\text{-}valued\text{-}set(X)\ \&\ little\text{-}set(U)\ \&\ little\text{-}set(V)\ \&\ little\text{-}set(W)\\
\&\ member(ordered\text{-}pair(U::'a,V),X)\ \&\ member(ordered\text{-}pair(U::'a,W),X) -->\\
equal(V::'a,W)) \;\&\\
\quad(\forall X.\ single\text{-}valued\text{-}set(X) \mid little\text{-}set(f19(X))) \;\&\\
\quad(\forall X.\ single\text{-}valued\text{-}set(X) \mid little\text{-}set(f20(X))) \;\&\\
\quad(\forall X.\ single\text{-}valued\text{-}set(X) \mid little\text{-}set(f21(X))) \;\&\\
\quad(\forall X.\ single\text{-}valued\text{-}set(X) \mid member(ordered\text{-}pair(f19(X),f20(X)),X)) \;\&\\
\quad(\forall X.\ single\text{-}valued\text{-}set(X) \mid member(ordered\text{-}pair(f19(X),f21(X)),X)) \;\&\\
\quad(\forall X.\ equal(f20(X),f21(X)) --> single\text{-}valued\text{-}set(X)) \;\&\\
\quad(\forall Xf.\ function(Xf) --> relation(Xf)) \;\&\\
\quad(\forall Xf.\ function(Xf) --> single\text{-}valued\text{-}set(Xf)) \;\&\\
\quad(\forall Xf.\ relation(Xf)\ \&\ single\text{-}valued\text{-}set(Xf) --> function(Xf)) \;\&\\
\quad(\forall Z\ X\ Xf.\ member(Z::'a,image'(X::'a,Xf)) --> ordered\text{-}pair\text{-}predicate(f22(Z::'a,X,Xf)))\\
\&\\
\quad(\forall Z\ X\ Xf.\ member(Z::'a,image'(X::'a,Xf)) --> member(f22(Z::'a,X,Xf),Xf))\\
\&\\
\quad(\forall Z\ Xf\ X.\ member(Z::'a,image'(X::'a,Xf)) --> member(first(f22(Z::'a,X,Xf)),X))\\
\&\\
\quad(\forall X\ Xf\ Z.\ member(Z::'a,image'(X::'a,Xf)) --> equal(second(f22(Z::'a,X,Xf)),Z))\\
\&\\
\quad(\forall Xf\ X\ Y\ Z.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Y)\ \&\ member(Y::'a,Xf)\ \&\\
member(first(Y),X)\ \&\ equal(second(Y),Z) --> member(Z::'a,image'(X::'a,Xf)))\\
\&\\
\quad(\forall X\ Xf.\ little\text{-}set(X)\ \&\ function(Xf) --> little\text{-}set(image'(X::'a,Xf))) \;\&\\
\quad(\forall X\ U\ Y.\ \sim(disjoint(X::'a,Y)\ \&\ member(U::'a,X)\ \&\ member(U::'a,Y))) \;\&
\end{array}
$$

$$
\begin{array}{l}
(\forall\, Y\ X.\ \mathit{disjoint}(X::'a,Y) \mid \mathit{member}(\mathit{f23}(X::'a,Y),X)) \ \&\\
(\forall\, X\ Y.\ \mathit{disjoint}(X::'a,Y) \mid \mathit{member}(\mathit{f23}(X::'a,Y),Y)) \ \&\\
(\forall\, X.\ \mathit{equal}(X::'a,\mathit{empty\text{-}set}) \mid \mathit{member}(\mathit{f24}(X),X)) \ \&\\
(\forall\, X.\ \mathit{equal}(X::'a,\mathit{empty\text{-}set}) \mid \mathit{disjoint}(\mathit{f24}(X),X)) \ \&\\
(\mathit{function}(\mathit{f25})) \ \&\\
(\forall\, X.\ \mathit{little\text{-}set}(X) \longrightarrow \mathit{equal}(X::'a,\mathit{empty\text{-}set}) \mid \mathit{member}(\mathit{f26}(X),X)) \ \&\\
(\forall\, X.\ \mathit{little\text{-}set}(X) \longrightarrow \mathit{equal}(X::'a,\mathit{empty\text{-}set}) \mid \mathit{member}(\mathit{ordered\text{-}pair}(X::'a,\mathit{f26}(X)),\mathit{f25}))\\
\&\\
(\forall\, Z\ X.\ \mathit{member}(Z::'a,\mathit{range\text{-}of}(X)) \longrightarrow \mathit{ordered\text{-}pair\text{-}predicate}(\mathit{f27}(Z::'a,X)))\\
\&\\
(\forall\, Z\ X.\ \mathit{member}(Z::'a,\mathit{range\text{-}of}(X)) \longrightarrow \mathit{member}(\mathit{f27}(Z::'a,X),X)) \ \&\\
(\forall\, Z\ X.\ \mathit{member}(Z::'a,\mathit{range\text{-}of}(X)) \longrightarrow \mathit{equal}(Z::'a,\mathit{second}(\mathit{f27}(Z::'a,X)))) \ \&\\
(\forall\, X\ Z\ \mathit{Xp}.\ \mathit{little\text{-}set}(Z) \ \&\ \mathit{ordered\text{-}pair\text{-}predicate}(\mathit{Xp}) \ \&\ \mathit{member}(\mathit{Xp}::'a,X) \ \&\\
\mathit{equal}(Z::'a,\mathit{second}(\mathit{Xp})) \longrightarrow \mathit{member}(Z::'a,\mathit{range\text{-}of}(X))) \ \&\\
(\forall\, Z.\ \mathit{member}(Z::'a,\mathit{identity\text{-}relation}) \longrightarrow \mathit{ordered\text{-}pair\text{-}predicate}(Z)) \ \&\\
(\forall\, Z.\ \mathit{member}(Z::'a,\mathit{identity\text{-}relation}) \longrightarrow \mathit{equal}(\mathit{first}(Z),\mathit{second}(Z))) \ \&\\
(\forall\, Z.\ \mathit{little\text{-}set}(Z) \ \&\ \mathit{ordered\text{-}pair\text{-}predicate}(Z) \ \&\ \mathit{equal}(\mathit{first}(Z),\mathit{second}(Z)) \longrightarrow\\
\mathit{member}(Z::'a,\mathit{identity\text{-}relation})) \ \&\\
(\forall\, X\ Y.\ \mathit{equal}(\mathit{restrct}(X::'a,Y),\mathit{intersection}(X::'a,\mathit{cross\text{-}product}(Y::'a,\mathit{universal\text{-}set}))))\\
\&\\
(\forall\, \mathit{Xf}.\ \mathit{one\text{-}to\text{-}one\text{-}function}(\mathit{Xf}) \longrightarrow \mathit{function}(\mathit{Xf})) \ \&\\
(\forall\, \mathit{Xf}.\ \mathit{one\text{-}to\text{-}one\text{-}function}(\mathit{Xf}) \longrightarrow \mathit{function}(\mathit{inv1}\ \mathit{Xf})) \ \&\\
(\forall\, \mathit{Xf}.\ \mathit{function}(\mathit{Xf}) \ \&\ \mathit{function}(\mathit{inv1}\ \mathit{Xf}) \longrightarrow \mathit{one\text{-}to\text{-}one\text{-}function}(\mathit{Xf})) \ \&\\
(\forall\, Z\ \mathit{Xf}\ Y.\ \mathit{member}(Z::'a,\mathit{apply}(\mathit{Xf}::'a,Y)) \longrightarrow \mathit{ordered\text{-}pair\text{-}predicate}(\mathit{f28}(Z::'a,\mathit{Xf},Y)))\\
\&\\
(\forall\, Z\ Y\ \mathit{Xf}.\ \mathit{member}(Z::'a,\mathit{apply}(\mathit{Xf}::'a,Y)) \longrightarrow \mathit{member}(\mathit{f28}(Z::'a,\mathit{Xf},Y),\mathit{Xf}))\\
\&\\
(\forall\, Z\ \mathit{Xf}\ Y.\ \mathit{member}(Z::'a,\mathit{apply}(\mathit{Xf}::'a,Y)) \longrightarrow \mathit{equal}(\mathit{first}(\mathit{f28}(Z::'a,\mathit{Xf},Y)),Y))\\
\&\\
(\forall\, Z\ \mathit{Xf}\ Y.\ \mathit{member}(Z::'a,\mathit{apply}(\mathit{Xf}::'a,Y)) \longrightarrow \mathit{member}(Z::'a,\mathit{second}(\mathit{f28}(Z::'a,\mathit{Xf},Y))))\\
\&\\
(\forall\, \mathit{Xf}\ Y\ Z\ W.\ \mathit{ordered\text{-}pair\text{-}predicate}(W) \ \&\ \mathit{member}(W::'a,\mathit{Xf}) \ \&\ \mathit{equal}(\mathit{first}(W),Y)\\
\&\ \mathit{member}(Z::'a,\mathit{second}(W)) \longrightarrow \mathit{member}(Z::'a,\mathit{apply}(\mathit{Xf}::'a,Y))) \ \&\\
(\forall\, \mathit{Xf}\ X\ Y.\ \mathit{equal}(\mathit{apply\text{-}to\text{-}two\text{-}arguments}(\mathit{Xf}::'a,X,Y),\mathit{apply}(\mathit{Xf}::'a,\mathit{ordered\text{-}pair}(X::'a,Y))))\\
\&\\
(\forall\, X\ Y\ \mathit{Xf}.\ \mathit{maps}(\mathit{Xf}::'a,X,Y) \longrightarrow \mathit{function}(\mathit{Xf})) \ \&\\
(\forall\, Y\ \mathit{Xf}\ X.\ \mathit{maps}(\mathit{Xf}::'a,X,Y) \longrightarrow \mathit{equal}(\mathit{domain\text{-}of}(\mathit{Xf}),X)) \ \&\\
(\forall\, X\ \mathit{Xf}\ Y.\ \mathit{maps}(\mathit{Xf}::'a,X,Y) \longrightarrow \mathit{ssubset}(\mathit{range\text{-}of}(\mathit{Xf}),Y)) \ \&\\
(\forall\, X\ \mathit{Xf}\ Y.\ \mathit{function}(\mathit{Xf}) \ \&\ \mathit{equal}(\mathit{domain\text{-}of}(\mathit{Xf}),X) \ \&\ \mathit{ssubset}(\mathit{range\text{-}of}(\mathit{Xf}),Y)\\
\longrightarrow \mathit{maps}(\mathit{Xf}::'a,X,Y)) \ \&\\
(\forall\, \mathit{Xf}\ \mathit{Xs}.\ \mathit{closed}(\mathit{Xs}::'a,\mathit{Xf}) \longrightarrow \mathit{little\text{-}set}(\mathit{Xs})) \ \&\\
(\forall\, \mathit{Xs}\ \mathit{Xf}.\ \mathit{closed}(\mathit{Xs}::'a,\mathit{Xf}) \longrightarrow \mathit{little\text{-}set}(\mathit{Xf})) \ \&\\
(\forall\, \mathit{Xf}\ \mathit{Xs}.\ \mathit{closed}(\mathit{Xs}::'a,\mathit{Xf}) \longrightarrow \mathit{maps}(\mathit{Xf}::'a,\mathit{cross\text{-}product}(\mathit{Xs}::'a,\mathit{Xs}),\mathit{Xs})) \ \&\\
(\forall\, \mathit{Xf}\ \mathit{Xs}.\ \mathit{little\text{-}set}(\mathit{Xs}) \ \&\ \mathit{little\text{-}set}(\mathit{Xf}) \ \&\ \mathit{maps}(\mathit{Xf}::'a,\mathit{cross\text{-}product}(\mathit{Xs}::'a,\mathit{Xs}),\mathit{Xs})\\
\longrightarrow \mathit{closed}(\mathit{Xs}::'a,\mathit{Xf})) \ \&\\
(\forall\, Z\ \mathit{Xf}\ \mathit{Xg}.\ \mathit{member}(Z::'a,\mathit{composition}(\mathit{Xf}::'a,\mathit{Xg})) \longrightarrow \mathit{little\text{-}set}(\mathit{f29}(Z::'a,\mathit{Xf},\mathit{Xg})))\\
\&\\
(\forall\, Z\ \mathit{Xf}\ \mathit{Xg}.\ \mathit{member}(Z::'a,\mathit{composition}(\mathit{Xf}::'a,\mathit{Xg})) \longrightarrow \mathit{little\text{-}set}(\mathit{f30}(Z::'a,\mathit{Xf},\mathit{Xg})))\\
\&
\end{array}
$$

$(\forall\ Z\ Xf\ Xg.\ member(Z::'a,composition(Xf::'a,Xg)) \longrightarrow little\text{-}set(f31(Z::'a,Xf,Xg)))$
&
$(\forall\ Z\ Xf\ Xg.\ member(Z::'a,composition(Xf::'a,Xg)) \longrightarrow equal(Z::'a,ordered\text{-}pair(f29(Z::'a,Xf,Xg),f30(Z::'a$
&
$(\forall\ Z\ Xg\ Xf.\ member(Z::'a,composition(Xf::'a,Xg)) \longrightarrow member(ordered\text{-}pair(f29(Z::'a,Xf,Xg),f31(Z::'a,X$
&
$(\forall\ Z\ Xf\ Xg.\ member(Z::'a,composition(Xf::'a,Xg)) \longrightarrow member(ordered\text{-}pair(f31(Z::'a,Xf,Xg),f30(Z::'a,X$
&
$(\forall\ Z\ X\ Xf\ W\ Y\ Xg.\ little\text{-}set(Z)\ \&\ little\text{-}set(X)\ \&\ little\text{-}set(Y)\ \&\ little\text{-}set(W)\ \&$
$equal(Z::'a,ordered\text{-}pair(X::'a,Y))\ \&\ member(ordered\text{-}pair(X::'a,W),Xf)\ \&\ mem\text{-}$
$ber(ordered\text{-}pair(W::'a,Y),Xg) \longrightarrow member(Z::'a,composition(Xf::'a,Xg)))\ \&$
$(\forall\ Xh\ Xs2\ Xf2\ Xs1\ Xf1.\ homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) \longrightarrow closed(Xs1::'a,Xf1))$
&
$(\forall\ Xh\ Xs1\ Xf1\ Xs2\ Xf2.\ homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) \longrightarrow closed(Xs2::'a,Xf2))$
&
$(\forall\ Xf1\ Xf2\ Xh\ Xs1\ Xs2.\ homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) \longrightarrow maps(Xh::'a,Xs1,Xs2))$
&
$(\forall\ Xs2\ Xs1\ Xf1\ Xf2\ X\ Xh\ Y.\ homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2)\ \&\ mem\text{-}$
$ber(X::'a,Xs1)\ \&\ member(Y::'a,Xs1) \longrightarrow equal(apply(Xh::'a,apply\text{-}to\text{-}two\text{-}arguments(Xf1::'a,X,Y)),apply\text{-}t$
&
$(\forall\ Xh\ Xf1\ Xs2\ Xf2\ Xs1.\ closed(Xs1::'a,Xf1)\ \&\ closed(Xs2::'a,Xf2)\ \&\ maps(Xh::'a,Xs1,Xs2)$
$\longrightarrow homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) \mid member(f32(Xh::'a,Xs1,Xf1,Xs2,Xf2),Xs1))$
&
$(\forall\ Xh\ Xf1\ Xs2\ Xf2\ Xs1.\ closed(Xs1::'a,Xf1)\ \&\ closed(Xs2::'a,Xf2)\ \&\ maps(Xh::'a,Xs1,Xs2)$
$\longrightarrow homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) \mid member(f33(Xh::'a,Xs1,Xf1,Xs2,Xf2),Xs1))$
&
$(\forall\ Xh\ Xs1\ Xf1\ Xs2\ Xf2.\ closed(Xs1::'a,Xf1)\ \&\ closed(Xs2::'a,Xf2)\ \&\ maps(Xh::'a,Xs1,Xs2)$
$\&\ equal(apply(Xh::'a,apply\text{-}to\text{-}two\text{-}arguments(Xf1::'a,f32(Xh::'a,Xs1,Xf1,Xs2,Xf2),f33(Xh::'a,Xs1,Xf1,Xs2,$
$\longrightarrow homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2))\ \&$
$(\forall\ A\ B\ C.\ equal(A::'a,B) \longrightarrow equal(f1(A::'a,C),f1(B::'a,C)))\ \&$
$(\forall\ D\ F'\ E.\ equal(D::'a,E) \longrightarrow equal(f1(F'::'a,D),f1(F'::'a,E)))\ \&$
$(\forall\ A2\ B2.\ equal(A2::'a,B2) \longrightarrow equal(f2(A2),f2(B2)))\ \&$
$(\forall\ G4\ H4.\ equal(G4::'a,H4) \longrightarrow equal(f3(G4),f3(H4)))\ \&$
$(\forall\ O7\ P7\ Q7.\ equal(O7::'a,P7) \longrightarrow equal(f4(O7::'a,Q7),f4(P7::'a,Q7)))\ \&$
$(\forall\ R7\ T7\ S7.\ equal(R7::'a,S7) \longrightarrow equal(f4(T7::'a,R7),f4(T7::'a,S7)))\ \&$
$(\forall\ U7\ V7\ W7.\ equal(U7::'a,V7) \longrightarrow equal(f5(U7::'a,W7),f5(V7::'a,W7)))\ \&$
$(\forall\ X7\ Z7\ Y7.\ equal(X7::'a,Y7) \longrightarrow equal(f5(Z7::'a,X7),f5(Z7::'a,Y7)))\ \&$
$(\forall\ A8\ B8\ C8.\ equal(A8::'a,B8) \longrightarrow equal(f6(A8::'a,C8),f6(B8::'a,C8)))\ \&$
$(\forall\ D8\ F8\ E8.\ equal(D8::'a,E8) \longrightarrow equal(f6(F8::'a,D8),f6(F8::'a,E8)))\ \&$
$(\forall\ G8\ H8\ I8.\ equal(G8::'a,H8) \longrightarrow equal(f7(G8::'a,I8),f7(H8::'a,I8)))\ \&$
$(\forall\ J8\ L8\ K8.\ equal(J8::'a,K8) \longrightarrow equal(f7(L8::'a,J8),f7(L8::'a,K8)))\ \&$
$(\forall\ M8\ N8\ O8.\ equal(M8::'a,N8) \longrightarrow equal(f8(M8::'a,O8),f8(N8::'a,O8)))\ \&$
$(\forall\ P8\ R8\ Q8.\ equal(P8::'a,Q8) \longrightarrow equal(f8(R8::'a,P8),f8(R8::'a,Q8)))\ \&$
$(\forall\ S8\ T8\ U8.\ equal(S8::'a,T8) \longrightarrow equal(f9(S8::'a,U8),f9(T8::'a,U8)))\ \&$
$(\forall\ V8\ X8\ W8.\ equal(V8::'a,W8) \longrightarrow equal(f9(X8::'a,V8),f9(X8::'a,W8)))\ \&$
$(\forall\ G\ H\ I'.\ equal(G::'a,H) \longrightarrow equal(f10(G::'a,I'),f10(H::'a,I')))\ \&$
$(\forall\ J\ L\ K'.\ equal(J::'a,K') \longrightarrow equal(f10(L::'a,J),f10(L::'a,K')))\ \&$
$(\forall\ M\ N\ O'.\ equal(M::'a,N) \longrightarrow equal(f11(M::'a,O'),f11(N::'a,O')))\ \&$
$(\forall\ P\ R\ Q.\ equal(P::'a,Q) \longrightarrow equal(f11(R::'a,P),f11(R::'a,Q)))\ \&$

$$
\begin{array}{l}
(\forall\, S'\ T'\ U.\ equal(S'::'a,T') \longrightarrow equal(f12(S'::'a,U),f12(T'::'a,U))) \ \&\\
(\forall\, V\ X\ W.\ equal(V::'a,W) \longrightarrow equal(f12(X::'a,V),f12(X::'a,W))) \ \&\\
(\forall\, Y\ Z\ A1.\ equal(Y::'a,Z) \longrightarrow equal(f13(Y::'a,A1),f13(Z::'a,A1))) \ \&\\
(\forall\, B1\ D1\ C1.\ equal(B1::'a,C1) \longrightarrow equal(f13(D1::'a,B1),f13(D1::'a,C1))) \ \&\\
(\forall\, E1\ F1\ G1.\ equal(E1::'a,F1) \longrightarrow equal(f14(E1::'a,G1),f14(F1::'a,G1))) \ \&\\
(\forall\, H1\ J1\ I1.\ equal(H1::'a,I1) \longrightarrow equal(f14(J1::'a,H1),f14(J1::'a,I1))) \ \&\\
(\forall\, K1\ L1\ M1.\ equal(K1::'a,L1) \longrightarrow equal(f16(K1::'a,M1),f16(L1::'a,M1))) \ \&\\
(\forall\, N1\ P1\ O1.\ equal(N1::'a,O1) \longrightarrow equal(f16(P1::'a,N1),f16(P1::'a,O1))) \ \&\\
(\forall\, Q1\ R1\ S1.\ equal(Q1::'a,R1) \longrightarrow equal(f17(Q1::'a,S1),f17(R1::'a,S1))) \ \&\\
(\forall\, T1\ V1\ U1.\ equal(T1::'a,U1) \longrightarrow equal(f17(V1::'a,T1),f17(V1::'a,U1))) \ \&\\
(\forall\, W1\ X1.\ equal(W1::'a,X1) \longrightarrow equal(f18(W1),f18(X1))) \ \&\\
(\forall\, Y1\ Z1.\ equal(Y1::'a,Z1) \longrightarrow equal(f19(Y1),f19(Z1))) \ \&\\
(\forall\, C2\ D2.\ equal(C2::'a,D2) \longrightarrow equal(f20(C2),f20(D2))) \ \&\\
(\forall\, E2\ F2.\ equal(E2::'a,F2) \longrightarrow equal(f21(E2),f21(F2))) \ \&\\
(\forall\, G2\ H2\ I2\ J2.\ equal(G2::'a,H2) \longrightarrow equal(f22(G2::'a,I2,J2),f22(H2::'a,I2,J2)))\\
\&\\
(\forall\, K2\ M2\ L2\ N2.\ equal(K2::'a,L2) \longrightarrow equal(f22(M2::'a,K2,N2),f22(M2::'a,L2,N2)))\\
\&\\
(\forall\, O2\ Q2\ R2\ P2.\ equal(O2::'a,P2) \longrightarrow equal(f22(Q2::'a,R2,O2),f22(Q2::'a,R2,P2)))\\
\&\\
(\forall\, S2\ T2\ U2.\ equal(S2::'a,T2) \longrightarrow equal(f23(S2::'a,U2),f23(T2::'a,U2))) \ \&\\
(\forall\, V2\ X2\ W2.\ equal(V2::'a,W2) \longrightarrow equal(f23(X2::'a,V2),f23(X2::'a,W2)))\\
\&\\
(\forall\, Y2\ Z2.\ equal(Y2::'a,Z2) \longrightarrow equal(f24(Y2),f24(Z2))) \ \&\\
(\forall\, A3\ B3.\ equal(A3::'a,B3) \longrightarrow equal(f26(A3),f26(B3))) \ \&\\
(\forall\, C3\ D3\ E3.\ equal(C3::'a,D3) \longrightarrow equal(f27(C3::'a,E3),f27(D3::'a,E3))) \ \&\\
(\forall\, F3\ H3\ G3.\ equal(F3::'a,G3) \longrightarrow equal(f27(H3::'a,F3),f27(H3::'a,G3))) \ \&\\
(\forall\, I3\ J3\ K3\ L3.\ equal(I3::'a,J3) \longrightarrow equal(f28(I3::'a,K3,L3),f28(J3::'a,K3,L3)))\\
\&\\
(\forall\, M3\ O3\ N3\ P3.\ equal(M3::'a,N3) \longrightarrow equal(f28(O3::'a,M3,P3),f28(O3::'a,N3,P3)))\\
\&\\
(\forall\, Q3\ S3\ T3\ R3.\ equal(Q3::'a,R3) \longrightarrow equal(f28(S3::'a,T3,Q3),f28(S3::'a,T3,R3)))\\
\&\\
(\forall\, U3\ V3\ W3\ X3.\ equal(U3::'a,V3) \longrightarrow equal(f29(U3::'a,W3,X3),f29(V3::'a,W3,X3)))\\
\&\\
(\forall\, Y3\ A4\ Z3\ B4.\ equal(Y3::'a,Z3) \longrightarrow equal(f29(A4::'a,Y3,B4),f29(A4::'a,Z3,B4)))\\
\&\\
(\forall\, C4\ E4\ F4\ D4.\ equal(C4::'a,D4) \longrightarrow equal(f29(E4::'a,F4,C4),f29(E4::'a,F4,D4)))\\
\&\\
(\forall\, I4\ J4\ K4\ L4.\ equal(I4::'a,J4) \longrightarrow equal(f30(I4::'a,K4,L4),f30(J4::'a,K4,L4)))\\
\&\\
(\forall\, M4\ O4\ N4\ P4.\ equal(M4::'a,N4) \longrightarrow equal(f30(O4::'a,M4,P4),f30(O4::'a,N4,P4)))\\
\&\\
(\forall\, Q4\ S4\ T4\ R4.\ equal(Q4::'a,R4) \longrightarrow equal(f30(S4::'a,T4,Q4),f30(S4::'a,T4,R4)))\\
\&\\
(\forall\, U4\ V4\ W4\ X4.\ equal(U4::'a,V4) \longrightarrow equal(f31(U4::'a,W4,X4),f31(V4::'a,W4,X4)))\\
\&\\
(\forall\, Y4\ A5\ Z4\ B5.\ equal(Y4::'a,Z4) \longrightarrow equal(f31(A5::'a,Y4,B5),f31(A5::'a,Z4,B5)))\\
\&
\end{array}
$$

$$(\forall\, C5\ E5\ F5\ D5.\ equal(C5::'a,D5) \longrightarrow equal(f31(E5::'a,F5,C5),f31(E5::'a,F5,D5)))$$
$$\&$$
$$(\forall\, G5\ H5\ I5\ J5\ K5\ L5.\ equal(G5::'a,H5) \longrightarrow equal(f32(G5::'a,I5,J5,K5,L5),f32(H5::'a,I5,J5,K5,L5)))$$
$$\&$$
$$(\forall\, M5\ O5\ N5\ P5\ Q5\ R5.\ equal(M5::'a,N5) \longrightarrow equal(f32(O5::'a,M5,P5,Q5,R5),f32(O5::'a,N5,P5,Q5,R5$$
$$\&$$
$$(\forall\, S5\ U5\ V5\ T5\ W5\ X5.\ equal(S5::'a,T5) \longrightarrow equal(f32(U5::'a,V5,S5,W5,X5),f32(U5::'a,V5,T5,W5,X5)$$
$$\&$$
$$(\forall\, Y5\ A6\ B6\ C6\ Z5\ D6.\ equal(Y5::'a,Z5) \longrightarrow equal(f32(A6::'a,B6,C6,Y5,D6),f32(A6::'a,B6,C6,Z5,D6)))$$
$$\&$$
$$(\forall\, E6\ G6\ H6\ I6\ J6\ F6.\ equal(E6::'a,F6) \longrightarrow equal(f32(G6::'a,H6,I6,J6,E6),f32(G6::'a,H6,I6,J6,F6)))$$
$$\&$$
$$(\forall\, K6\ L6\ M6\ N6\ O6\ P6.\ equal(K6::'a,L6) \longrightarrow equal(f33(K6::'a,M6,N6,O6,P6),f33(L6::'a,M6,N6,O6,P6)$$
$$\&$$
$$(\forall\, Q6\ S6\ R6\ T6\ U6\ V6.\ equal(Q6::'a,R6) \longrightarrow equal(f33(S6::'a,Q6,T6,U6,V6),f33(S6::'a,R6,T6,U6,V6)))$$
$$\&$$
$$(\forall\, W6\ Y6\ Z6\ X6\ A7\ B7.\ equal(W6::'a,X6) \longrightarrow equal(f33(Y6::'a,Z6,W6,A7,B7),f33(Y6::'a,Z6,X6,A7,B7)$$
$$\&$$
$$(\forall\, C7\ E7\ F7\ G7\ D7\ H7.\ equal(C7::'a,D7) \longrightarrow equal(f33(E7::'a,F7,G7,C7,H7),f33(E7::'a,F7,G7,D7,H7)))$$
$$\&$$
$$(\forall\, I7\ K7\ L7\ M7\ N7\ J7.\ equal(I7::'a,J7) \longrightarrow equal(f33(K7::'a,L7,M7,N7,I7),f33(K7::'a,L7,M7,N7,J7)))$$
$$\&$$
$$(\forall\, A\ B\ C.\ equal(A::'a,B) \longrightarrow equal(apply(A::'a,C),apply(B::'a,C))) \ \&$$
$$(\forall\, D\ F'\ E.\ equal(D::'a,E) \longrightarrow equal(apply(F'::'a,D),apply(F'::'a,E))) \ \&$$
$$(\forall\, G\ H\ I'\ J.\ equal(G::'a,H) \longrightarrow equal(apply\text{-}to\text{-}two\text{-}arguments(G::'a,I',J),apply\text{-}to\text{-}two\text{-}arguments(H::'a,I',$$
$$\&$$
$$(\forall\, K'\ M\ L\ N.\ equal(K'::'a,L) \longrightarrow equal(apply\text{-}to\text{-}two\text{-}arguments(M::'a,K',N),apply\text{-}to\text{-}two\text{-}arguments(M::'a$$
$$\&$$
$$(\forall\, O'\ Q\ R\ P.\ equal(O'::'a,P) \longrightarrow equal(apply\text{-}to\text{-}two\text{-}arguments(Q::'a,R,O'),apply\text{-}to\text{-}two\text{-}arguments(Q::'a,$$
$$\&$$
$$(\forall\, S'\ T'.\ equal(S'::'a,T') \longrightarrow equal(complement(S'),complement(T'))) \ \&$$
$$(\forall\, U\ V\ W.\ equal(U::'a,V) \longrightarrow equal(composition(U::'a,W),composition(V::'a,W)))$$
$$\&$$
$$(\forall\, X\ Z\ Y.\ equal(X::'a,Y) \longrightarrow equal(composition(Z::'a,X),composition(Z::'a,Y)))$$
$$\&$$
$$(\forall\, A1\ B1.\ equal(A1::'a,B1) \longrightarrow equal(inv1\ A1,inv1\ B1)) \ \&$$
$$(\forall\, C1\ D1\ E1.\ equal(C1::'a,D1) \longrightarrow equal(cross\text{-}product(C1::'a,E1),cross\text{-}product(D1::'a,E1)))$$
$$\&$$
$$(\forall\, F1\ H1\ G1.\ equal(F1::'a,G1) \longrightarrow equal(cross\text{-}product(H1::'a,F1),cross\text{-}product(H1::'a,G1)))$$
$$\&$$
$$(\forall\, I1\ J1.\ equal(I1::'a,J1) \longrightarrow equal(domain\text{-}of(I1),domain\text{-}of(J1))) \ \&$$
$$(\forall\, I10\ J10.\ equal(I10::'a,J10) \longrightarrow equal(first(I10),first(J10))) \ \&$$
$$(\forall\, Q10\ R10.\ equal(Q10::'a,R10) \longrightarrow equal(flip\text{-}range\text{-}of(Q10),flip\text{-}range\text{-}of(R10)))$$
$$\&$$
$$(\forall\, S10\ T10\ U10.\ equal(S10::'a,T10) \longrightarrow equal(image'(S10::'a,U10),image'(T10::'a,U10)))$$
$$\&$$
$$(\forall\, V10\ X10\ W10.\ equal(V10::'a,W10) \longrightarrow equal(image'(X10::'a,V10),image'(X10::'a,W10)))$$
$$\&$$
$$(\forall\, Y10\ Z10\ A11.\ equal(Y10::'a,Z10) \longrightarrow equal(intersection(Y10::'a,A11),intersection(Z10::'a,A11)))$$

$$
\begin{array}{l}
\& \\
\quad (\forall\, B11\ D11\ C11.\ equal(B11::'a,C11) \longrightarrow equal(intersection(D11::'a,B11),intersection(D11::'a,C11))) \\
\& \\
\quad (\forall\, E11\ F11\ G11.\ equal(E11::'a,F11) \longrightarrow equal(non\text{-}ordered\text{-}pair(E11::'a,G11),non\text{-}ordered\text{-}pair(F11::'a,G \\
\& \\
\quad (\forall\, H11\ J11\ I11.\ equal(H11::'a,I11) \longrightarrow equal(non\text{-}ordered\text{-}pair(J11::'a,H11),non\text{-}ordered\text{-}pair(J11::'a,I11) \\
\& \\
\quad (\forall\, K11\ L11\ M11.\ equal(K11::'a,L11) \longrightarrow equal(ordered\text{-}pair(K11::'a,M11),ordered\text{-}pair(L11::'a,M11))) \\
\& \\
\quad (\forall\, N11\ P11\ O11.\ equal(N11::'a,O11) \longrightarrow equal(ordered\text{-}pair(P11::'a,N11),ordered\text{-}pair(P11::'a,O11))) \\
\& \\
\quad (\forall\, Q11\ R11.\ equal(Q11::'a,R11) \longrightarrow equal(powerset(Q11),powerset(R11))) \ \& \\
\quad (\forall\, S11\ T11.\ equal(S11::'a,T11) \longrightarrow equal(range\text{-}of(S11),range\text{-}of(T11))) \ \& \\
\quad (\forall\, U11\ V11\ W11.\ equal(U11::'a,V11) \longrightarrow equal(restrct(U11::'a,W11),restrct(V11::'a,W11))) \\
\& \\
\quad (\forall\, X11\ Z11\ Y11.\ equal(X11::'a,Y11) \longrightarrow equal(restrct(Z11::'a,X11),restrct(Z11::'a,Y11))) \\
\& \\
\quad (\forall\, A12\ B12.\ equal(A12::'a,B12) \longrightarrow equal(rot\text{-}right(A12),rot\text{-}right(B12))) \ \& \\
\quad (\forall\, C12\ D12.\ equal(C12::'a,D12) \longrightarrow equal(second(C12),second(D12))) \ \& \\
\quad (\forall\, K12\ L12.\ equal(K12::'a,L12) \longrightarrow equal(sigma(K12),sigma(L12))) \ \& \\
\quad (\forall\, M12\ N12.\ equal(M12::'a,N12) \longrightarrow equal(singleton\text{-}set(M12),singleton\text{-}set(N12))) \\
\& \\
\quad (\forall\, O12\ P12.\ equal(O12::'a,P12) \longrightarrow equal(successor(O12),successor(P12))) \ \& \\
\quad (\forall\, Q12\ R12\ S12.\ equal(Q12::'a,R12) \longrightarrow equal(union(Q12::'a,S12),union(R12::'a,S12))) \\
\& \\
\quad (\forall\, T12\ V12\ U12.\ equal(T12::'a,U12) \longrightarrow equal(union(V12::'a,T12),union(V12::'a,U12))) \\
\& \\
\quad (\forall\, W12\ X12\ Y12.\ equal(W12::'a,X12)\ \&\ closed(W12::'a,Y12) \longrightarrow closed(X12::'a,Y12)) \\
\& \\
\quad (\forall\, Z12\ B13\ A13.\ equal(Z12::'a,A13)\ \&\ closed(B13::'a,Z12) \longrightarrow closed(B13::'a,A13)) \\
\& \\
\quad (\forall\, C13\ D13\ E13.\ equal(C13::'a,D13)\ \&\ disjoint(C13::'a,E13) \longrightarrow disjoint(D13::'a,E13)) \\
\& \\
\quad (\forall\, F13\ H13\ G13.\ equal(F13::'a,G13)\ \&\ disjoint(H13::'a,F13) \longrightarrow disjoint(H13::'a,G13)) \\
\& \\
\quad (\forall\, I13\ J13.\ equal(I13::'a,J13)\ \&\ function(I13) \longrightarrow function(J13)) \ \& \\
\quad (\forall\, K13\ L13\ M13\ N13\ O13\ P13.\ equal(K13::'a,L13)\ \&\ homomorphism(K13::'a,M13,N13,O13,P13) \\
\longrightarrow homomorphism(L13::'a,M13,N13,O13,P13)) \ \& \\
\quad (\forall\, Q13\ S13\ R13\ T13\ U13\ V13.\ equal(Q13::'a,R13)\ \&\ homomorphism(S13::'a,Q13,T13,U13,V13) \\
\longrightarrow homomorphism(S13::'a,R13,T13,U13,V13)) \ \& \\
\quad (\forall\, W13\ Y13\ Z13\ X13\ A14\ B14.\ equal(W13::'a,X13)\ \&\ homomorphism(Y13::'a,Z13,W13,A14,B14) \\
\longrightarrow homomorphism(Y13::'a,Z13,X13,A14,B14)) \ \& \\
\quad (\forall\, C14\ E14\ F14\ G14\ D14\ H14.\ equal(C14::'a,D14)\ \&\ homomorphism(E14::'a,F14,G14,C14,H14) \\
\longrightarrow homomorphism(E14::'a,F14,G14,D14,H14)) \ \& \\
\quad (\forall\, I14\ K14\ L14\ M14\ N14\ J14.\ equal(I14::'a,J14)\ \&\ homomorphism(K14::'a,L14,M14,N14,I14) \\
\longrightarrow homomorphism(K14::'a,L14,M14,N14,J14)) \ \& \\
\quad (\forall\, O14\ P14.\ equal(O14::'a,P14)\ \&\ little\text{-}set(O14) \longrightarrow little\text{-}set(P14)) \ \& \\
\quad (\forall\, Q14\ R14\ S14\ T14.\ equal(Q14::'a,R14)\ \&\ maps(Q14::'a,S14,T14) \longrightarrow maps(R14::'a,S14,T14)) \\
\&
\end{array}
$$

(∀ U14 W14 V14 X14. equal(U14::'a,V14) & maps(W14::'a,U14,X14) −−>
maps(W14::'a,V14,X14)) &
(∀ Y14 A15 B15 Z14. equal(Y14::'a,Z14) & maps(A15::'a,B15,Y14) −−> maps(A15::'a,B15,Z14))
&
(∀ C15 D15 E15. equal(C15::'a,D15) & member(C15::'a,E15) −−> member(D15::'a,E15))
&
(∀ F15 H15 G15. equal(F15::'a,G15) & member(H15::'a,F15) −−> member(H15::'a,G15))
&
(∀ I15 J15. equal(I15::'a,J15) & one-to-one-function(I15) −−> one-to-one-function(J15))
&
(∀ K15 L15. equal(K15::'a,L15) & ordered-pair-predicate(K15) −−> ordered-pair-predicate(L15))
&
(∀ M15 N15 O15. equal(M15::'a,N15) & proper-subset(M15::'a,O15) −−> proper-subset(N15::'a,O15))
&
(∀ P15 R15 Q15. equal(P15::'a,Q15) & proper-subset(R15::'a,P15) −−> proper-subset(R15::'a,Q15))
&
(∀ S15 T15. equal(S15::'a,T15) & relation(S15) −−> relation(T15)) &
(∀ U15 V15. equal(U15::'a,V15) & single-valued-set(U15) −−> single-valued-set(V15))
&
(∀ W15 X15 Y15. equal(W15::'a,X15) & ssubset(W15::'a,Y15) −−> ssubset(X15::'a,Y15))
&
(∀ Z15 B16 A16. equal(Z15::'a,A16) & ssubset(B16::'a,Z15) −−> ssubset(B16::'a,A16))
&
(~little-set(ordered-pair(a::'a,b))) −−> False
⟨proof⟩

lemma *SET046-5*:
(∀ Y X. ~(element(X::'a,a) & element(X::'a,Y) & element(Y::'a,X))) &
(∀ X. element(X::'a,f(X)) | element(X::'a,a)) &
(∀ X. element(f(X),X) | element(X::'a,a)) −−> False
⟨proof⟩

lemma *SET047-5*:
(∀ X Z Y. set-equal(X::'a,Y) & element(Z::'a,X) −−> element(Z::'a,Y)) &
(∀ Y Z X. set-equal(X::'a,Y) & element(Z::'a,Y) −−> element(Z::'a,X)) &
(∀ X Y. element(f(X::'a,Y),X) | element(f(X::'a,Y),Y) | set-equal(X::'a,Y))
&
(∀ X Y. element(f(X::'a,Y),Y) & element(f(X::'a,Y),X) −−> set-equal(X::'a,Y))
&
(set-equal(a::'a,b) | set-equal(b::'a,a)) &
(~(set-equal(b::'a,a) & set-equal(a::'a,b))) −−> False
⟨proof⟩

lemma *SYN034-1*:
(∀ A. p(A::'a,a) | p(A::'a,f(A))) &

$(\forall A.\ p(A::'a,a) \mid p(f(A),A))$ &
$(\forall A\ B.\ \sim(p(A::'a,B)\ \&\ p(B::'a,A)\ \&\ p(B::'a,a))) \longrightarrow False$
⟨*proof*⟩

lemma *SYN071-1*:
EQU001-0-ax equal &
$(equal(a::'a,b) \mid equal(c::'a,d))$ &
$(equal(a::'a,c) \mid equal(b::'a,d))$ &
$(\sim equal(a::'a,d))$ &
$(\sim equal(b::'a,c)) \longrightarrow False$
⟨*proof*⟩

lemma *SYN349-1*:
$(\forall X\ Y.\ f(w(X),g(X::'a,Y)) \longrightarrow f(X::'a,g(X::'a,Y)))$ &
$(\forall X\ Y.\ f(X::'a,g(X::'a,Y)) \longrightarrow f(w(X),g(X::'a,Y)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y))\ \&\ f(Y::'a,g(X::'a,Y)) \longrightarrow f(g(X::'a,Y),Y) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(g(X::'a,Y),Y)\ \&\ f(Y::'a,g(X::'a,Y)) \longrightarrow f(X::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),Y) \mid f(Y::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),Y) \longrightarrow f(Y::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),w(X)) \longrightarrow f(g(X::'a,Y),Y) \mid f(Y::'a,g(X::'a,Y)))$ &
$(\forall Y\ X.\ f(g(X::'a,Y),Y)\ \&\ f(g(X::'a,Y),w(X)) \longrightarrow f(X::'a,g(X::'a,Y)) \mid f(Y::'a,g(X::'a,Y)))$ &
$(\forall Y\ X.\ f(Y::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),w(X)) \longrightarrow f(X::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),Y))$ &
$(\forall Y\ X.\ \sim(f(X::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),Y)\ \&\ f(Y::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),w(X)))) \longrightarrow False$
⟨*proof*⟩

lemma *SYN352-1*:
$(f(a::'a,b))$ &
$(\forall X\ Y.\ f(X::'a,Y) \longrightarrow f(b::'a,z(X::'a,Y)) \mid f(Y::'a,z(X::'a,Y)))$ &
$(\forall X\ Y.\ f(X::'a,Y) \mid f(z(X::'a,Y),z(X::'a,Y)))$ &
$(\forall X\ Y.\ f(b::'a,z(X::'a,Y)) \mid f(X::'a,z(X::'a,Y)) \mid f(z(X::'a,Y),z(X::'a,Y)))$ &
$(\forall X\ Y.\ f(b::'a,z(X::'a,Y))\ \&\ f(X::'a,z(X::'a,Y)) \longrightarrow f(z(X::'a,Y),z(X::'a,Y)))$ &
$(\forall X\ Y.\ \sim(f(X::'a,Y)\ \&\ f(X::'a,z(X::'a,Y))\ \&\ f(Y::'a,z(X::'a,Y))))$ &
$(\forall X\ Y.\ f(X::'a,Y) \longrightarrow f(X::'a,z(X::'a,Y)) \mid f(Y::'a,z(X::'a,Y))) \longrightarrow False$
⟨*proof*⟩

lemma *TOP001-2*:

$(\forall\ Vf\ U.\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf))\ \longrightarrow\ element\text{-}of\text{-}set(U::'a,f1(Vf::'a,U)))$
&
$(\forall\ U\ Vf.\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf))\ \longrightarrow\ element\text{-}of\text{-}collection(f1(Vf::'a,U),Vf))$
&
$(\forall\ U\ \ Uu1\ \ Vf.\ \ element\text{-}of\text{-}set(U::'a,Uu1)\ \ \&\ \ element\text{-}of\text{-}collection(Uu1::'a,Vf)$
$\longrightarrow\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf)))$ &
$(\forall\ Vf\ X.\ basis(X::'a,Vf)\ \longrightarrow\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X))$ &
$(\forall\ Vf\ U\ X.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \&\ element\text{-}of\text{-}set(X::'a,U)$
$\longrightarrow\ element\text{-}of\text{-}set(X::'a,f10(Vf::'a,U,X)))$ &
$(\forall\ U\ X\ Vf.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \&\ element\text{-}of\text{-}set(X::'a,U)$
$\longrightarrow\ element\text{-}of\text{-}collection(f10(Vf::'a,U,X),Vf))$ &
$(\forall\ X.\ subset\text{-}sets(X::'a,X))$ &
$(\forall\ X\ U\ Y.\ subset\text{-}sets(X::'a,Y)\ \&\ element\text{-}of\text{-}set(U::'a,X)\ \longrightarrow\ element\text{-}of\text{-}set(U::'a,Y))$
&
$(\forall\ X\ \ Y.\ equal\text{-}sets(X::'a,Y)\ \longrightarrow\ subset\text{-}sets(X::'a,Y))$ &
$(\forall\ Y\ X.\ subset\text{-}sets(X::'a,Y)\ \mid\ element\text{-}of\text{-}set(in\text{-}1st\text{-}set(X::'a,Y),X))$ &
$(\forall\ X\ \ Y.\ element\text{-}of\text{-}set(in\text{-}1st\text{-}set(X::'a,Y),Y)\ \longrightarrow\ subset\text{-}sets(X::'a,Y))$ &
$(basis(cx::'a,f))$ &
$(\sim subset\text{-}sets(union\text{-}of\text{-}members(top\text{-}of\text{-}basis(f)),cx)\ \longrightarrow\ False$
$\langle proof\rangle$

lemma *TOP002-2*:
$(\forall\ Vf\ U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \mid\ element\text{-}of\text{-}set(f11(Vf::'a,U),U))$
&
$(\forall\ X.\ \sim element\text{-}of\text{-}set(X::'a,empty\text{-}set))$ &
$(\sim element\text{-}of\text{-}collection(empty\text{-}set::'a,top\text{-}of\text{-}basis(f)))\ \longrightarrow\ False$
$\langle proof\rangle$

lemma *TOP004-1*:
$(\forall\ Vf\ U.\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf))\ \longrightarrow\ element\text{-}of\text{-}set(U::'a,f1(Vf::'a,U)))$
&
$(\forall\ U\ Vf.\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf))\ \longrightarrow\ element\text{-}of\text{-}collection(f1(Vf::'a,U),Vf))$
&
$(\forall\ U\ \ Uu1\ \ Vf.\ \ element\text{-}of\text{-}set(U::'a,Uu1)\ \ \&\ \ element\text{-}of\text{-}collection(Uu1::'a,Vf)$
$\longrightarrow\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf)))$ &
$(\forall\ Vf\ U\ Va.\ element\text{-}of\text{-}set(U::'a,intersection\text{-}of\text{-}members(Vf))\ \&\ element\text{-}of\text{-}collection(Va::'a,Vf)$
$\longrightarrow\ element\text{-}of\text{-}set(U::'a,Va))$ &
$(\forall\ U\ Vf.\ element\text{-}of\text{-}set(U::'a,intersection\text{-}of\text{-}members(Vf))\ \mid\ element\text{-}of\text{-}collection(f2(Vf::'a,U),Vf))$
&
$(\forall\ Vf\ U.\ element\text{-}of\text{-}set(U::'a,f2(Vf::'a,U))\ \longrightarrow\ element\text{-}of\text{-}set(U::'a,intersection\text{-}of\text{-}members(Vf)))$
&
$(\forall\ Vt\ X.\ topological\text{-}space(X::'a,Vt)\ \longrightarrow\ equal\text{-}sets(union\text{-}of\text{-}members(Vt),X))$
&
$(\forall\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \longrightarrow\ element\text{-}of\text{-}collection(empty\text{-}set::'a,Vt))$
&
$(\forall\ X\ \ Vt.\ topological\text{-}space(X::'a,Vt)\ \longrightarrow\ element\text{-}of\text{-}collection(X::'a,Vt))$ &
$(\forall\ X\ \ Y\ \ Z\ \ Vt.\ topological\text{-}space(X::'a,Vt)\ \ \&\ \ element\text{-}of\text{-}collection(Y::'a,Vt)\ \ \&$

$element\text{-}of\text{-}collection(Z::'a,Vt) \longrightarrow element\text{-}of\text{-}collection(intersection\text{-}of\text{-}sets(Y::'a,Z),Vt))$
&
$(\forall X\ Vf\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}collections(Vf::'a,Vt) \longrightarrow$
$element\text{-}of\text{-}collection(union\text{-}of\text{-}members(Vf),Vt))\ \&$
$(\forall X\ Vt.\ equal\text{-}sets(union\text{-}of\text{-}members(Vt),X)\ \&\ element\text{-}of\text{-}collection(empty\text{-}set::'a,Vt)$
$\&\ element\text{-}of\text{-}collection(X::'a,Vt) \longrightarrow topological\text{-}space(X::'a,Vt) \mid element\text{-}of\text{-}collection(f3(X::'a,Vt),Vt)$
$\mid subset\text{-}collections(f5(X::'a,Vt),Vt))\ \&$
$(\forall X\ Vt.\ equal\text{-}sets(union\text{-}of\text{-}members(Vt),X)\ \&\ element\text{-}of\text{-}collection(empty\text{-}set::'a,Vt)$
$\&\ element\text{-}of\text{-}collection(X::'a,Vt)\ \&\ element\text{-}of\text{-}collection(union\text{-}of\text{-}members(f5(X::'a,Vt)),Vt)$
$\longrightarrow topological\text{-}space(X::'a,Vt) \mid element\text{-}of\text{-}collection(f3(X::'a,Vt),Vt))\ \&$
$(\forall X\ Vt.\ equal\text{-}sets(union\text{-}of\text{-}members(Vt),X)\ \&\ element\text{-}of\text{-}collection(empty\text{-}set::'a,Vt)$
$\&\ element\text{-}of\text{-}collection(X::'a,Vt) \longrightarrow topological\text{-}space(X::'a,Vt) \mid element\text{-}of\text{-}collection(f4(X::'a,Vt),Vt)$
$\mid subset\text{-}collections(f5(X::'a,Vt),Vt))\ \&$
$(\forall X\ Vt.\ equal\text{-}sets(union\text{-}of\text{-}members(Vt),X)\ \&\ element\text{-}of\text{-}collection(empty\text{-}set::'a,Vt)$
$\&\ element\text{-}of\text{-}collection(X::'a,Vt)\ \&\ element\text{-}of\text{-}collection(union\text{-}of\text{-}members(f5(X::'a,Vt)),Vt)$
$\longrightarrow topological\text{-}space(X::'a,Vt) \mid element\text{-}of\text{-}collection(f4(X::'a,Vt),Vt))\ \&$
$(\forall X\ Vt.\ equal\text{-}sets(union\text{-}of\text{-}members(Vt),X)\ \&\ element\text{-}of\text{-}collection(empty\text{-}set::'a,Vt)$
$\&\ element\text{-}of\text{-}collection(X::'a,Vt)\ \&\ element\text{-}of\text{-}collection(intersection\text{-}of\text{-}sets(f3(X::'a,Vt),f4(X::'a,Vt)),Vt)$
$\longrightarrow topological\text{-}space(X::'a,Vt) \mid subset\text{-}collections(f5(X::'a,Vt),Vt))\ \&$
$(\forall X\ Vt.\ equal\text{-}sets(union\text{-}of\text{-}members(Vt),X)\ \&\ element\text{-}of\text{-}collection(empty\text{-}set::'a,Vt)$
$\&\ element\text{-}of\text{-}collection(X::'a,Vt)\ \&\ element\text{-}of\text{-}collection(intersection\text{-}of\text{-}sets(f3(X::'a,Vt),f4(X::'a,Vt)),Vt)$
$\&\ element\text{-}of\text{-}collection(union\text{-}of\text{-}members(f5(X::'a,Vt)),Vt) \longrightarrow topological\text{-}space(X::'a,Vt))$
&
$(\forall U\ X\ Vt.\ open(U::'a,X,Vt) \longrightarrow topological\text{-}space(X::'a,Vt))\ \&$
$(\forall X\ U\ Vt.\ open(U::'a,X,Vt) \longrightarrow element\text{-}of\text{-}collection(U::'a,Vt))\ \&$
$(\forall X\ U\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ element\text{-}of\text{-}collection(U::'a,Vt) \longrightarrow$
$open(U::'a,X,Vt))\ \&$
$(\forall U\ X\ Vt.\ closed(U::'a,X,Vt) \longrightarrow topological\text{-}space(X::'a,Vt))\ \&$
$(\forall U\ X\ Vt.\ closed(U::'a,X,Vt) \longrightarrow open(relative\text{-}complement\text{-}sets(U::'a,X),X,Vt))$
&
$(\forall U\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ open(relative\text{-}complement\text{-}sets(U::'a,X),X,Vt)$
$\longrightarrow closed(U::'a,X,Vt))\ \&$
$(\forall Vs\ X\ Vt.\ finer(Vt::'a,Vs,X) \longrightarrow topological\text{-}space(X::'a,Vt))\ \&$
$(\forall Vt\ X\ Vs.\ finer(Vt::'a,Vs,X) \longrightarrow topological\text{-}space(X::'a,Vs))\ \&$
$(\forall X\ Vs\ Vt.\ finer(Vt::'a,Vs,X) \longrightarrow subset\text{-}collections(Vs::'a,Vt))\ \&$
$(\forall X\ Vs\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ topological\text{-}space(X::'a,Vs)\ \&\ subset\text{-}collections(Vs::'a,Vt)$
$\longrightarrow finer(Vt::'a,Vs,X))\ \&$
$(\forall Vf\ X.\ basis(X::'a,Vf) \longrightarrow equal\text{-}sets(union\text{-}of\text{-}members(Vf),X))\ \&$
$(\forall X\ Vf\ Y\ Vb1\ Vb2.\ basis(X::'a,Vf)\ \&\ element\text{-}of\text{-}set(Y::'a,X)\ \&\ element\text{-}of\text{-}collection(Vb1::'a,Vf)$
$\&\ element\text{-}of\text{-}collection(Vb2::'a,Vf)\ \&\ element\text{-}of\text{-}set(Y::'a,intersection\text{-}of\text{-}sets(Vb1::'a,Vb2))$
$\longrightarrow element\text{-}of\text{-}set(Y::'a,f6(X::'a,Vf,Y,Vb1,Vb2)))\ \&$
$(\forall X\ Y\ Vb1\ Vb2\ Vf.\ basis(X::'a,Vf)\ \&\ element\text{-}of\text{-}set(Y::'a,X)\ \&\ element\text{-}of\text{-}collection(Vb1::'a,Vf)$
$\&\ element\text{-}of\text{-}collection(Vb2::'a,Vf)\ \&\ element\text{-}of\text{-}set(Y::'a,intersection\text{-}of\text{-}sets(Vb1::'a,Vb2))$
$\longrightarrow element\text{-}of\text{-}collection(f6(X::'a,Vf,Y,Vb1,Vb2),Vf))\ \&$
$(\forall X\ Vf\ Y\ Vb1\ Vb2.\ basis(X::'a,Vf)\ \&\ element\text{-}of\text{-}set(Y::'a,X)\ \&\ element\text{-}of\text{-}collection(Vb1::'a,Vf)$
$\&\ element\text{-}of\text{-}collection(Vb2::'a,Vf)\ \&\ element\text{-}of\text{-}set(Y::'a,intersection\text{-}of\text{-}sets(Vb1::'a,Vb2))$
$\longrightarrow subset\text{-}sets(f6(X::'a,Vf,Y,Vb1,Vb2),intersection\text{-}of\text{-}sets(Vb1::'a,Vb2)))\ \&$
$(\forall Vf\ X.\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X) \longrightarrow basis(X::'a,Vf) \mid element\text{-}of\text{-}set(f7(X::'a,Vf),X))$
&

$(\forall\, X\ Vf.\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X) \longrightarrow basis(X::'a,Vf) \mid element\text{-}of\text{-}collection(f8(X::'a,Vf),Vf)$
$\&$
$(\forall\, X\ Vf.\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X) \longrightarrow basis(X::'a,Vf) \mid element\text{-}of\text{-}collection(f9(X::'a,Vf),Vf)$
$\&$
$(\forall\, X\ Vf.\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X) \longrightarrow basis(X::'a,Vf) \mid element\text{-}of\text{-}set(f7(X::'a,Vf),intersectio$
$\&$
$(\forall\, Uu9\ X\ Vf.\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X)\ \&\ element\text{-}of\text{-}set(f7(X::'a,Vf),Uu9)$
$\&\ element\text{-}of\text{-}collection(Uu9::'a,Vf)\ \&\ subset\text{-}sets(Uu9::'a,intersection\text{-}of\text{-}sets(f8(X::'a,Vf),f9(X::'a,Vf)))$
$\longrightarrow basis(X::'a,Vf))\ \&$
$(\forall\, Vf\ U\ X.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \&\ element\text{-}of\text{-}set(X::'a,U)$
$\longrightarrow element\text{-}of\text{-}set(X::'a,f10(Vf::'a,U,X)))\ \&$
$(\forall\, U\ X\ Vf.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \&\ element\text{-}of\text{-}set(X::'a,U)$
$\longrightarrow element\text{-}of\text{-}collection(f10(Vf::'a,U,X),Vf))\ \&$
$(\forall\, Vf\ X\ U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \&\ element\text{-}of\text{-}set(X::'a,U)$
$\longrightarrow subset\text{-}sets(f10(Vf::'a,U,X),U))\ \&$
$(\forall\, Vf\ U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf)) \mid element\text{-}of\text{-}set(f11(Vf::'a,U),U))$
$\&$
$(\forall\, Vf\ Uu11\ U.\ element\text{-}of\text{-}set(f11(Vf::'a,U),Uu11)\ \&\ element\text{-}of\text{-}collection(Uu11::'a,Vf)$
$\&\ subset\text{-}sets(Uu11::'a,U) \longrightarrow element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf)))\ \&$
$(\forall\, U\ Y\ X\ Vt.\ element\text{-}of\text{-}collection(U::'a,subspace\text{-}topology(X::'a,Vt,Y)) \longrightarrow$
$topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\, U\ Vt\ Y\ X.\ element\text{-}of\text{-}collection(U::'a,subspace\text{-}topology(X::'a,Vt,Y)) \longrightarrow$
$subset\text{-}sets(Y::'a,X))\ \&$
$(\forall\, X\ Y\ U\ Vt.\ element\text{-}of\text{-}collection(U::'a,subspace\text{-}topology(X::'a,Vt,Y)) \longrightarrow$
$element\text{-}of\text{-}collection(f12(X::'a,Vt,Y,U),Vt))\ \&$
$(\forall\, X\ Vt\ Y\ U.\ element\text{-}of\text{-}collection(U::'a,subspace\text{-}topology(X::'a,Vt,Y)) \longrightarrow$
$equal\text{-}sets(U::'a,intersection\text{-}of\text{-}sets(Y::'a,f12(X::'a,Vt,Y,U))))\ \&$
$(\forall\, X\ Vt\ U\ Y\ Uu12.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Y::'a,X)\ \&\ element\text{-}of\text{-}collection(Uu12::'a,Vt)$
$\&\ equal\text{-}sets(U::'a,intersection\text{-}of\text{-}sets(Y::'a,Uu12)) \longrightarrow element\text{-}of\text{-}collection(U::'a,subspace\text{-}topology(X::'a$
$\&$
$(\forall\, U\ Y\ X\ Vt.\ element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt)) \longrightarrow topological\text{-}space(X::'a,Vt))$
$\&$
$(\forall\, U\ Vt\ Y\ X.\ element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt)) \longrightarrow subset\text{-}sets(Y::'a,X))$
$\&$
$(\forall\, Y\ X\ Vt\ U.\ element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt)) \longrightarrow element\text{-}of\text{-}set(U::'a,f13(Y::'a,X,Vt,U)))$
$\&$
$(\forall\, X\ Vt\ U\ Y.\ element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt)) \longrightarrow subset\text{-}sets(f13(Y::'a,X,Vt,U),Y))$
$\&$
$(\forall\, Y\ U\ X\ Vt.\ element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt)) \longrightarrow open(f13(Y::'a,X,Vt,U),X,Vt))$
$\&$
$(\forall\, U\ Y\ Uu13\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Y::'a,X)\ \&\ element\text{-}of\text{-}set(U::'a,Uu13)$
$\&\ subset\text{-}sets(Uu13::'a,Y)\ \&\ open(Uu13::'a,X,Vt) \longrightarrow element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt)))$
$\&$
$(\forall\, U\ Y\ X\ Vt.\ element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,Vt)) \longrightarrow topological\text{-}space(X::'a,Vt))$
$\&$
$(\forall\, U\ Vt\ Y\ X.\ element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,Vt)) \longrightarrow subset\text{-}sets(Y::'a,X))$
$\&$
$(\forall\, Y\ X\ Vt\ U\ V.\ element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,Vt))\ \&\ subset\text{-}sets(Y::'a,V)$
$\&\ closed(V::'a,X,Vt) \longrightarrow element\text{-}of\text{-}set(U::'a,V))\ \&$

(∀ Y X Vt U. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) −−> element-of-set(U::'a,closure(Y::'a,X,
| subset-sets(Y::'a,f14(Y::'a,X,Vt,U))) &
(∀ Y U X Vt. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) −−> element-of-set(U::'a,closure(Y::'a,X,
| closed(f14(Y::'a,X,Vt,U),X,Vt)) &
(∀ Y X Vt U. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) & element-of-set(U::'a,f14(Y::'a,X,Vt,U))
−−> element-of-set(U::'a,closure(Y::'a,X,Vt))) &
(∀ U Y X Vt. neighborhood(U::'a,Y,X,Vt) −−> topological-space(X::'a,Vt)) &
(∀ Y U X Vt. neighborhood(U::'a,Y,X,Vt) −−> open(U::'a,X,Vt)) &
(∀ X Vt Y U. neighborhood(U::'a,Y,X,Vt) −−> element-of-set(Y::'a,U)) &
(∀ X Vt Y U. topological-space(X::'a,Vt) & open(U::'a,X,Vt) & element-of-set(Y::'a,U)
−−> neighborhood(U::'a,Y,X,Vt)) &
(∀ Z Y X Vt. limit-point(Z::'a,Y,X,Vt) −−> topological-space(X::'a,Vt)) &
(∀ Z Vt Y X. limit-point(Z::'a,Y,X,Vt) −−> subset-sets(Y::'a,X)) &
(∀ Z X Vt U Y. limit-point(Z::'a,Y,X,Vt) & neighborhood(U::'a,Z,X,Vt) −−>
element-of-set(f15(Z::'a,Y,X,Vt,U),intersection-of-sets(U::'a,Y))) &
(∀ Y X Vt U Z. ~(limit-point(Z::'a,Y,X,Vt) & neighborhood(U::'a,Z,X,Vt) &
eq-p(f15(Z::'a,Y,X,Vt,U),Z))) &
(∀ Y Z X Vt. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) −−> limit-point(Z::'a,Y,X,Vt)
| neighborhood(f16(Z::'a,Y,X,Vt),Z,X,Vt)) &
(∀ X Vt Y Uu16 Z. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) & element-of-set(Uu16::'a,intersection
−−> limit-point(Z::'a,Y,X,Vt) | eq-p(Uu16::'a,Z)) &
(∀ U Y X Vt. element-of-set(U::'a,boundary(Y::'a,X,Vt)) −−> topological-space(X::'a,Vt))
&
(∀ U Y X Vt. element-of-set(U::'a,boundary(Y::'a,X,Vt)) −−> element-of-set(U::'a,closure(Y::'a,X,Vt)))
&
(∀ U Y X Vt. element-of-set(U::'a,boundary(Y::'a,X,Vt)) −−> element-of-set(U::'a,closure(relative-comple
&
(∀ U Y X Vt. topological-space(X::'a,Vt) & element-of-set(U::'a,closure(Y::'a,X,Vt))
& element-of-set(U::'a,closure(relative-complement-sets(Y::'a,X),X,Vt)) −−> element-of-set(U::'a,boundary
&
(∀ X Vt. hausdorff(X::'a,Vt) −−> topological-space(X::'a,Vt)) &
(∀ X-2 X-1 X Vt. hausdorff(X::'a,Vt) & element-of-set(X-1::'a,X) & element-of-set(X-2::'a,X)
−−> eq-p(X-1::'a,X-2) | neighborhood(f17(X::'a,Vt,X-1,X-2),X-1,X,Vt)) &
(∀ X-1 X-2 X Vt. hausdorff(X::'a,Vt) & element-of-set(X-1::'a,X) & element-of-set(X-2::'a,X)
−−> eq-p(X-1::'a,X-2) | neighborhood(f18(X::'a,Vt,X-1,X-2),X-2,X,Vt)) &
(∀ X Vt X-1 X-2. hausdorff(X::'a,Vt) & element-of-set(X-1::'a,X) & element-of-set(X-2::'a,X)
−−> eq-p(X-1::'a,X-2) | disjoint-s(f17(X::'a,Vt,X-1,X-2),f18(X::'a,Vt,X-1,X-2)))
&
(∀ Vt X. topological-space(X::'a,Vt) −−> hausdorff(X::'a,Vt) | element-of-set(f19(X::'a,Vt),X))
&
(∀ Vt X. topological-space(X::'a,Vt) −−> hausdorff(X::'a,Vt) | element-of-set(f20(X::'a,Vt),X))
&
(∀ X Vt. topological-space(X::'a,Vt) & eq-p(f19(X::'a,Vt),f20(X::'a,Vt)) −−>
hausdorff(X::'a,Vt)) &
(∀ X Vt Uu19 Uu20. topological-space(X::'a,Vt) & neighborhood(Uu19::'a,f19(X::'a,Vt),X,Vt)
& neighborhood(Uu20::'a,f20(X::'a,Vt),X,Vt) & disjoint-s(Uu19::'a,Uu20) −−>
hausdorff(X::'a,Vt)) &
(∀ Va1 Va2 X Vt. separation(Va1::'a,Va2,X,Vt) −−> topological-space(X::'a,Vt))
&

$(\forall\ Va2\ X\ Vt\ Va1.\ \sim(separation(Va1::'a,Va2,X,Vt)\ \&\ equal\text{-}sets(Va1::'a,empty\text{-}set)))$
$\&$
$(\forall\ Va1\ X\ Vt\ Va2.\ \sim(separation(Va1::'a,Va2,X,Vt)\ \&\ equal\text{-}sets(Va2::'a,empty\text{-}set)))$
$\&$
$(\forall\ Va2\ X\ Va1\ Vt.\ separation(Va1::'a,Va2,X,Vt) \longrightarrow element\text{-}of\text{-}collection(Va1::'a,Vt))$
$\&$
$(\forall\ Va1\ X\ Va2\ Vt.\ separation(Va1::'a,Va2,X,Vt) \longrightarrow element\text{-}of\text{-}collection(Va2::'a,Vt))$
$\&$
$(\forall\ Vt\ Va1\ Va2\ X.\ separation(Va1::'a,Va2,X,Vt) \longrightarrow equal\text{-}sets(union\text{-}of\text{-}sets(Va1::'a,Va2),X))$
$\&$
$(\forall\ X\ Vt\ Va1\ Va2.\ separation(Va1::'a,Va2,X,Vt) \longrightarrow disjoint\text{-}s(Va1::'a,Va2))$
$\&$
$(\forall\ Vt\ X\ Va1\ Va2.\ topological\text{-}space(X::'a,Vt)\ \&\ element\text{-}of\text{-}collection(Va1::'a,Vt)$
$\&\ element\text{-}of\text{-}collection(Va2::'a,Vt)\ \&\ equal\text{-}sets(union\text{-}of\text{-}sets(Va1::'a,Va2),X)\ \&$
$disjoint\text{-}s(Va1::'a,Va2) \longrightarrow separation(Va1::'a,Va2,X,Vt) \mid equal\text{-}sets(Va1::'a,empty\text{-}set)$
$\mid equal\text{-}sets(Va2::'a,empty\text{-}set))\ \&$
$(\forall\ X\ Vt.\ connected\text{-}space(X::'a,Vt) \longrightarrow topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ Va1\ Va2\ X\ Vt.\ \sim(connected\text{-}space(X::'a,Vt)\ \&\ separation(Va1::'a,Va2,X,Vt)))$
$\&$
$(\forall\ X\ Vt.\ topological\text{-}space(X::'a,Vt) \longrightarrow connected\text{-}space(X::'a,Vt) \mid separa$
$tion(f21(X::'a,Vt),f22(X::'a,Vt),X,Vt))\ \&$
$(\forall\ Va\ X\ Vt.\ connected\text{-}set(Va::'a,X,Vt) \longrightarrow topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ Vt\ Va\ X.\ connected\text{-}set(Va::'a,X,Vt) \longrightarrow subset\text{-}sets(Va::'a,X))\ \&$
$(\forall\ X\ Vt\ Va.\ connected\text{-}set(Va::'a,X,Vt) \longrightarrow connected\text{-}space(Va::'a,subspace\text{-}topology(X::'a,Vt,Va)))$
$\&$
$(\forall\ X\ Vt\ Va.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Va::'a,X)\ \&\ connected\text{-}space(Va::'a,subspace\text{-}topology(X$
$\longrightarrow connected\text{-}set(Va::'a,X,Vt))\ \&$
$(\forall\ Vf\ X\ Vt.\ open\text{-}covering(Vf::'a,X,Vt) \longrightarrow topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ X\ Vf\ Vt.\ open\text{-}covering(Vf::'a,X,Vt) \longrightarrow subset\text{-}collections(Vf::'a,Vt))\ \&$
$(\forall\ Vt\ Vf\ X.\ open\text{-}covering(Vf::'a,X,Vt) \longrightarrow equal\text{-}sets(union\text{-}of\text{-}members(Vf),X))$
$\&$
$(\forall\ Vt\ Vf\ X.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}collections(Vf::'a,Vt)\ \&\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X$
$\longrightarrow open\text{-}covering(Vf::'a,X,Vt))\ \&$
$(\forall\ X\ Vt.\ compact\text{-}space(X::'a,Vt) \longrightarrow topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ X\ Vt\ Vf1.\ compact\text{-}space(X::'a,Vt)\ \&\ open\text{-}covering(Vf1::'a,X,Vt) \longrightarrow fi$
$nite'(f23(X::'a,Vt,Vf1)))\ \&$
$(\forall\ X\ Vt\ Vf1.\ compact\text{-}space(X::'a,Vt)\ \&\ open\text{-}covering(Vf1::'a,X,Vt) \longrightarrow subset\text{-}collections(f23(X::'a,Vt,V$
$\&$
$(\forall\ Vf1\ X\ Vt.\ compact\text{-}space(X::'a,Vt)\ \&\ open\text{-}covering(Vf1::'a,X,Vt) \longrightarrow open\text{-}covering(f23(X::'a,Vt,Vf1)$
$\&$
$(\forall\ X\ Vt.\ topological\text{-}space(X::'a,Vt) \longrightarrow compact\text{-}space(X::'a,Vt) \mid open\text{-}covering(f24(X::'a,Vt),X,Vt))$
$\&$
$(\forall\ Uu24\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ finite'(Uu24)\ \&\ subset\text{-}collections(Uu24::'a,f24(X::'a,Vt))$
$\&\ open\text{-}covering(Uu24::'a,X,Vt) \longrightarrow compact\text{-}space(X::'a,Vt))\ \&$
$(\forall\ Va\ X\ Vt.\ compact\text{-}set(Va::'a,X,Vt) \longrightarrow topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ Vt\ Va\ X.\ compact\text{-}set(Va::'a,X,Vt) \longrightarrow subset\text{-}sets(Va::'a,X))\ \&$
$(\forall\ X\ Vt\ Va.\ compact\text{-}set(Va::'a,X,Vt) \longrightarrow compact\text{-}space(Va::'a,subspace\text{-}topology(X::'a,Vt,Va)))$
$\&$
$(\forall\ X\ Vt\ Va.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Va::'a,X)\ \&\ compact\text{-}space(Va::'a,subspace\text{-}topology(X:$

$--> compact\text{-}set(Va::'a,X,Vt))$ &
$(basis(cx::'a,f))$ &
$(\forall\, U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(f)))$ &
$(\forall\, V.\ element\text{-}of\text{-}collection(V::'a,top\text{-}of\text{-}basis(f)))$ &
$(\forall\, U\ V.\ \sim element\text{-}of\text{-}collection(intersection\text{-}of\text{-}sets(U::'a,V),top\text{-}of\text{-}basis(f))) -->$
False
⟨*proof*⟩

lemma *TOP004-2*:
$(\forall\, U\ Uu1\ Vf.\ element\text{-}of\text{-}set(U::'a,Uu1)$ & $element\text{-}of\text{-}collection(Uu1::'a,Vf) -->$
$element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf)))$ &
$(\forall\, Vf\ X.\ basis(X::'a,Vf) --> equal\text{-}sets(union\text{-}of\text{-}members(Vf),X))$ &
$(\forall\, X\ Vf\ Y\ Vb1\ Vb2.\ basis(X::'a,Vf)$ & $element\text{-}of\text{-}set(Y::'a,X)$ & $element\text{-}of\text{-}collection(Vb1::'a,Vf)$
& $element\text{-}of\text{-}collection(Vb2::'a,Vf)$ & $element\text{-}of\text{-}set(Y::'a,intersection\text{-}of\text{-}sets(Vb1::'a,Vb2))$
$--> element\text{-}of\text{-}set(Y::'a,f6(X::'a,Vf,Y,Vb1,Vb2)))$ &
$(\forall\, X\ Y\ Vb1\ Vb2\ Vf.\ basis(X::'a,Vf)$ & $element\text{-}of\text{-}set(Y::'a,X)$ & $element\text{-}of\text{-}collection(Vb1::'a,Vf)$
& $element\text{-}of\text{-}collection(Vb2::'a,Vf)$ & $element\text{-}of\text{-}set(Y::'a,intersection\text{-}of\text{-}sets(Vb1::'a,Vb2))$
$--> element\text{-}of\text{-}collection(f6(X::'a,Vf,Y,Vb1,Vb2),Vf))$ &
$(\forall\, X\ Vf\ Y\ Vb1\ Vb2.\ basis(X::'a,Vf)$ & $element\text{-}of\text{-}set(Y::'a,X)$ & $element\text{-}of\text{-}collection(Vb1::'a,Vf)$
& $element\text{-}of\text{-}collection(Vb2::'a,Vf)$ & $element\text{-}of\text{-}set(Y::'a,intersection\text{-}of\text{-}sets(Vb1::'a,Vb2))$
$--> subset\text{-}sets(f6(X::'a,Vf,Y,Vb1,Vb2),intersection\text{-}of\text{-}sets(Vb1::'a,Vb2)))$ &
$(\forall\, Vf\ U\ X.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))$ & $element\text{-}of\text{-}set(X::'a,U)$
$--> element\text{-}of\text{-}set(X::'a,f10(Vf::'a,U,X)))$ &
$(\forall\, U\ X\ Vf.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))$ & $element\text{-}of\text{-}set(X::'a,U)$
$--> element\text{-}of\text{-}collection(f10(Vf::'a,U,X),Vf))$ &
$(\forall\, Vf\ X\ U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))$ & $element\text{-}of\text{-}set(X::'a,U)$
$--> subset\text{-}sets(f10(Vf::'a,U,X),U))$ &
$(\forall\, Vf\ U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf)) \mid element\text{-}of\text{-}set(f11(Vf::'a,U),U))$
&
$(\forall\, Vf\ Uu11\ U.\ element\text{-}of\text{-}set(f11(Vf::'a,U),Uu11)$ & $element\text{-}of\text{-}collection(Uu11::'a,Vf)$
& $subset\text{-}sets(Uu11::'a,U) --> element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf)))$ &
$(\forall\, Y\ X\ Z.\ subset\text{-}sets(X::'a,Y)$ & $subset\text{-}sets(Y::'a,Z) --> subset\text{-}sets(X::'a,Z))$
&
$(\forall\, Y\ Z\ X.\ element\text{-}of\text{-}set(Z::'a,intersection\text{-}of\text{-}sets(X::'a,Y)) --> element\text{-}of\text{-}set(Z::'a,X))$
&
$(\forall\, X\ Z\ Y.\ element\text{-}of\text{-}set(Z::'a,intersection\text{-}of\text{-}sets(X::'a,Y)) --> element\text{-}of\text{-}set(Z::'a,Y))$
&
$(\forall\, X\ Z\ Y.\ element\text{-}of\text{-}set(Z::'a,X)$ & $element\text{-}of\text{-}set(Z::'a,Y) --> element\text{-}of\text{-}set(Z::'a,intersection\text{-}of\text{-}sets(X$
&
$(\forall\, X\ U\ Y\ V.\ subset\text{-}sets(X::'a,Y)$ & $subset\text{-}sets(U::'a,V) --> subset\text{-}sets(intersection\text{-}of\text{-}sets(X::'a,U),inte$
&
$(\forall\, X\ Z\ Y.\ equal\text{-}sets(X::'a,Y)$ & $element\text{-}of\text{-}set(Z::'a,X) --> element\text{-}of\text{-}set(Z::'a,Y))$
&
$(\forall\, Y\ X.\ equal\text{-}sets(intersection\text{-}of\text{-}sets(X::'a,Y),intersection\text{-}of\text{-}sets(Y::'a,X)))$ &
$(basis(cx::'a,f))$ &
$(\forall\, U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(f)))$ &
$(\forall\, V.\ element\text{-}of\text{-}collection(V::'a,top\text{-}of\text{-}basis(f)))$ &

(∀ U V. ~element-of-collection(intersection-of-sets(U::'a,V),top-of-basis(f))) −−>
False
⟨proof⟩

lemma *TOP005-2*:
(∀ Vf U. element-of-set(U::'a,union-of-members(Vf)) −−> element-of-set(U::'a,f1(Vf::'a,U)))
&
(∀ U Vf. element-of-set(U::'a,union-of-members(Vf)) −−> element-of-collection(f1(Vf::'a,U),Vf))
&
(∀ Vf U X. element-of-collection(U::'a,top-of-basis(Vf)) & element-of-set(X::'a,U)
−−> element-of-set(X::'a,f10(Vf::'a,U,X))) &
(∀ U X Vf. element-of-collection(U::'a,top-of-basis(Vf)) & element-of-set(X::'a,U)
−−> element-of-collection(f10(Vf::'a,U,X),Vf)) &
(∀ Vf X U. element-of-collection(U::'a,top-of-basis(Vf)) & element-of-set(X::'a,U)
−−> subset-sets(f10(Vf::'a,U,X),U)) &
(∀ Vf U. element-of-collection(U::'a,top-of-basis(Vf)) | element-of-set(f11(Vf::'a,U),U))
&
(∀ Vf Uu11 U. element-of-set(f11(Vf::'a,U),Uu11) & element-of-collection(Uu11::'a,Vf)
& subset-sets(Uu11::'a,U) −−> element-of-collection(U::'a,top-of-basis(Vf))) &
(∀ X U Y. element-of-set(U::'a,X) −−> subset-sets(X::'a,Y) | element-of-set(U::'a,Y))
&
(∀ Y X Z. subset-sets(X::'a,Y) & element-of-collection(Y::'a,Z) −−> subset-sets(X::'a,union-of-members(Z
&
(∀ X U Y. subset-collections(X::'a,Y) & element-of-collection(U::'a,X) −−>
element-of-collection(U::'a,Y)) &
(subset-collections(g::'a,top-of-basis(f))) &
(~element-of-collection(union-of-members(g),top-of-basis(f))) −−> False
⟨proof⟩

end

References

[1] K. McMillan. Lecture notes on verification of digital and hybrid systems. NATO summer school, http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial/toc.html.

[2] K. McMillan. *Symbolic Model Checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.