The Supplemental Isabelle/HOL Library

June 21, 2010

Contents

1 Abstract-Rat: Abstract rational numbers

theory *Abstract-Rat*
imports *Complex-Main*
begin

types *Num* = *int* × *int*

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

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

definition
isnormNum :: *Num* ⇒ *bool*
where
isnormNum = (λ(*a*,*b*). (*if a* = *0 then b* = *0 else b* > *0* ∧ *gcd a b* = *1*))

definition
normNum :: *Num* ⇒ *Num*
where
normNum = (λ(*a*,*b*). (*if a*=*0* ∨ *b* = *0 then* (*0*,*0*) *else*
(*let g* = *gcd a b*
in if b > *0 then* (*a div g*, *b div g*) *else* (− (*a div g*), − (*b div g*)))))

declare *gcd-dvd1-int*[*presburger*]
declare *gcd-dvd2-int*[*presburger*]
lemma *normNum-isnormNum* [*simp*]: *isnormNum* (*normNum x*)
⟨*proof*⟩

Arithmetic over Num

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

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

definition
Nneg :: *Num* ⇒ *Num* ($\sim_N$)
where

Nneg ≡ (λ(*a*,*b*). (−*a*,*b*))

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

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

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

lemma *Nneg-normN*[*simp*]: *isnormNum x* ⟹ *isnormNum* ($\sim_N$ *x*)
⟨*proof*⟩
lemma *Nadd-normN*[*simp*]: *isnormNum* (*x* $+_N$ *y*)
⟨*proof*⟩
lemma *Nsub-normN*[*simp*]: ⟦ *isnormNum y*⟧ ⟹ *isnormNum* (*x* $-_N$ *y*)
⟨*proof*⟩
lemma *Nmul-normN*[*simp*]: **assumes** *xn*:*isnormNum x* **and** *yn*: *isnormNum y*
shows *isnormNum* (*x* $*_N$ *y*)
⟨*proof*⟩

lemma *Ninv-normN*[*simp*]: *isnormNum x* ⟹ *isnormNum* (*Ninv x*)
⟨*proof*⟩

lemma *isnormNum-int*[*simp*]:
isnormNum 0_N *isnormNum* (*1*::*int*)$_N$ *i* ≠ *0* ⟹ *isnormNum* i_N
⟨*proof*⟩

Relations over Num

definition
Nlt0:: *Num* ⇒ *bool* (*0*>$_N$)
where
Nlt0 = (λ(*a*,*b*). *a* < *0*)

definition
Nle0:: *Num* ⇒ *bool* (*0*≥$_N$)
where
Nle0 = (λ(*a*,*b*). *a* ≤ *0*)

definition
Ngt0:: *Num* ⇒ *bool* (*0*<$_N$)
where
Ngt0 = (λ(*a*,*b*). *a* > *0*)

definition
Nge0:: *Num* ⇒ *bool* (*0*≤$_N$)
where
Nge0 = (λ(*a*,*b*). *a* ≥ *0*)

definition
Nlt :: *Num* ⇒ *Num* ⇒ *bool* (**infix** <$_N$ *55*)
where
Nlt = (λ*a b*. *0*>$_N$ (*a* −$_N$ *b*))

definition
Nle :: *Num* ⇒ *Num* ⇒ *bool* (**infix** ≤$_N$ *55*)
where
Nle = (λ*a b*. *0*≥$_N$ (*a* −$_N$ *b*))

definition
INum = (λ(*a*,*b*). *of-int a* / *of-int b*)

lemma *INum-int* [*simp*]: *INum i*$_N$ = ((*of-int i*) ::′*a*::*field*) *INum 0*$_N$ = (*0*::′*a*::*field*)
⟨*proof*⟩

lemma *isnormNum-unique*[*simp*]:
assumes *na*: *isnormNum x* **and** *nb*: *isnormNum y*
shows ((*INum x* ::′*a*::{*field-char-0*, *field-inverse-zero*}) = *INum y*) = (*x* = *y*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *isnormNum0*[*simp*]: *isnormNum x* ⟹ (*INum x* = (*0*::′*a*::{*field-char-0*, *field-inverse-zero*})) = (*x* = *0*$_N$)
⟨*proof*⟩

lemma *of-int-div-aux*: *d* ~= *0* ==> ((*of-int x*)::′*a*::*field-char-0*) / (*of-int d*) =
of-int (*x div d*) + (*of-int* (*x mod d*)) / ((*of-int d*)::′*a*)
⟨*proof*⟩

lemma *of-int-div*: (*d*::*int*) ~= *0* ==> *d dvd n* ==>
(*of-int*(*n div d*)::′*a*::*field-char-0*) = *of-int n* / *of-int d*
⟨*proof*⟩

lemma *normNum*[*simp*]: *INum* (*normNum x*) = (*INum x* :: ′*a*::{*field-char-0*, *field-inverse-zero*})
⟨*proof*⟩

lemma *INum-normNum-iff*: (*INum x* ::′*a*::{*field-char-0*, *field-inverse-zero*}) = *INum y* ⟷ *normNum x* = *normNum y* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Nadd*[*simp*]: *INum* $(x +_N y) =$ *INum* $x +$ (*INum* $y ::$ *'a* :: {*field-char-0*, *field-inverse-zero*})
⟨*proof*⟩

lemma *Nmul*[*simp*]: *INum* $(x *_N y) =$ *INum* $x *$ (*INum* $y::$ *'a* :: {*field-char-0*, *field-inverse-zero*})
⟨*proof*⟩

lemma *Nneg*[*simp*]: *INum* $(\sim_N x) = -$ (*INum* $x ::'a::$ *field*)
⟨*proof*⟩

lemma *Nsub*[*simp*]: **shows** *INum* $(x -_N y) =$ *INum* $x -$ (*INum* $y::$ *'a* :: {*field-char-0*, *field-inverse-zero*})
⟨*proof*⟩

lemma *Ninv*[*simp*]: *INum* (*Ninv* x) = ($1::'a$ :: *field-inverse-zero*) / (*INum* x)
⟨*proof*⟩

lemma *Ndiv*[*simp*]: *INum* $(x \div_N y) =$ *INum* x / (*INum* $y ::'a$ :: {*field-char-0*, *field-inverse-zero*}) ⟨*proof*⟩

lemma *Nlt0-iff*[*simp*]: **assumes** *nx*: *isnormNum* x
shows ((*INum* $x ::$ *'a* :: {*field-char-0*, *linordered-field-inverse-zero*})$< 0) = 0 >_N$ x
⟨*proof*⟩

lemma *Nle0-iff*[*simp*]:**assumes** *nx*: *isnormNum* x
shows ((*INum* $x ::$ *'a* :: {*field-char-0*, *linordered-field-inverse-zero*}) $\leq 0) = 0 \geq_N$ x
⟨*proof*⟩

lemma *Ngt0-iff*[*simp*]:**assumes** *nx*: *isnormNum* x **shows** ((*INum* $x ::$ *'a* :: {*field-char-0*, *linordered-field-inverse-zero*})$> 0) = 0 <_N x$
⟨*proof*⟩
lemma *Nge0-iff*[*simp*]:**assumes** *nx*: *isnormNum* x
shows ((*INum* $x ::$ *'a* :: {*field-char-0*, *linordered-field-inverse-zero*}) $\geq 0) = 0 \leq_N$ x
⟨*proof*⟩

lemma *Nlt-iff*[*simp*]: **assumes** *nx*: *isnormNum* x **and** *ny*: *isnormNum* y
shows ((*INum* $x ::$ *'a* :: {*field-char-0*, *linordered-field-inverse-zero*}) $<$ *INum* y) = ($x <_N y$)
⟨*proof*⟩

lemma *Nle-iff*[*simp*]: **assumes** *nx*: *isnormNum* x **and** *ny*: *isnormNum* y
shows ((*INum* $x ::$ *'a* :: {*field-char-0*, *linordered-field-inverse-zero*})$\leq$ *INum* y) = ($x \leq_N y$)
⟨*proof*⟩

lemma *Nadd-commute*:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows $x +_N y = y +_N x$
⟨*proof*⟩

lemma [*simp*]:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows $(0, b) +_N y = normNum\ y$
and $(a, 0) +_N y = normNum\ y$
and $x +_N (0, b) = normNum\ x$
and $x +_N (a, 0) = normNum\ x$
⟨*proof*⟩

lemma *normNum-nilpotent-aux*[*simp*]:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
assumes *nx*: *isnormNum x*
shows *normNum x* = *x*
⟨*proof*⟩

lemma *normNum-nilpotent*[*simp*]:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows *normNum* (*normNum x*) = *normNum x*
⟨*proof*⟩

lemma *normNum0*[*simp*]: *normNum* (*0*,*b*) = 0_N *normNum* (*a*,*0*) = 0_N
⟨*proof*⟩

lemma *normNum-Nadd*:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows *normNum* $(x +_N y) = x +_N y$ ⟨*proof*⟩

lemma *Nadd-normNum1*[*simp*]:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows *normNum x* $+_N$ *y* = *x* $+_N$ *y*
⟨*proof*⟩

lemma *Nadd-normNum2*[*simp*]:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows *x* $+_N$ *normNum y* = *x* $+_N$ *y*
⟨*proof*⟩

lemma *Nadd-assoc*:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows $x +_N y +_N z = x +_N (y +_N z)$
⟨*proof*⟩

lemma *Nmul-commute*: *isnormNum x* $\Longrightarrow$ *isnormNum y* $\Longrightarrow$ $x *_N y = y *_N x$
⟨*proof*⟩

lemma *Nmul-assoc*:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
assumes *nx*: *isnormNum x* **and** *ny*:*isnormNum y* **and** *nz*:*isnormNum z*
shows $x *_N y *_N z = x *_N (y *_N z)$
⟨*proof*⟩

lemma *Nsub0*:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
assumes *x*: *isnormNum x* **and** *y*:*isnormNum y* **shows** $(x -_N y = 0_N) = (x = y)$
⟨*proof*⟩

lemma *Nmul0*[*simp*]: $c *_N 0_N = 0_N \quad 0_N *_N c = 0_N$
⟨*proof*⟩

lemma *Nmul-eq0*[*simp*]:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
assumes *nx*:*isnormNum x* **and** *ny*: *isnormNum y*
shows $(x *_N y = 0_N) = (x = 0_N \vee y = 0_N)$
⟨*proof*⟩
lemma *Nneg-Nneg*[*simp*]: $\sim_N (\sim_N c) = c$
⟨*proof*⟩

lemma *Nmul1*[*simp*]:
isnormNum c $\Longrightarrow 1_N *_N c = c$
isnormNum c $\Longrightarrow c *_N 1_N = c$
⟨*proof*⟩

end

2 Mapping: An abstract view on maps for code generation.

theory *Mapping*
imports *Main*
begin

2.1 Type definition and primitive operations

datatype ('*a*, '*b*) *mapping* = *Mapping* '*a* ⇀ '*b*

definition *empty* :: ('*a*, '*b*) *mapping* **where**
empty = *Mapping* (λ-. *None*)

primrec *lookup* :: ('*a*, '*b*) *mapping* ⇒ '*a* ⇀ '*b* **where**
lookup (*Mapping f*) = *f*

primrec *update* :: *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping* **where**
update k v (*Mapping f*) = *Mapping* (*f* (*k* ↦ *v*))

primrec *delete* :: *'a* ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping* **where**
delete k (*Mapping f*) = *Mapping* (*f* (*k* := *None*))

2.2 Derived operations

definition *keys* :: (*'a*, *'b*) *mapping* ⇒ *'a set* **where**
keys m = *dom* (*lookup m*)

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*)

definition *map-entry* :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping*
where
map-entry k f m = (*case lookup m k of None* ⇒ *m*
| *Some v* ⇒ *update k* (*f v*) *m*)

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 *tabulate* :: *'a list* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a*, *'b*) *mapping* **where**
tabulate ks f = *Mapping* (*map-of* (*map* (λ*k*. (*k*, *f k*)) *ks*))

definition *bulkload* :: *'a list* ⇒ (*nat*, *'a*) *mapping* **where**
bulkload xs = *Mapping* (λ*k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None*)

2.3 Properties

lemma *lookup-inject* [*simp*]:
lookup m = *lookup n* ⟷ *m* = *n*
⟨*proof*⟩

lemma *mapping-eqI*:
assumes *lookup m* = *lookup n*

shows $m = n$
⟨*proof*⟩

lemma *keys-is-none-lookup* [*code-inline*]:
$k \in keys\ m \longleftrightarrow \neg\ (Option.is\text{-}none\ (lookup\ m\ k))$
⟨*proof*⟩

lemma *lookup-empty* [*simp*]:
$lookup\ empty = Map.empty$
⟨*proof*⟩

lemma *lookup-update* [*simp*]:
$lookup\ (update\ k\ v\ m) = (lookup\ m)\ (k \mapsto v)$
⟨*proof*⟩

lemma *lookup-delete* [*simp*]:
$lookup\ (delete\ k\ m) = (lookup\ m)\ (k := None)$
⟨*proof*⟩

lemma *lookup-map-entry* [*simp*]:
$lookup\ (map\text{-}entry\ k\ f\ m) = (lookup\ m)\ (k := Option.map\ f\ (lookup\ m\ k))$
⟨*proof*⟩

lemma *lookup-tabulate* [*simp*]:
$lookup\ (tabulate\ ks\ f) = (Some\ o\ f)\ |`\ set\ ks$
⟨*proof*⟩

lemma *lookup-bulkload* [*simp*]:
$lookup\ (bulkload\ xs) = (\lambda k.\ if\ k < length\ xs\ then\ Some\ (xs\ !\ k)\ else\ None)$
⟨*proof*⟩

lemma *update-update*:
$update\ k\ v\ (update\ k\ w\ m) = update\ k\ v\ m$
$k \neq l \Longrightarrow update\ k\ v\ (update\ l\ w\ m) = update\ l\ w\ (update\ k\ v\ m)$
⟨*proof*⟩

lemma *update-delete* [*simp*]:
$update\ k\ v\ (delete\ k\ m) = update\ k\ v\ m$
⟨*proof*⟩

lemma *delete-update*:
$delete\ k\ (update\ k\ v\ m) = delete\ k\ m$
$k \neq l \Longrightarrow delete\ k\ (update\ l\ v\ m) = update\ l\ v\ (delete\ k\ m)$
⟨*proof*⟩

lemma *delete-empty* [*simp*]:
$delete\ k\ empty = empty$
⟨*proof*⟩

lemma *replace-update*:
$k \notin keys\ m \Longrightarrow replace\ k\ v\ m = m$
$k \in keys\ m \Longrightarrow replace\ k\ v\ m = update\ k\ v\ m$
⟨*proof*⟩

lemma *size-empty* [*simp*]:
$size\ empty = 0$
⟨*proof*⟩

lemma *size-update*:
$finite\ (keys\ m) \Longrightarrow size\ (update\ k\ v\ m) =$
$(if\ k \in keys\ m\ then\ size\ m\ else\ Suc\ (size\ m))$
⟨*proof*⟩

lemma *size-delete*:
$size\ (delete\ k\ m) = (if\ k \in keys\ m\ then\ size\ m - 1\ else\ size\ m)$
⟨*proof*⟩

lemma *size-tabulate* [*simp*]:
$size\ (tabulate\ ks\ f) = length\ (remdups\ ks)$
⟨*proof*⟩

lemma *bulkload-tabulate*:
$bulkload\ xs = tabulate\ [0..<length\ xs]\ (nth\ xs)$
⟨*proof*⟩

lemma *is-empty-empty*:
$is\text{-}empty\ m \longleftrightarrow m = Mapping\ Map.empty$
⟨*proof*⟩

lemma *is-empty-empty'* [*simp*]:
$is\text{-}empty\ empty$
⟨*proof*⟩

lemma *is-empty-update* [*simp*]:
$\neg\ is\text{-}empty\ (update\ k\ v\ m)$
⟨*proof*⟩

lemma *is-empty-delete*:
$is\text{-}empty\ (delete\ k\ m) \longleftrightarrow is\text{-}empty\ m \vee keys\ m = \{k\}$
⟨*proof*⟩

lemma *is-empty-replace* [*simp*]:
$is\text{-}empty\ (replace\ k\ v\ m) \longleftrightarrow is\text{-}empty\ m$
⟨*proof*⟩

lemma *is-empty-default* [*simp*]:
$\neg\ is\text{-}empty\ (default\ k\ v\ m)$
⟨*proof*⟩

lemma *is-empty-map-entry* [*simp*]:
is-empty (*map-entry k f m*) ⟷ *is-empty m*
⟨*proof*⟩

lemma *is-empty-map-default* [*simp*]:
¬ *is-empty* (*map-default k v f 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-tabulate* [*simp*]:
keys (*tabulate ks f*) = *set ks*
⟨*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*⟩

2.4 Some technical code lemmas

lemma [*code*]:

mapping-case f m = f (Mapping.lookup m)
⟨*proof*⟩

lemma [*code*]:
mapping-rec f m = f (Mapping.lookup m)
⟨*proof*⟩

lemma [*code*]:
Nat.size (m :: (-, -) mapping) = 0
⟨*proof*⟩

lemma [*code*]:
mapping-size f g m = 0
⟨*proof*⟩

hide-const (**open**) *empty is-empty lookup update delete ordered-keys keys size replace default map-entry map-default tabulate bulkload*

end

3 AssocList: Map operations implemented on association lists

theory *AssocList*
imports *Main Mapping*
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.

3.1 *update* and *updates*

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

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 al = foldl (λal (k, v). update k v al) al (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*⟩

3.2 *delete*

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*⟩

3.3 *restrict*

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*) ⟹ *distinct* (*map fst* (*restrict A al*))
⟨*proof*⟩

lemma *restr-empty* [*simp*]:
restrict {} *al* = []
restrict A [] = []
⟨*proof*⟩

lemma *restr-in* [*simp*]: *x* ∈ *A* ⟹ *map-of* (*restrict A al*) *x* = *map-of al x*
⟨*proof*⟩

lemma *restr-out* [*simp*]: *x* ∉ *A* ⟹ *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 *upate-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*⟩

3.4 *clearjunk*

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*⟩

3.5 *map-ran*

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

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

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

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

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

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

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

3.6 *merge*

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

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*: $fst\ `\ set\ (merge\ xs\ ys) = fst\ `\ set\ xs \cup fst\ `\ set\ ys$
⟨*proof*⟩

lemma *distinct-merge*:

assumes *distinct* (*map fst xs*)
shows *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*, *standard*]

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*⟩

3.7 *compose*

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*]:
assumes *map-of xs k* = *None*
shows *map-of* (*compose xs ys*) *k* = *None*
⟨*proof*⟩

lemma *compose-conv*:
shows *map-of* (*compose xs ys*) *k* = (*map-of ys* ∘$_m$ *map-of xs*) *k*
⟨*proof*⟩

lemma *compose-conv'*:
shows *map-of* (*compose xs ys*) = (*map-of ys* ∘$_m$ *map-of xs*)
⟨*proof*⟩

lemma *compose-first-Some* [*simp*]:
assumes *map-of xs k* = *Some v*
shows *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*⟩

3.8 Implementation of mappings

definition *Mapping* :: ('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *mapping* **where**
Mapping xs = *Mapping.Mapping* (*map-of xs*)

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*) ⟷ *null xs*
⟨*proof*⟩

lemma *update-Mapping* [*code*]:
Mapping.update k v (*Mapping xs*) = *Mapping* (*update k v xs*)
⟨*proof*⟩

lemma *delete-Mapping* [*code*]:
Mapping.delete k (*Mapping xs*) = *Mapping* (*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 [*code*, *code del*]:
HOL.eq (*x* :: (-, -) *mapping*) *y* ⟷ *x* = *y* ⟨*proof*⟩

end

4 SetsAndFunctions: Operations on sets and functions

theory *SetsAndFunctions*
imports *Main*
begin

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

4.1 Basic definitions

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

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

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

instance ⟨*proof*⟩

end

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

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

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

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

instance *fun* :: (*type*,*semigroup-add*)*semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*,*comm-monoid-add*)*comm-monoid-add*
⟨*proof*⟩

instance *fun* :: (*type*,*ab-group-add*)*ab-group-add*
⟨*proof*⟩

instance *fun* :: (*type*,*semigroup-mult*)*semigroup-mult*
⟨*proof*⟩

instance *fun* :: (*type*,*comm-monoid-mult*)*comm-monoid-mult*
⟨*proof*⟩

instance *fun* :: (*type*,*comm-ring-1*)*comm-ring-1*
⟨*proof*⟩

interpretation *set-semigroup-add*: *semigroup-add op* ⊕ :: ('*a*::*semigroup-add*) *set*
=> '*a set* => '*a set*
⟨*proof*⟩

interpretation *set-semigroup-mult*: *semigroup-mult op* ⊗ :: ('*a*::*semigroup-mult*)
set => '*a set* => '*a set*
⟨*proof*⟩

interpretation *set-comm-monoid-add*: *comm-monoid-add op* ⊕ :: ('*a*::*comm-monoid-add*)
set => '*a set* => '*a set* {*0*}
⟨*proof*⟩

interpretation *set-comm-monoid-mult*: *comm-monoid-mult op* ⊗ :: ('*a*::*comm-monoid-mult*)
set => '*a set* => '*a set* {*1*}
⟨*proof*⟩

4.2 Basic properties

lemma *set-plus-intro* [*intro*]: *a* : *C* ==> *b* : *D* ==> *a* + *b* : *C* ⊕ *D*
⟨*proof*⟩

lemma *set-plus-intro2* [*intro*]: *b* : *C* ==> *a* + *b* : *a* +*o* *C*
⟨*proof*⟩

lemma *set-plus-rearrange*: ((*a*::'*a*::*comm-monoid-add*) +*o* *C*) ⊕
(*b* +*o* *D*) = (*a* + *b*) +*o* (*C* ⊕ *D*)
⟨*proof*⟩

lemma *set-plus-rearrange2*: (*a*::'*a*::*semigroup-add*) +*o* (*b* +*o* *C*) =
(*a* + *b*) +*o* *C*
⟨*proof*⟩

lemma *set-plus-rearrange3*: ((*a*::'*a*::*semigroup-add*) +*o* *B*) ⊕ *C* =
a +*o* (*B* ⊕ *C*)
⟨*proof*⟩

theorem *set-plus-rearrange4*: *C* ⊕ ((*a*::'*a*::*comm-monoid-add*) +*o* *D*) =
a +*o* (*C* ⊕ *D*)
⟨*proof*⟩

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

lemma *set-plus-mono* [*intro*!]: $C <= D ==> a +o C <= a +o D$
⟨*proof*⟩

lemma *set-plus-mono2* [*intro*]: $(C::('a::plus)\ set) <= D ==> E <= F ==> C \oplus E <= D \oplus F$
⟨*proof*⟩

lemma *set-plus-mono3* [*intro*]: $a : C ==> a +o D <= C \oplus D$
⟨*proof*⟩

lemma *set-plus-mono4* [*intro*]: $(a::'a::comm\text{-}monoid\text{-}add) : C ==> a +o D <= D \oplus C$
⟨*proof*⟩

lemma *set-plus-mono5*: $a:C ==> B <= D ==> a +o B <= C \oplus D$
⟨*proof*⟩

lemma *set-plus-mono-b*: $C <= D ==> x : a +o C ==> x : a +o D$
⟨*proof*⟩

lemma *set-plus-mono2-b*: $C <= D ==> E <= F ==> x : C \oplus E ==> x : D \oplus F$
⟨*proof*⟩

lemma *set-plus-mono3-b*: $a : C ==> x : a +o D ==> x : C \oplus D$
⟨*proof*⟩

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

lemma *set-zero-plus* [*simp*]: $(0::'a::comm\text{-}monoid\text{-}add) +o C = C$
⟨*proof*⟩

lemma *set-zero-plus2*: $(0::'a::comm\text{-}monoid\text{-}add) : A ==> B <= A \oplus B$
⟨*proof*⟩

lemma *set-plus-imp-minus*: $(a::'a::ab\text{-}group\text{-}add) : b +o C ==> (a - b) : C$
⟨*proof*⟩

lemma *set-minus-imp-plus*: $(a::'a::ab\text{-}group\text{-}add) - b : C ==> a : b +o C$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *set-times-intro* [*intro*]: $a : C \Longrightarrow b : D \Longrightarrow a * b : C \otimes D$
⟨*proof*⟩

lemma *set-times-intro2* [*intro*!]: $b : C \Longrightarrow a * b : a *o C$
⟨*proof*⟩

lemma *set-times-rearrange*: $((a::'a::comm\text{-}monoid\text{-}mult) *o C) \otimes (b *o D) = (a * b) *o (C \otimes D)$
⟨*proof*⟩

lemma *set-times-rearrange2*: $(a::'a::semigroup\text{-}mult) *o (b *o C) = (a * b) *o C$
⟨*proof*⟩

lemma *set-times-rearrange3*: $((a::'a::semigroup\text{-}mult) *o B) \otimes C = a *o (B \otimes C)$
⟨*proof*⟩

theorem *set-times-rearrange4*: $C \otimes ((a::'a::comm\text{-}monoid\text{-}mult) *o D) = a *o (C \otimes D)$
⟨*proof*⟩

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

lemma *set-times-mono* [*intro*]: $C <= D \Longrightarrow a *o C <= a *o D$
⟨*proof*⟩

lemma *set-times-mono2* [*intro*]: $(C::('a::times)\ set) <= D \Longrightarrow E <= F \Longrightarrow C \otimes E <= D \otimes F$
⟨*proof*⟩

lemma *set-times-mono3* [*intro*]: $a : C \Longrightarrow a *o D <= C \otimes D$
⟨*proof*⟩

lemma *set-times-mono4* [*intro*]: $(a::'a::comm\text{-}monoid\text{-}mult) : C \Longrightarrow a *o D <= D \otimes C$
⟨*proof*⟩

lemma *set-times-mono5*: $a:C \Longrightarrow B <= D \Longrightarrow a *o B <= C \otimes D$
⟨*proof*⟩

lemma *set-times-mono-b*: $C <= D \Longrightarrow x : a *o C \Longrightarrow x : a *o D$
⟨*proof*⟩

lemma *set-times-mono2-b*: $C <= D \Longrightarrow E <= F \Longrightarrow x : C \otimes E \Longrightarrow$

$x : D \otimes F$
⟨*proof*⟩

lemma *set-times-mono3-b*: $a : C \Longrightarrow x : a *o D \Longrightarrow x : C \otimes D$
⟨*proof*⟩

lemma *set-times-mono4-b*: $(a::'a::comm\text{-}monoid\text{-}mult) : C \Longrightarrow$
$x : a *o D \Longrightarrow x : D \otimes C$
⟨*proof*⟩

lemma *set-one-times* [*simp*]: $(1::'a::comm\text{-}monoid\text{-}mult) *o C = C$
⟨*proof*⟩

lemma *set-times-plus-distrib*: $(a::'a::semiring) *o (b +o C) =$
$(a * b) +o (a *o C)$
⟨*proof*⟩

lemma *set-times-plus-distrib2*: $(a::'a::semiring) *o (B \oplus C) =$
$(a *o B) \oplus (a *o C)$
⟨*proof*⟩

lemma *set-times-plus-distrib3*: $((a::'a::semiring) +o C) \otimes D <=$
$a *o D \oplus C \otimes D$
⟨*proof*⟩

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

lemma *set-neg-intro*: $(a::'a::ring\text{-}1) : (-1) *o C \Longrightarrow$
$-a : C$
⟨*proof*⟩

lemma *set-neg-intro2*: $(a::'a::ring\text{-}1) : C \Longrightarrow$
$-a : (-1) *o C$
⟨*proof*⟩

end

5 BigO: Big O notation

theory *BigO*
imports *Complex-Main SetsAndFunctions*
begin

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

The main changes in this version are as follows:

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

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

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

5.1 Definitions

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

lemma *bigo-pos-const*: (*EX* (*c*::'*a*::*linordered-idom*).
ALL x. (*abs* (*h x*)) <= (*c* * (*abs* (*f x*))))
= (*EX c*. *0* < *c* & (*ALL x*. (*abs*(*h x*)) <= (*c* * (*abs* (*f x*))))))
⟨*proof*⟩

lemma *bigo-alt-def*: *O*(*f*) =
{*h*. *EX c*. (*0* < *c* & (*ALL x*. *abs* (*h x*) <= *c* * *abs* (*f x*)))}
⟨*proof*⟩

lemma *bigo-elt-subset* [*intro*]: *f* : *O*(*g*) ==> *O*(*f*) <= *O*(*g*)
⟨*proof*⟩

lemma *bigo-refl* [*intro*]: *f* : *O*(*f*)
⟨*proof*⟩

lemma *bigo-zero*: *0* : *O*(*g*)
⟨*proof*⟩

lemma *bigo-zero2*: *O*(%*x*.*0*) = {%*x*.*0*}
⟨*proof*⟩

lemma *bigo-plus-self-subset* [*intro*]:

$O(f) \oplus O(f) <= O(f)$
⟨*proof*⟩

lemma *bigo-plus-idemp* [*simp*]: $O(f) \oplus O(f) = O(f)$
⟨*proof*⟩

lemma *bigo-plus-subset* [*intro*]: $O(f + g) <= O(f) \oplus O(g)$
⟨*proof*⟩

lemma *bigo-plus-subset2* [*intro*]: $A <= O(f) ==> B <= O(f) ==> A \oplus B <= O(f)$
⟨*proof*⟩

lemma *bigo-plus-eq*: $ALL\ x.\ 0 <= f\ x ==> ALL\ x.\ 0 <= g\ x ==>$
$O(f + g) = O(f) \oplus O(g)$
⟨*proof*⟩

lemma *bigo-bounded-alt*: $ALL\ x.\ 0 <= f\ x ==> ALL\ x.\ f\ x <= c * g\ x ==>$
$f : O(g)$
⟨*proof*⟩

lemma *bigo-bounded*: $ALL\ x.\ 0 <= f\ x ==> ALL\ x.\ f\ x <= g\ x ==>$
$f : O(g)$
⟨*proof*⟩

lemma *bigo-bounded2*: $ALL\ x.\ lb\ x <= f\ x ==> ALL\ x.\ f\ x <= lb\ x + g\ x ==>$
$f : lb +o O(g)$
⟨*proof*⟩

lemma *bigo-abs*: $(\%x.\ abs(f\ x)) =o O(f)$
⟨*proof*⟩

lemma *bigo-abs2*: $f =o O(\%x.\ abs(f\ x))$
⟨*proof*⟩

lemma *bigo-abs3*: $O(f) = O(\%x.\ abs(f\ x))$
⟨*proof*⟩

lemma *bigo-abs4*: $f =o g +o O(h) ==>$
$(\%x.\ abs\ (f\ x)) =o (\%x.\ abs\ (g\ x)) +o O(h)$
⟨*proof*⟩

lemma *bigo-abs5*: $f =o O(g) ==> (\%x.\ abs(f\ x)) =o O(g)$
⟨*proof*⟩

lemma *bigo-elt-subset2* [*intro*]: $f : g +o O(h) ==> O(f) <= O(g) \oplus O(h)$
⟨*proof*⟩

lemma *bigo-mult* [*intro*]: $O(f) \otimes O(g) <= O(f * g)$

⟨*proof*⟩

lemma *bigo-mult2* [*intro*]: $f *o O(g) <= O(f * g)$
⟨*proof*⟩

lemma *bigo-mult3*: $f : O(h) ==> g : O(j) ==> f * g : O(h * j)$
⟨*proof*⟩

lemma *bigo-mult4* [*intro*]:$f : k +o O(h) ==> g * f : (g * k) +o O(g * h)$
⟨*proof*⟩

lemma *bigo-mult5*: $ALL\ x.\ f\ x \sim= 0 ==>$
$O(f * g) <= (f::'a => ('b::linordered\text{-}field)) *o O(g)$
⟨*proof*⟩

lemma *bigo-mult6*: $ALL\ x.\ f\ x \sim= 0 ==>$
$O(f * g) = (f::'a => ('b::linordered\text{-}field)) *o O(g)$
⟨*proof*⟩

lemma *bigo-mult7*: $ALL\ x.\ f\ x \sim= 0 ==>$
$O(f * g) <= O(f::'a => ('b::linordered\text{-}field)) \otimes O(g)$
⟨*proof*⟩

lemma *bigo-mult8*: $ALL\ x.\ f\ x \sim= 0 ==>$
$O(f * g) = O(f::'a => ('b::linordered\text{-}field)) \otimes O(g)$
⟨*proof*⟩

lemma *bigo-minus* [*intro*]: $f : O(g) ==> - f : O(g)$
⟨*proof*⟩

lemma *bigo-minus2*: $f : g +o O(h) ==> -f : -g +o O(h)$
⟨*proof*⟩

lemma *bigo-minus3*: $O(-f) = O(f)$
⟨*proof*⟩

lemma *bigo-plus-absorb-lemma1*: $f : O(g) ==> f +o O(g) <= O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb-lemma2*: $f : O(g) ==> O(g) <= f +o O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb* [*simp*]: $f : O(g) ==> f +o O(g) = O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb2* [*intro*]: $f : O(g) ==> A <= O(g) ==> f +o A <= O(g)$
⟨*proof*⟩

lemma *bigo-add-commute-imp*: $f : g +o O(h) ==> g : f +o O(h)$
⟨*proof*⟩

lemma *bigo-add-commute*: $(f : g +o O(h)) = (g : f +o O(h))$
⟨*proof*⟩

lemma *bigo-const1*: $(\%x.\ c) : O(\%x.\ 1)$
⟨*proof*⟩

lemma *bigo-const2* [*intro*]: $O(\%x.\ c) <= O(\%x.\ 1)$
⟨*proof*⟩

lemma *bigo-const3*: $(c::'a::linordered\text{-}field) \sim= 0 ==> (\%x.\ 1) : O(\%x.\ c)$
⟨*proof*⟩

lemma *bigo-const4*: $(c::'a::linordered\text{-}field) \sim= 0 ==> O(\%x.\ 1) <= O(\%x.\ c)$
⟨*proof*⟩

lemma *bigo-const* [*simp*]: $(c::'a::linordered\text{-}field) \sim= 0 ==>$
$O(\%x.\ c) = O(\%x.\ 1)$
⟨*proof*⟩

lemma *bigo-const-mult1*: $(\%x.\ c * f\ x) : O(f)$
⟨*proof*⟩

lemma *bigo-const-mult2*: $O(\%x.\ c * f\ x) <= O(f)$
⟨*proof*⟩

lemma *bigo-const-mult3*: $(c::'a::linordered\text{-}field) \sim= 0 ==> f : O(\%x.\ c * f\ x)$
⟨*proof*⟩

lemma *bigo-const-mult4*: $(c::'a::linordered\text{-}field) \sim= 0 ==>$
$O(f) <= O(\%x.\ c * f\ x)$
⟨*proof*⟩

lemma *bigo-const-mult* [*simp*]: $(c::'a::linordered\text{-}field) \sim= 0 ==>$
$O(\%x.\ c * f\ x) = O(f)$
⟨*proof*⟩

lemma *bigo-const-mult5* [*simp*]: $(c::'a::linordered\text{-}field) \sim= 0 ==>$
$(\%x.\ c) *o O(f) = O(f)$
⟨*proof*⟩

lemma *bigo-const-mult6* [*intro*]: $(\%x.\ c) *o O(f) <= O(f)$
⟨*proof*⟩

lemma *bigo-const-mult7* [*intro*]: $f =o O(g) ==> (\%x.\ c * f\ x) =o O(g)$
⟨*proof*⟩

lemma *bigo-compose1*: $f =o O(g) ==> (\%x.\ f(k\ x)) =o O(\%x.\ g(k\ x))$
⟨*proof*⟩

lemma *bigo-compose2*: $f =o g +o O(h) ==> (\%x.\ f(k\ x)) =o (\%x.\ g(k\ x)) +o$
$O(\%x.\ h(k\ x))$
⟨*proof*⟩

5.2 Setsum

lemma *bigo-setsum-main*: $ALL\ x.\ ALL\ y : A\ x.\ 0 <= h\ x\ y ==>$
$EX\ c.\ ALL\ x.\ ALL\ y : A\ x.\ abs(f\ x\ y) <= c * (h\ x\ y) ==>$
$(\%x.\ SUM\ y : A\ x.\ f\ x\ y) =o O(\%x.\ SUM\ y : A\ x.\ h\ x\ y)$
⟨*proof*⟩

lemma *bigo-setsum1*: $ALL\ x\ y.\ 0 <= h\ x\ y ==>$
$EX\ c.\ ALL\ x\ y.\ abs(f\ x\ y) <= c * (h\ x\ y) ==>$
$(\%x.\ SUM\ y : A\ x.\ f\ x\ y) =o O(\%x.\ SUM\ y : A\ x.\ h\ x\ y)$
⟨*proof*⟩

lemma *bigo-setsum2*: $ALL\ y.\ 0 <= h\ y ==>$
$EX\ c.\ ALL\ y.\ abs(f\ y) <= c * (h\ y) ==>$
$(\%x.\ SUM\ y : A\ x.\ f\ y) =o O(\%x.\ SUM\ y : A\ x.\ h\ y)$
⟨*proof*⟩

lemma *bigo-setsum3*: $f =o O(h) ==>$
$(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * f(k\ x\ y)) =o$
$O(\%x.\ SUM\ y : A\ x.\ abs(l\ x\ y * h(k\ x\ y)))$
⟨*proof*⟩

lemma *bigo-setsum4*: $f =o g +o O(h) ==>$
$(\%x.\ SUM\ y : A\ x.\ l\ x\ y * f(k\ x\ y)) =o$
$(\%x.\ SUM\ y : A\ x.\ l\ x\ y * g(k\ x\ y)) +o$
$O(\%x.\ SUM\ y : A\ x.\ abs(l\ x\ y * h(k\ x\ y)))$
⟨*proof*⟩

lemma *bigo-setsum5*: $f =o O(h) ==> ALL\ x\ y.\ 0 <= l\ x\ y ==>$
$ALL\ x.\ 0 <= h\ x ==>$
$(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * f(k\ x\ y)) =o$
$O(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * h(k\ x\ y))$
⟨*proof*⟩

lemma *bigo-setsum6*: $f =o g +o O(h) ==> ALL\ x\ y.\ 0 <= l\ x\ y ==>$
$ALL\ x.\ 0 <= h\ x ==>$
$(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * f(k\ x\ y)) =o$
$(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * g(k\ x\ y)) +o$
$O(\%x.\ SUM\ y : A\ x.\ (l\ x\ y) * h(k\ x\ y))$
⟨*proof*⟩

5.3 Misc useful stuff

lemma *bigo-useful-intro*: $A <= O(f) ==> B <= O(f) ==>$
$A \oplus B <= O(f)$
⟨*proof*⟩

lemma *bigo-useful-add*: $f =o O(h) ==> g =o O(h) ==> f + g =o O(h)$
⟨*proof*⟩

lemma *bigo-useful-const-mult*: $(c::'a::linordered\text{-}field) ~= 0 ==>$
$(\%x.\ c) * f =o O(h) ==> f =o O(h)$
⟨*proof*⟩

lemma *bigo-fix*: $(\%x.\ f\ ((x::nat) + 1)) =o O(\%x.\ h(x + 1)) ==> f\ 0 = 0 ==>$
$f =o O(h)$
⟨*proof*⟩

lemma *bigo-fix2*:
$(\%x.\ f\ ((x::nat) + 1)) =o (\%x.\ g(x + 1)) +o O(\%x.\ h(x + 1)) ==>$
$f\ 0 = g\ 0 ==> f =o g +o O(h)$
⟨*proof*⟩

5.4 Less than or equal to

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

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

lemma *bigo-lesseq2*: $f =o O(h) ==> ALL\ x.\ abs\ (g\ x) <= f\ x ==>$
$g =o O(h)$
⟨*proof*⟩

lemma *bigo-lesseq3*: $f =o O(h) ==> ALL\ x.\ 0 <= g\ x ==> ALL\ x.\ g\ x <= f\ x ==>$
$g =o O(h)$
⟨*proof*⟩

lemma *bigo-lesseq4*: $f =o O(h) ==>$
$ALL\ x.\ 0 <= g\ x ==> ALL\ x.\ g\ x <= abs\ (f\ x) ==>$
$g =o O(h)$
⟨*proof*⟩

lemma *bigo-lesso1*: $ALL\ x.\ f\ x <= g\ x ==> f <o g =o O(h)$
⟨*proof*⟩

lemma *bigo-lesso2*: $f =o g +o O(h) ==>$
$ALL\ x.\ 0 <= k\ x ==> ALL\ x.\ k\ x <= f\ x ==>$
$k <o g =o O(h)$
⟨*proof*⟩

lemma *bigo-lesso3*: $f =o g +o O(h) ==>$
$ALL\ x.\ 0 <= k\ x ==> ALL\ x.\ g\ x <= k\ x ==>$
$f <o k =o O(h)$
⟨*proof*⟩

lemma *bigo-lesso4*: $f <o g =o O(k::'a=>'b::linordered\text{-}field) ==>$
$g =o h +o O(k) ==> f <o h =o O(k)$
⟨*proof*⟩

lemma *bigo-lesso5*: $f <o g =o O(h) ==>$
$EX\ C.\ ALL\ x.\ f\ x <= g\ x + C * abs(h\ x)$
⟨*proof*⟩

lemma *lesso-add*: $f <o g =o O(h) ==>$
$k <o l =o O(h) ==> (f + k) <o (g + l) =o O(h)$
⟨*proof*⟩

lemma *bigo-LIMSEQ1*: $f =o O(g) ==> g ----> 0 ==> f ----> (0::real)$
⟨*proof*⟩

lemma *bigo-LIMSEQ2*: $f =o g +o O(h) ==> h ----> 0 ==> f ----> a$
$==> g ----> (a::real)$
⟨*proof*⟩

end

6 Binomial: Binomial Coefficients

theory *Binomial*
imports *Complex-Main*
begin

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

primrec *binomial* :: *nat* ⇒ *nat* ⇒ *nat* (**infixl** *choose 65*) **where**
binomial-0: $(0\ choose\ k) = (if\ k = 0\ then\ 1\ else\ 0)$
| *binomial-Suc*: $(Suc\ n\ choose\ k) =$
$(if\ k = 0\ then\ 1\ else\ (n\ choose\ (k - 1)) + (n\ choose\ k))$

lemma *binomial-n-0* [*simp*]: $(n\ choose\ 0) = 1$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *binomial-Suc-Suc* [*simp*]:
(*Suc n choose Suc k*) = (*n choose k*) + (*n choose Suc k*)
⟨*proof*⟩

lemma *binomial-eq-0*: !!*k*. *n* < *k* ==> (*n choose k*) = *0*
⟨*proof*⟩

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

lemma *binomial-n-n* [*simp*]: (*n choose n*) = *1*
⟨*proof*⟩

lemma *binomial-Suc-n* [*simp*]: (*Suc n choose n*) = *Suc n*
⟨*proof*⟩

lemma *binomial-1* [*simp*]: (*n choose Suc 0*) = *n*
⟨*proof*⟩

lemma *zero-less-binomial*: *k* ≤ *n* ==> (*n choose k*) > *0*
⟨*proof*⟩

lemma *binomial-eq-0-iff*: (*n choose k* = *0*) = (*n*<*k*)
⟨*proof*⟩

lemma *zero-less-binomial-iff*: (*n choose k* > *0*) = (*k*≤*n*)
⟨*proof*⟩

lemma *Suc-times-binomial-eq*:
!!*k*. *k* ≤ *n* ==> *Suc n* * (*n choose k*) = (*Suc n choose Suc k*) * *Suc k*
⟨*proof*⟩

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

lemma *binomial-Suc-Suc-eq-times*:
k ≤ *n* ==> (*Suc n choose Suc k*) = (*Suc n* * (*n choose k*)) *div Suc k*
⟨*proof*⟩

Another version, with -1 instead of Suc.

lemma *times-binomial-minus1-eq*:
[|*k* ≤ *n*; *0*<*k*|] ==> (*n choose k*) * *k* = *n* * ((*n* − *1*) *choose* (*k* − *1*))
⟨*proof*⟩

6.1 Theorems about *choose*

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

lemma *card-s-0-eq-empty*:

finite A ==> card {B. B ⊆ A & card B = 0} = 1
⟨*proof*⟩

lemma *choose-deconstruct*: *finite M ==> x ∉ M*
==> {s. s <= insert x M & card(s) = Suc k}
= {s. s <= M & card(s) = Suc k} Un
{s. EX t. t <= M & card(t) = k & s = insert x t}
⟨*proof*⟩

lemma *finite-bex-subset*[*simp*]:
finite B ⟹ (!!A. A<=B ⟹ finite{x. P x A}) ⟹ finite{x. EX A<=B. P x A}
⟨*proof*⟩

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

lemma *constr-bij*:
[|*finite A*; *x ∉ A*|] *==>*
card {B. EX C. C <= A & card(C) = k & B = insert x C} =
card {B. B <= A & card(B) = k}
⟨*proof*⟩

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

lemma *n-sub-lemma*:
!!A. finite A ==> card {B. B <= A & card B = k} = (card A choose k)
⟨*proof*⟩

theorem *n-subsets*:
finite A ==> card {B. B <= A & card B = k} = (card A choose k)
⟨*proof*⟩

The binomial theorem (courtesy of Tobias Nipkow):

theorem *binomial*: *(a+b::nat)^n = (∑k=0..n. (n choose k) * a^k * b^(n−k))*
⟨*proof*⟩

6.2 Pochhammer's symbol : generalized raising factorial

definition *pochhammer (a::'a::comm-semiring-1) n = (if n = 0 then 1 else setprod (λn. a + of-nat n) {0 .. n − 1})*

lemma *pochhammer-0*[*simp*]: *pochhammer a 0 = 1*
⟨*proof*⟩

lemma *pochhammer-1*[*simp*]: *pochhammer a 1 = a* ⟨*proof*⟩
lemma *pochhammer-Suc0*[*simp*]: *pochhammer a (Suc 0) = a*
⟨*proof*⟩

lemma *pochhammer-Suc-setprod*: *pochhammer a (Suc n) = setprod (λn. a + of-nat n) {0 .. n}*

⟨*proof*⟩

lemma *setprod-nat-ivl-Suc*: *setprod f* {*0* .. *Suc n*} = *setprod f* {*0*..*n*} * *f* (*Suc n*)
⟨*proof*⟩

lemma *setprod-nat-ivl-1-Suc*: *setprod f* {*0* .. *Suc n*} = *f 0* * *setprod f* {*1*.. *Suc n*}
⟨*proof*⟩

lemma *pochhammer-Suc*: *pochhammer a* (*Suc n*) = *pochhammer a n* * (*a* + *of-nat n*)
⟨*proof*⟩

lemma *pochhammer-rec*: *pochhammer a* (*Suc n*) = *a* * *pochhammer* (*a* + *1*) *n*
⟨*proof*⟩

lemma *pochhammer-fact*: *of-nat* (*fact n*) = *pochhammer 1 n*
⟨*proof*⟩

lemma *pochhammer-of-nat-eq-0-lemma*: **assumes** *kn*: *k* > *n*
shows *pochhammer* (− (*of-nat n* :: 'a:: *idom*)) *k* = *0*
⟨*proof*⟩

lemma *pochhammer-of-nat-eq-0-lemma'*: **assumes** *kn*: *k* ≤ *n*
shows *pochhammer* (− (*of-nat n* :: 'a:: {*idom*, *ring-char-0*})) *k* ≠ *0*
⟨*proof*⟩

lemma *pochhammer-of-nat-eq-0-iff*:
shows *pochhammer* (− (*of-nat n* :: 'a:: {*idom*, *ring-char-0*})) *k* = *0* ⟷ *k* > *n*
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *pochhammer-eq-0-iff*:
pochhammer a n = (*0*::'a::*field-char-0*) ⟷ (*EX k* < *n* . *a* = − *of-nat k*)
⟨*proof*⟩

lemma *pochhammer-eq-0-mono*:
pochhammer a n = (*0*::'a::*field-char-0*) ⟹ *m* ≥ *n* ⟹ *pochhammer a m* = *0*
⟨*proof*⟩

lemma *pochhammer-neq-0-mono*:
pochhammer a m ≠ (*0*::'a::*field-char-0*) ⟹ *m* ≥ *n* ⟹ *pochhammer a n* ≠ *0*
⟨*proof*⟩

lemma *pochhammer-minus*:
assumes *kn*: *k* ≤ *n*
shows *pochhammer* (− *b*) *k* = ((− *1*) ^ *k* :: 'a::*comm-ring-1*) * *pochhammer* (*b*

$-$ *of-nat k + 1*) *k*
⟨*proof*⟩

lemma *pochhammer-minus'*:
assumes *kn*: $k \leq n$
shows *pochhammer* (*b* $-$ *of-nat k + 1*) *k* = ((*−* 1) ^ *k* :: *'a*::*comm-ring-1*) $*$ *pochhammer* (*−* *b*) *k*
⟨*proof*⟩

lemma *pochhammer-same*: *pochhammer* (*−* *of-nat n*) *n* = ((*−* 1) ^ *n* :: *'a*::*comm-ring-1*) $*$ *of-nat* (*fact n*)
⟨*proof*⟩

6.3 Generalized binomial coefficients

definition *gbinomial* :: *'a*::*field-char-0* ⇒ *nat* ⇒ *'a* (**infixl** *gchoose 65*)
where *a gchoose n* = (*if n* = *0 then 1 else* (*setprod* (*λi*. *a* $-$ *of-nat i*) {*0* .. *n* $-$ *1*}) / *of-nat* (*fact n*))

lemma *gbinomial-0*[*simp*]: *a gchoose 0* = *1 0 gchoose* (*Suc n*) = *0*
⟨*proof*⟩

lemma *gbinomial-pochhammer*: *a gchoose n* = (*−* *1*) ^ *n* $*$ *pochhammer* (*−* *a*) *n* / *of-nat* (*fact n*)
⟨*proof*⟩

lemma *binomial-fact-lemma*:
k ≤ *n* ⟹ *fact k* $*$ *fact* (*n* $-$ *k*) $*$ (*n choose k*) = *fact n*
⟨*proof*⟩

lemma *binomial-fact*:
assumes *kn*: $k \leq n$
shows (*of-nat* (*n choose k*) :: *'a*::*field-char-0*) = *of-nat* (*fact n*) / (*of-nat* (*fact k*) $*$ *of-nat* (*fact* (*n* $-$ *k*)))
⟨*proof*⟩

lemma *binomial-gbinomial*: *of-nat* (*n choose k*) = *of-nat n gchoose k*
⟨*proof*⟩

lemma *gbinomial-1*[*simp*]: *a gchoose 1* = *a*
⟨*proof*⟩

lemma *gbinomial-Suc0*[*simp*]: *a gchoose* (*Suc 0*) = *a*
⟨*proof*⟩

lemma *gbinomial-mult-1*: *a* $*$ (*a gchoose n*) = *of-nat n* $*$ (*a gchoose n*) + *of-nat* (*Suc n*) $*$ (*a gchoose* (*Suc n*)) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *gbinomial-mult-1'*: $(a\ gchoose\ n) * a = of\text{-}nat\ n * (a\ gchoose\ n) + of\text{-}nat\ (Suc\ n) * (a\ gchoose\ (Suc\ n))$
⟨*proof*⟩

lemma *gbinomial-Suc*: $a\ gchoose\ (Suc\ k) = (setprod\ (\lambda i.\ a - of\text{-}nat\ i)\ \{0\ ..\ k\})\ /\ of\text{-}nat\ (fact\ (Suc\ k))$
⟨*proof*⟩

lemma *gbinomial-mult-fact*:
$(of\text{-}nat\ (fact\ (Suc\ k)) :: 'a) * ((a::'a::field\text{-}char\text{-}0)\ gchoose\ (Suc\ k)) = (setprod\ (\lambda i.\ a - of\text{-}nat\ i)\ \{0\ ..\ k\})$
⟨*proof*⟩

lemma *gbinomial-mult-fact'*:
$((a::'a::field\text{-}char\text{-}0)\ gchoose\ (Suc\ k)) * (of\text{-}nat\ (fact\ (Suc\ k)) :: 'a) = (setprod\ (\lambda i.\ a - of\text{-}nat\ i)\ \{0\ ..\ k\})$
⟨*proof*⟩

lemma *gbinomial-Suc-Suc*: $((a::'a::field\text{-}char\text{-}0) + 1)\ gchoose\ (Suc\ k) = a\ gchoose\ k + (a\ gchoose\ (Suc\ k))$
⟨*proof*⟩

lemma *binomial-symmetric*: **assumes** *kn*: $k \leq n$
shows $n\ choose\ k = n\ choose\ (n - k)$
⟨*proof*⟩

end

7 Bit: The Field of Integers mod 2

theory *Bit*
imports *Main*
begin

7.1 Bits as a datatype

typedef (**open**) *bit* = *UNIV* :: *bool set* ⟨*proof*⟩

instantiation *bit* :: {*zero*, *one*}
begin

definition *zero-bit-def*:
0 = *Abs-bit False*

definition *one-bit-def*:
1 = *Abs-bit True*

instance ⟨*proof*⟩

end

rep-datatype (*bit*) *0*::*bit* *1*::*bit*
⟨*proof*⟩

lemma *bit-not-0-iff* [*iff*]: (*x*::*bit*) ≠ *0* ⟷ *x* = *1*
⟨*proof*⟩

lemma *bit-not-1-iff* [*iff*]: (*x*::*bit*) ≠ *1* ⟷ *x* = *0*
⟨*proof*⟩

7.2 Type *bit* forms a field

instantiation *bit* :: *field-inverse-zero*
begin

definition *plus-bit-def*:
x + *y* = *bit-case y* (*bit-case 1 0 y*) *x*

definition *times-bit-def*:
x ∗ *y* = *bit-case 0 y x*

definition *uminus-bit-def* [*simp*]:
− *x* = (*x* :: *bit*)

definition *minus-bit-def* [*simp*]:
x − *y* = (*x* + *y* :: *bit*)

definition *inverse-bit-def* [*simp*]:
inverse x = (*x* :: *bit*)

definition *divide-bit-def* [*simp*]:
x / *y* = (*x* ∗ *y* :: *bit*)

lemmas *field-bit-defs* =
plus-bit-def times-bit-def minus-bit-def uminus-bit-def
divide-bit-def inverse-bit-def

instance ⟨*proof*⟩

end

lemma *bit-add-self*: *x* + *x* = (*0* :: *bit*)
⟨*proof*⟩

lemma *bit-mult-eq-1-iff* [*simp*]: *x* ∗ *y* = (*1* :: *bit*) ⟷ *x* = *1* ∧ *y* = *1*
⟨*proof*⟩

Not sure whether the next two should be simp rules.

lemma *bit-add-eq-0-iff*: $x + y = (0 :: bit) \longleftrightarrow x = y$
⟨*proof*⟩

lemma *bit-add-eq-1-iff*: $x + y = (1 :: bit) \longleftrightarrow x \neq y$
⟨*proof*⟩

7.3 Numerals at type *bit*

instantiation *bit* :: *number-ring*
begin

definition *number-of-bit-def*:
$(number\text{-}of\ w :: bit) = of\text{-}int\ w$

instance ⟨*proof*⟩

end

All numerals reduce to either 0 or 1.

lemma *bit-minus1* [*simp*]: $-1 = (1 :: bit)$
⟨*proof*⟩

lemma *bit-number-of-even* [*simp*]: $number\text{-}of\ (Int.Bit0\ w) = (0 :: bit)$
⟨*proof*⟩

lemma *bit-number-of-odd* [*simp*]: $number\text{-}of\ (Int.Bit1\ w) = (1 :: bit)$
⟨*proof*⟩

end

8 Boolean-Algebra: Boolean Algebras

theory *Boolean-Algebra*
imports *Main*
begin

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

assumes *conj-disj-distrib*: $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
assumes *disj-conj-distrib*: $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
assumes *conj-one-right* [*simp*]: $x \sqcap \mathbf{1} = x$
assumes *disj-zero-right* [*simp*]: $x \sqcup \mathbf{0} = x$
assumes *conj-cancel-right* [*simp*]: $x \sqcap \sim x = \mathbf{0}$
assumes *disj-cancel-right* [*simp*]: $x \sqcup \sim x = \mathbf{1}$

sublocale *boolean* $<$ *conj*!: *abel-semigroup conj* ⟨*proof*⟩

sublocale *boolean* $<$ *disj*!: *abel-semigroup disj* ⟨*proof*⟩

context *boolean*
begin

lemmas *conj-left-commute* = *conj.left-commute*

lemmas *disj-left-commute* = *disj.left-commute*

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

lemma *dual*: *boolean disj conj compl one zero*
⟨*proof*⟩

8.1 Complement

lemma *complement-unique*:
assumes *1*: $a \sqcap x = \mathbf{0}$
assumes *2*: $a \sqcup x = \mathbf{1}$
assumes *3*: $a \sqcap y = \mathbf{0}$
assumes *4*: $a \sqcup y = \mathbf{1}$
shows $x = y$
⟨*proof*⟩

lemma *compl-unique*: $[\![x \sqcap y = \mathbf{0};\ x \sqcup y = \mathbf{1}]\!] \Longrightarrow \sim x = y$
⟨*proof*⟩

lemma *double-compl* [*simp*]: $\sim (\sim x) = x$
⟨*proof*⟩

lemma *compl-eq-compl-iff* [*simp*]: $(\sim x = \sim y) = (x = y)$
⟨*proof*⟩

8.2 Conjunction

lemma *conj-absorb* [*simp*]: $x \sqcap x = x$
⟨*proof*⟩

lemma *conj-zero-right* [*simp*]: $x \sqcap \mathbf{0} = \mathbf{0}$
⟨*proof*⟩

lemma *compl-one* [*simp*]: $\sim \mathbf{1} = \mathbf{0}$
⟨*proof*⟩

lemma *conj-zero-left* [*simp*]: $\mathbf{0} \sqcap x = \mathbf{0}$
⟨*proof*⟩

lemma *conj-one-left* [*simp*]: $\mathbf{1} \sqcap x = x$
⟨*proof*⟩

lemma *conj-cancel-left* [*simp*]: $\sim x \sqcap x = \mathbf{0}$
⟨*proof*⟩

lemma *conj-left-absorb* [*simp*]: $x \sqcap (x \sqcap y) = x \sqcap y$
⟨*proof*⟩

lemma *conj-disj-distrib2*:
$(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
⟨*proof*⟩

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

8.3 Disjunction

lemma *disj-absorb* [*simp*]: $x \sqcup x = x$
⟨*proof*⟩

lemma *disj-one-right* [*simp*]: $x \sqcup \mathbf{1} = \mathbf{1}$
⟨*proof*⟩

lemma *compl-zero* [*simp*]: $\sim \mathbf{0} = \mathbf{1}$
⟨*proof*⟩

lemma *disj-zero-left* [*simp*]: $\mathbf{0} \sqcup x = x$
⟨*proof*⟩

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$
⟨*proof*⟩

lemma *disj-cancel-left* [*simp*]: $\sim x \sqcup x = \mathbf{1}$
⟨*proof*⟩

lemma *disj-left-absorb* [*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
⟨*proof*⟩

lemma *disj-conj-distrib2*:
$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
⟨*proof*⟩

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

8.4 De Morgan's Laws

lemma *de-Morgan-conj* [*simp*]: $\sim (x \sqcap y) = \sim x \sqcup \sim y$
⟨*proof*⟩

lemma *de-Morgan-disj* [*simp*]: $\sim (x \sqcup y) = \sim x \sqcap \sim y$
⟨*proof*⟩

end

8.5 Symmetric Difference

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

sublocale *boolean-xor* < *xor*!: *abel-semigroup xor* ⟨*proof*⟩

context *boolean-xor*
begin

lemmas *xor-assoc* = *xor.assoc*
lemmas *xor-commute* = *xor.commute*
lemmas *xor-left-commute* = *xor.left-commute*

lemmas *xor-ac* = *xor.assoc xor.commute xor.left-commute*

lemma *xor-def2*:
$x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$
⟨*proof*⟩

lemma *xor-zero-right* [*simp*]: $x \oplus \mathbf{0} = x$
⟨*proof*⟩

lemma *xor-zero-left* [*simp*]: $\mathbf{0} \oplus x = x$
⟨*proof*⟩

lemma *xor-one-right* [*simp*]: $x \oplus \mathbf{1} = \sim x$
⟨*proof*⟩

lemma *xor-one-left* [*simp*]: $\mathbf{1} \oplus x = \sim x$
⟨*proof*⟩

lemma *xor-self* [*simp*]: $x \oplus x = \mathbf{0}$
⟨*proof*⟩

lemma *xor-left-self* [*simp*]: $x \oplus (x \oplus y) = y$
⟨*proof*⟩

lemma *xor-compl-left* [*simp*]: $\sim x \oplus y = \sim (x \oplus y)$
⟨*proof*⟩

lemma *xor-compl-right* [*simp*]: $x \oplus \sim y = \sim (x \oplus y)$
⟨*proof*⟩

lemma *xor-cancel-right*: $x \oplus \sim x = \mathbf{1}$
⟨*proof*⟩

lemma *xor-cancel-left*: $\sim x \oplus x = \mathbf{1}$
⟨*proof*⟩

lemma *conj-xor-distrib*: $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
⟨*proof*⟩

lemma *conj-xor-distrib2*:
$(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$
⟨*proof*⟩

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

end

end

9 Product-ord: Order on product types

theory *Product-ord*
imports *Main*
begin

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

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

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

instance ⟨*proof*⟩

end

lemma [*code*]:
$(x1::'a::\{ord,\ eq\},\ y1) \leq (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 \leq y2$
$(x1::'a::\{ord,\ eq\},\ y1) < (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 < y2$
⟨*proof*⟩

instance $*$ :: (*preorder*, *preorder*) *preorder* ⟨*proof*⟩

instance $*$ :: (*order*, *order*) *order* ⟨*proof*⟩

instance $*$ :: (*linorder*, *linorder*) *linorder* ⟨*proof*⟩

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

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

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

instance ⟨*proof*⟩

end

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

definition
bot-prod-def: $bot = (bot,\ bot)$

instance ⟨*proof*⟩

end

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

definition
top-prod-def: $top = (top,\ top)$

instance ⟨*proof*⟩

end

end

10 Char-nat: Mapping between characters and natural numbers

theory *Char-nat*
imports *List Main*
begin

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

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

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

lemma *nibble-of-nat-norm*:
nibble-of-nat (*n mod 16*) = *nibble-of-nat n*

⟨*proof*⟩

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

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

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

lemma *nibble-of-nat-of-nibble*: *nibble-of-nat* (*nat-of-nibble n*) = *n*
⟨*proof*⟩

lemma *nat-of-nibble-of-nat*: *nat-of-nibble* (*nibble-of-nat n*) = *n mod 16*
⟨*proof*⟩

lemma *inj-nat-of-nibble*: *inj nat-of-nibble*
⟨*proof*⟩

lemma *nat-of-nibble-eq*: *nat-of-nibble n* = *nat-of-nibble m* ⟷ *n* = *m*
⟨*proof*⟩

lemma *nat-of-nibble-less-16*: *nat-of-nibble n* < *16*
⟨*proof*⟩

lemma *nat-of-nibble-div-16*: *nat-of-nibble n div 16* = *0*
⟨*proof*⟩

Conversion between chars and nats.

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

lemma *nibble-of-pair* [*code*]:
nibble-pair-of-nat n = (*nibble-of-nat* (*n div 16*), *nibble-of-nat n*)
⟨*proof*⟩

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

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

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

lemma *Char-char-of-nat*:
Char n m = *char-of-nat* (*nat-of-nibble n* ∗ *16* + *nat-of-nibble m*)
⟨*proof*⟩

lemma *char-of-nat-of-char*:
char-of-nat (*nat-of-char c*) = *c*
⟨*proof*⟩

lemma *nat-of-char-of-nat*:
nat-of-char (*char-of-nat n*) = *n mod 256*
⟨*proof*⟩

lemma *nibble-pair-of-nat-char*:
nibble-pair-of-nat (*nat-of-char* (*Char n m*)) = (*n*, *m*)
⟨*proof*⟩

Code generator setup

code-modulename *SML*
Char-nat String

code-modulename *OCaml*
Char-nat String

code-modulename *Haskell*
Char-nat String

end

11 Char-ord: Order on characters

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

instantiation *nibble* :: *linorder*
begin

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

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

instance ⟨*proof*⟩

end

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

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

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

instance ⟨*proof*⟩

end

instantiation *char* :: *linorder*
begin

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

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

instance
⟨*proof*⟩

end

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

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

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

instance ⟨*proof*⟩

end

lemma [*simp*, *code*]:
shows *char-less-eq-simp*: *Char n1 m1* ≤ *Char n2 m2* ⟷ *n1* < *n2* ∨ *n1* = *n2* ∧ *m1* ≤ *m2*
and *char-less-simp*: *Char n1 m1* < *Char n2 m2* ⟷ *n1* < *n2* ∨ *n1* = *n2* ∧ *m1* < *m2*
⟨*proof*⟩

end

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

theory *Code-Char*
imports *List Code-Evaluation Main*
begin

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

⟨*ML*⟩

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

code-reserved *SML*
char

code-reserved *OCaml*
char

code-reserved *Scala*
char

code-const *eq-class.eq* :: *char* ⇒ *char* ⇒ *bool*
(*SML* !((- : *char*) = -))
(*OCaml* !((- : *char*) = -))

(*Haskell* **infixl** *4* ==)
(*Scala* **infixl** *5* ==)

code-const *Code-Evaluation.term-of* :: *char* ⇒ *term*
(*Eval HOLogic.mk'-char*/ (*IntInf.fromInt*/ (*Char.ord*/ -)))

definition *implode* :: *string* ⇒ *String.literal* **where**
implode = *STR*

primrec *explode* :: *String.literal* ⇒ *string* **where**
explode (*STR s*) = *s*

lemma [*code*]:
literal-case f s = *f* (*explode s*)
literal-rec f s = *f* (*explode s*)
⟨*proof*⟩

code-reserved *SML String*

code-const *implode*
(*SML String.implode*)
(*OCaml failwith*/ *implode*)
(*Haskell* -)
(*Scala List.toString*((-)))

code-const *explode*
(*SML String.explode*)
(*OCaml failwith*/ *explode*)
(*Haskell* -)
(*Scala List.fromString*((-)))

end

13 Code-Integer: Pretty integer literals for code generation

theory *Code-Integer*
imports *Main*
begin

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

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

(*Haskell Integer*)
(*Scala BigInt*)

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

⟨*ML*⟩

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

code-const *Int.pred*
(*SML IntInf.*− ((-), *1*))
(*OCaml Big'-int.pred'-big'-int*)
(*Haskell* !(-/ −/ *1*))
(*Scala* !(-/ −/ *1*))

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

code-const *op* + :: *int* ⇒ *int* ⇒ *int*
(*SML IntInf.*+ ((-), (-)))
(*OCaml Big'-int.add'-big'-int*)
(*Haskell* **infixl** *6* +)
(*Scala* **infixl** *7* +)

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

(*Scala* !(− -))

code-const *op* − :: *int* ⇒ *int* ⇒ *int*
(*SML IntInf*.− ((-), (-)))
(*OCaml Big'-int.sub'-big'-int*)
(*Haskell* **infixl** *6* −)
(*Scala* **infixl** *7* −)

code-const *op* ∗ :: *int* ⇒ *int* ⇒ *int*
(*SML IntInf*.∗ ((-), (-)))
(*OCaml Big'-int.mult'-big'-int*)
(*Haskell* **infixl** *7* ∗)
(*Scala* **infixl** *8* ∗)

code-const *pdivmod*
(*SML IntInf.divMod*/ (*IntInf.abs* -,/ *IntInf.abs* -))
(*OCaml Big'-int.quomod'-big'-int*/ (*Big'-int.abs'-big'-int* -)/ (*Big'-int.abs'-big'-int* -))
(*Haskell divMod*/ (*abs* -)/ (*abs* -))
(*Scala* !(*-.abs '*/% *-.abs*))

code-const *eq-class.eq* :: *int* ⇒ *int* ⇒ *bool*
(*SML* !((- : *IntInf.int*) = -))
(*OCaml Big'-int.eq'-big'-int*)
(*Haskell* **infixl** *4* ==)
(*Scala* **infixl** *5* ==)

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

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

code-const *Code-Numeral.int-of*
(*SML IntInf.fromInt*)
(*OCaml* -)
(*Haskell toEnum*)
(*Scala* !*BigInt*((-)))

Evaluation

code-const *Code-Evaluation.term-of* :: *int* ⇒ *term*
(*Eval HOLogic.mk'-number*/ *HOLogic.intT*)

end

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

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

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

lemma [*code*]:
nat-of-char = nat o int-of-char
⟨*proof*⟩

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

lemma [*code*]:
char-of-nat = char-of-int o int
⟨*proof*⟩

code-const *int-of-char* **and** *char-of-int*
(*SML* !(*IntInf.fromInt o Char.ord*) **and** !(*Char.chr o IntInf.toInt*))
(*OCaml Big'-int.big'-int'-of'-int* (*Char.code* -) **and** *Char.chr* (*Big'-int.int'-of'-big'-int* -))
(*Haskell toInteger* (*fromEnum* (- :: *Char*)) **and** !(*let chr k* | (*0* <= *k* && *k* < *256*) = *toEnum k* :: *Char in chr* . *fromInteger*))
(*Scala BigInt*(-.*toInt*) **and** !((*k*: *BigInt*) => *if* (*BigInt*(*0*) <= *k* && *k* < *BigInt*(*256*)) *k.charValue else error*(*character value out of range*)))

end

15 Continuity: Continuity and iterations (of set transformers)

theory *Continuity*
imports *Transitive-Closure Main*
begin

15.1 Continuity for complete lattices

definition
chain :: (*nat* ⇒ *'a*::*complete-lattice*) ⇒ *bool* **where**
chain M ⟷ (∀ *i*. *M i* ≤ *M* (*Suc i*))

definition
continuous :: ('*a*::*complete-lattice* ⇒ '*a*::*complete-lattice*) ⇒ *bool* **where**
continuous F ⟷ (∀ *M*. *chain M* ⟶ *F* (*SUP i*. *M i*) = (*SUP i*. *F* (*M i*)))

lemma *SUP-nat-conv*:
(*SUP n*. *M n*) = *sup* (*M 0*) (*SUP n*. *M*(*Suc n*))
⟨*proof*⟩

lemma *continuous-mono*: **fixes** *F* :: '*a*::*complete-lattice* ⇒ '*a*::*complete-lattice*
assumes *continuous F* **shows** *mono F*
⟨*proof*⟩

lemma *continuous-lfp*:
assumes *continuous F* **shows** *lfp F* = (*SUP i*. (*F* ^^ *i*) *bot*)
⟨*proof*⟩

The following development is just for sets but presents an up and a down version of chains and continuity and covers *gfp*.

15.2 Chains

definition
up-chain :: (*nat* => '*a set*) => *bool* **where**
up-chain F = (∀ *i*. *F i* ⊆ *F* (*Suc i*))

lemma *up-chainI*: (!!*i*. *F i* ⊆ *F* (*Suc i*)) ==> *up-chain F*
⟨*proof*⟩

lemma *up-chainD*: *up-chain F* ==> *F i* ⊆ *F* (*Suc i*)
⟨*proof*⟩

lemma *up-chain-less-mono*:
up-chain F ==> *x* < *y* ==> *F x* ⊆ *F y*
⟨*proof*⟩

lemma *up-chain-mono*: *up-chain F* ==> *x* ≤ *y* ==> *F x* ⊆ *F y*
⟨*proof*⟩

definition
down-chain :: (*nat* => '*a set*) => *bool* **where**
down-chain F = (∀ *i*. *F* (*Suc i*) ⊆ *F i*)

lemma *down-chainI*: (!!*i*. *F* (*Suc i*) ⊆ *F i*) ==> *down-chain F*
⟨*proof*⟩

lemma *down-chainD*: *down-chain F* ==> *F* (*Suc i*) ⊆ *F i*
⟨*proof*⟩

lemma *down-chain-less-mono*:
down-chain F ==> x < y ==> F y ⊆ F x
⟨*proof*⟩

lemma *down-chain-mono*: *down-chain F ==> x ≤ y ==> F y ⊆ F x*
⟨*proof*⟩

15.3 Continuity

definition
up-cont :: (*'a set => 'a set*) => *bool* **where**
up-cont f = (∀ *F*. *up-chain F --> f* (⋃(*range F*)) = ⋃(*f ' range F*))

lemma *up-contI*:
(!!*F*. *up-chain F ==> f* (⋃(*range F*)) = ⋃(*f ' range F*)) ==> *up-cont f*
⟨*proof*⟩

lemma *up-contD*:
up-cont f ==> up-chain F ==> f (⋃(*range F*)) = ⋃(*f ' range F*)
⟨*proof*⟩

lemma *up-cont-mono*: *up-cont f ==> mono f*
⟨*proof*⟩

definition
down-cont :: (*'a set => 'a set*) => *bool* **where**
down-cont f =
(∀ *F*. *down-chain F --> f* (*Inter* (*range F*)) = *Inter* (*f ' range F*))

lemma *down-contI*:
(!!*F*. *down-chain F ==> f* (*Inter* (*range F*)) = *Inter* (*f ' range F*)) ==>
down-cont f
⟨*proof*⟩

lemma *down-contD*: *down-cont f ==> down-chain F ==>*
f (*Inter* (*range F*)) = *Inter* (*f ' range F*)
⟨*proof*⟩

lemma *down-cont-mono*: *down-cont f ==> mono f*
⟨*proof*⟩

15.4 Iteration

definition
up-iterate :: (*'a set => 'a set*) => *nat => 'a set* **where**
up-iterate f n = (*f ^^ n*) {}

lemma *up-iterate-0* [*simp*]: *up-iterate f 0 = {}*
⟨*proof*⟩

lemma *up-iterate-Suc* [*simp*]: *up-iterate f (Suc i) = f (up-iterate f i)*
⟨*proof*⟩

lemma *up-iterate-chain*: *mono F ==> up-chain (up-iterate F)*
⟨*proof*⟩

lemma *UNION-up-iterate-is-fp*:
up-cont F ==>
F (UNION UNIV (up-iterate F)) = UNION UNIV (up-iterate F)
⟨*proof*⟩

lemma *UNION-up-iterate-lowerbound*:
mono F ==> F P = P ==> UNION UNIV (up-iterate F) ⊆ P
⟨*proof*⟩

lemma *UNION-up-iterate-is-lfp*:
up-cont F ==> lfp F = UNION UNIV (up-iterate F)
⟨*proof*⟩

definition
down-iterate :: *('a set => 'a set) => nat => 'a set* **where**
down-iterate f n = (f ^^ n) UNIV

lemma *down-iterate-0* [*simp*]: *down-iterate f 0 = UNIV*
⟨*proof*⟩

lemma *down-iterate-Suc* [*simp*]:
down-iterate f (Suc i) = f (down-iterate f i)
⟨*proof*⟩

lemma *down-iterate-chain*: *mono F ==> down-chain (down-iterate F)*
⟨*proof*⟩

lemma *INTER-down-iterate-is-fp*:
down-cont F ==>
F (INTER UNIV (down-iterate F)) = INTER UNIV (down-iterate F)
⟨*proof*⟩

lemma *INTER-down-iterate-upperbound*:
mono F ==> F P = P ==> P ⊆ INTER UNIV (down-iterate F)
⟨*proof*⟩

lemma *INTER-down-iterate-is-gfp*:
down-cont F ==> gfp F = INTER UNIV (down-iterate F)
⟨*proof*⟩

end

16 ContNotDenum: Non-denumerability of the Continuum.

theory *ContNotDenum*
imports *Complex-Main*
begin

16.1 Abstract

The following document presents a proof that the Continuum is uncountable. It is formalised in the Isabelle/Isar theorem proving system.

Theorem: The Continuum $\mathbb{R}$ is not denumerable. In other words, there does not exist a function f:$\mathbb{N}\Rightarrow\mathbb{R}$ such that f is surjective.

Outline: An elegant informal proof of this result uses Cantor's Diagonalisation argument. The proof presented here is not this one. First we formalise some properties of closed intervals, then we prove the Nested Interval Property. This property relies on the completeness of the Real numbers and is the foundation for our argument. Informally it states that an intersection of countable closed intervals (where each successive interval is a subset of the last) is non-empty. We then assume a surjective function f:$\mathbb{N}\Rightarrow\mathbb{R}$ exists and find a real x such that x is not in the range of f by generating a sequence of closed intervals then using the NIP.

16.2 Closed Intervals

This section formalises some properties of closed intervals.

16.2.1 Definition

definition
closed-int :: *real* ⇒ *real* ⇒ *real set* **where**
closed-int x y = $\{z.\ x \leq z \wedge z \leq y\}$

16.2.2 Properties

lemma *closed-int-subset*:
assumes *xy*: *x1* ≥ *x0 y1* ≤ *y0*
shows *closed-int x1 y1* ⊆ *closed-int x0 y0*
⟨*proof*⟩

lemma *closed-int-least*:
assumes *a*: *a* ≤ *b*

shows $a \in$ *closed-int a b* $\wedge$ $(\forall x \in$ *closed-int a b*. $a \leq x)$
⟨*proof*⟩

lemma *closed-int-most*:
assumes *a*: $a \leq b$
shows $b \in$ *closed-int a b* $\wedge$ $(\forall x \in$ *closed-int a b*. $x \leq b)$
⟨*proof*⟩

lemma *closed-not-empty*:
shows $a \leq b \Longrightarrow \exists x.\ x \in$ *closed-int a b*
⟨*proof*⟩

lemma *closed-mem*:
assumes $a \leq c$ **and** $c \leq b$
shows $c \in$ *closed-int a b*
⟨*proof*⟩

lemma *closed-subset*:
assumes *ac*: $a \leq b \;\; c \leq d$
assumes *closed*: *closed-int a b* $\subseteq$ *closed-int c d*
shows $b \geq c$
⟨*proof*⟩

16.3 Nested Interval Property

theorem *NIP*:
fixes *f*::*nat* $\Rightarrow$ *real set*
assumes *subset*: $\forall n.\ f\ (Suc\ n) \subseteq f\ n$
and *closed*: $\forall n.\ \exists a\ b.\ f\ n =$ *closed-int a b* $\wedge\ a \leq b$
shows $(\bigcap n.\ f\ n) \neq \{\}$
⟨*proof*⟩

16.4 Generating the intervals

16.4.1 Existence of non-singleton closed intervals

This lemma asserts that given any non-singleton closed interval (a,b) and any element c, there exists a closed interval that is a subset of (a,b) and that does not contain c and is a non-singleton itself.

lemma *closed-subset-ex*:
fixes *c*::*real*
assumes *alb*: $a < b$
shows
$\exists ka\ kb.\ ka < kb \wedge$ *closed-int ka kb* $\subseteq$ *closed-int a b* $\wedge\ c \notin$ (*closed-int ka kb*)
⟨*proof*⟩

16.5 newInt: Interval generation

Given a function f:ℕ⇒ℝ, newInt (Suc n) f returns a closed interval such that $newInt\ (Suc\ n)\ f \subseteq newInt\ n\ f$ and does not contain $f\ (Suc\ n)$. With the base case defined such that $(f\ 0) \notin newInt\ 0\ f$.

16.5.1 Definition

primrec *newInt* :: *nat* ⇒ (*nat* ⇒ *real*) ⇒ (*real set*) **where**
  newInt 0 f = *closed-int* (*f 0* + *1*) (*f 0* + *2*)
  | *newInt* (*Suc n*) *f* =
     (*SOME e*. (∃ *e1 e2*.
      e1 < *e2* ∧
      e = *closed-int e1 e2* ∧
      e ⊆ (*newInt n f*) ∧
      (*f* (*Suc n*)) ∉ *e*)
)

declare *newInt.simps* [*code del*]

16.5.2 Properties

We now show that every application of newInt returns an appropriate interval.

lemma *newInt-ex*:
  ∃ *a b*. *a* < *b* ∧
   newInt (*Suc n*) *f* = *closed-int a b* ∧
   newInt (*Suc n*) *f* ⊆ *newInt n f* ∧
   f (*Suc n*) ∉ *newInt* (*Suc n*) *f*
⟨*proof*⟩

lemma *newInt-subset*:
  newInt (*Suc n*) *f* ⊆ *newInt n f*
  ⟨*proof*⟩

Another fundamental property is that no element in the range of f is in the intersection of all closed intervals generated by newInt.

lemma *newInt-inter*:
  ∀ *n*. *f n* ∉ (⋂ *n*. *newInt n f*)
⟨*proof*⟩

lemma *newInt-notempty*:
  (⋂ *n*. *newInt n f*) ≠ {}
⟨*proof*⟩

16.6 Final Theorem

theorem *real-non-denum*:

shows $\neg\ (\exists f{::}nat \Rightarrow real.\ surj\ f)$
⟨*proof*⟩

end

17 FrechetDeriv: Frechet Derivative

theory *FrechetDeriv*
imports *Lim Complex-Main*
begin

definition
fderiv ::
$['a{::}real\text{-}normed\text{-}vector \Rightarrow 'b{::}real\text{-}normed\text{-}vector,\ 'a,\ 'a \Rightarrow 'b] \Rightarrow bool$
— Frechet derivative: D is derivative of function f at x
((*FDERIV* (-)/ (-)/ :> (-)) [*1000*, *1000*, *60*] *60*) **where**
FDERIV f x :> D = (*bounded-linear D* ∧
(λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *D h*) / *norm h*) −− *0* −−> *0*)

lemma *FDERIV-I*:
⟦*bounded-linear D*; (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *D h*) / *norm h*) −− *0* −−> *0*⟧
⟹ *FDERIV f x :> D*
⟨*proof*⟩

lemma *FDERIV-D*:
FDERIV f x :> D ⟹ (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *D h*) / *norm h*) −− *0* −−> *0*
⟨*proof*⟩

lemma *FDERIV-bounded-linear*: *FDERIV f x :> D* ⟹ *bounded-linear D*
⟨*proof*⟩

lemma *bounded-linear-zero*:
bounded-linear (λ*x*::'*a*::*real-normed-vector*. *0*::'*b*::*real-normed-vector*)
⟨*proof*⟩

lemma *FDERIV-const*: *FDERIV* (λ*x*. *k*) *x* :> (λ*h*. *0*)
⟨*proof*⟩

lemma *bounded-linear-ident*:
bounded-linear (λ*x*::'*a*::*real-normed-vector*. *x*)
⟨*proof*⟩

lemma *FDERIV-ident*: *FDERIV* (λ*x*. *x*) *x* :> (λ*h*. *h*)
⟨*proof*⟩

17.1 Addition

lemma *bounded-linear-add*:
assumes *bounded-linear f*
assumes *bounded-linear g*
shows *bounded-linear* $(\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

lemma *norm-ratio-ineq*:
fixes $x\ y :: 'a$::*real-normed-vector*
fixes $h :: 'b$::*real-normed-vector*
shows $norm\ (x + y)\ /\ norm\ h \leq norm\ x\ /\ norm\ h + norm\ y\ /\ norm\ h$
⟨*proof*⟩

lemma *FDERIV-add*:
assumes *f*: $FDERIV\ f\ x :> F$
assumes *g*: $FDERIV\ g\ x :> G$
shows $FDERIV\ (\lambda x.\ f\ x + g\ x)\ x :> (\lambda h.\ F\ h + G\ h)$
⟨*proof*⟩

17.2 Subtraction

lemma *bounded-linear-minus*:
assumes *bounded-linear f*
shows *bounded-linear* $(\lambda x.\ -\ f\ x)$
⟨*proof*⟩

lemma *FDERIV-minus*:
$FDERIV\ f\ x :> F \Longrightarrow FDERIV\ (\lambda x.\ -\ f\ x)\ x :> (\lambda h.\ -\ F\ h)$
⟨*proof*⟩

lemma *FDERIV-diff*:
$[\![FDERIV\ f\ x :> F;\ FDERIV\ g\ x :> G]\!]$
$\Longrightarrow FDERIV\ (\lambda x.\ f\ x - g\ x)\ x :> (\lambda h.\ F\ h - G\ h)$
⟨*proof*⟩

17.3 Continuity

lemma *FDERIV-isCont*:
assumes *f*: $FDERIV\ f\ x :> F$
shows $isCont\ f\ x$
⟨*proof*⟩

17.4 Composition

lemma *real-divide-cancel-lemma*:
fixes $a\ b\ c :: real$
shows $(b = 0 \Longrightarrow a = 0) \Longrightarrow (a\ /\ b) * (b\ /\ c) = a\ /\ c$
⟨*proof*⟩

lemma *bounded-linear-compose*:
assumes *bounded-linear f*
assumes *bounded-linear g*
shows *bounded-linear* $(\lambda x.\ f\ (g\ x))$
⟨*proof*⟩

lemma *FDERIV-compose*:
fixes $f :: 'a$::*real-normed-vector* $\Rightarrow$ $'b$::*real-normed-vector*
fixes $g :: 'b$::*real-normed-vector* $\Rightarrow$ $'c$::*real-normed-vector*
assumes *f*: *FDERIV f x* :> *F*
assumes *g*: *FDERIV g* (*f x*) :> *G*
shows *FDERIV* $(\lambda x.\ g\ (f\ x))\ x$:> $(\lambda h.\ G\ (F\ h))$
⟨*proof*⟩

17.5 Product Rule

lemma (**in** *bounded-bilinear*) *FDERIV-lemma*:
$a' ** b' - a ** b - (a ** B + A ** b)$
$= a ** (b' - b - B) + (a' - a - A) ** b' + A ** (b' - b)$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *FDERIV*:
fixes $x :: 'd$::*real-normed-vector*
assumes *f*: *FDERIV f x* :> *F*
assumes *g*: *FDERIV g x* :> *G*
shows *FDERIV* $(\lambda x.\ f\ x ** g\ x)\ x$:> $(\lambda h.\ f\ x ** G\ h + F\ h ** g\ x)$
⟨*proof*⟩

lemmas *FDERIV-mult* = *mult.FDERIV*

lemmas *FDERIV-scaleR* = *scaleR.FDERIV*

17.6 Powers

lemma *FDERIV-power-Suc*:
fixes $x :: 'a$::{*real-normed-algebra,comm-ring-1*}
shows *FDERIV* $(\lambda x.\ x\ \hat{}\ Suc\ n)\ x$:> $(\lambda h.\ (1 + of\text{-}nat\ n) * x\ \hat{}\ n * h)$
⟨*proof*⟩

lemma *FDERIV-power*:
fixes $x :: 'a$::{*real-normed-algebra,comm-ring-1*}
shows *FDERIV* $(\lambda x.\ x\ \hat{}\ n)\ x$:> $(\lambda h.\ of\text{-}nat\ n * x\ \hat{}\ (n - 1) * h)$
⟨*proof*⟩

17.7 Inverse

lemmas *bounded-linear-mult-const* =
mult.bounded-linear-left [*THEN bounded-linear-compose*]

lemmas *bounded-linear-const-mult* =

mult.bounded-linear-right [*THEN bounded-linear-compose*]

lemma *FDERIV-inverse*:
fixes x :: *'a*::*real-normed-div-algebra*
assumes x: $x \neq 0$
shows *FDERIV inverse x* :> $(\lambda h.\ -\ (inverse\ x * h * inverse\ x))$
(**is** *FDERIV ?inv* - :> -)
⟨*proof*⟩

17.8 Alternate definition

lemma *field-fderiv-def*:
fixes x :: *'a*::*real-normed-field* **shows**
FDERIV f x :> $(\lambda h.\ h * D) = (\lambda h.\ (f\ (x + h) - f\ x)\ /\ h)$ −− 0 −−> D
⟨*proof*⟩

end

18 Inner-Product: Inner Product Spaces and the Gradient Derivative

theory *Inner-Product*
imports *Complex-Main FrechetDeriv*
begin

18.1 Real inner product spaces

Temporarily relax type constraints for *open*, *dist*, and *norm*.

⟨*ML*⟩

class *real-inner* = *real-vector* + *sgn-div-norm* + *dist-norm* + *open-dist* +
fixes *inner* :: *'a* ⇒ *'a* ⇒ *real*
assumes *inner-commute*: *inner x y* = *inner y x*
and *inner-add-left*: *inner* $(x + y)\ z$ = *inner x z* + *inner y z*
and *inner-scaleR-left* [*simp*]: *inner* $(scaleR\ r\ x)\ y = r * (inner\ x\ y)$
and *inner-ge-zero* [*simp*]: $0 \leq inner\ x\ x$
and *inner-eq-zero-iff* [*simp*]: *inner x x* = 0 ⟷ $x = 0$
and *norm-eq-sqrt-inner*: *norm x* = *sqrt* (*inner x x*)
begin

lemma *inner-zero-left* [*simp*]: *inner 0 x* = 0
⟨*proof*⟩

lemma *inner-minus-left* [*simp*]: *inner* $(-\ x)\ y = -\ inner\ x\ y$
⟨*proof*⟩

lemma *inner-diff-left*: *inner* $(x - y)\ z = inner\ x\ z - inner\ y\ z$

⟨*proof*⟩

Transfer distributivity rules to right argument.

lemma *inner-add-right*: *inner x* (*y* + *z*) = *inner x y* + *inner x z*
⟨*proof*⟩

lemma *inner-scaleR-right* [*simp*]: *inner x* (*scaleR r y*) = *r* ∗ (*inner x y*)
⟨*proof*⟩

lemma *inner-zero-right* [*simp*]: *inner x* 0 = 0
⟨*proof*⟩

lemma *inner-minus-right* [*simp*]: *inner x* (− *y*) = − *inner x y*
⟨*proof*⟩

lemma *inner-diff-right*: *inner x* (*y* − *z*) = *inner x y* − *inner x z*
⟨*proof*⟩

lemmas *inner-add* [*algebra-simps*] = *inner-add-left inner-add-right*
lemmas *inner-diff* [*algebra-simps*] = *inner-diff-left inner-diff-right*
lemmas *inner-scaleR* = *inner-scaleR-left inner-scaleR-right*

Legacy theorem names

lemmas *inner-left-distrib* = *inner-add-left*
lemmas *inner-right-distrib* = *inner-add-right*
lemmas *inner-distrib* = *inner-left-distrib inner-right-distrib*

lemma *inner-gt-zero-iff* [*simp*]: 0 < *inner x x* ⟷ *x* ≠ 0
⟨*proof*⟩

lemma *power2-norm-eq-inner*: (*norm x*)^2 = *inner x x*
⟨*proof*⟩

lemma *Cauchy-Schwarz-ineq*:
(*inner x y*)^2 ≤ *inner x x* ∗ *inner y y*
⟨*proof*⟩

lemma *Cauchy-Schwarz-ineq2*:
|*inner x y*| ≤ *norm x* ∗ *norm y*
⟨*proof*⟩

subclass *real-normed-vector*
⟨*proof*⟩

end

Re-enable constraints for *open*, *dist*, and *norm*.

⟨*ML*⟩

interpretation *inner*:
bounded-bilinear inner::'*a*::*real-inner* ⇒ '*a* ⇒ *real*
⟨*proof*⟩

interpretation *inner-left*:
bounded-linear λ*x*::'*a*::*real-inner*. *inner x y*
⟨*proof*⟩

interpretation *inner-right*:
bounded-linear λ*y*::'*a*::*real-inner*. *inner x y*
⟨*proof*⟩

18.2 Class instances

instantiation *real* :: *real-inner*
begin

definition *inner-real-def* [*simp*]: *inner* = *op* ∗

instance ⟨*proof*⟩

end

instantiation *complex* :: *real-inner*
begin

definition *inner-complex-def*:
inner x y = *Re x* ∗ *Re y* + *Im x* ∗ *Im y*

instance ⟨*proof*⟩

end

18.3 Gradient derivative

definition
gderiv ::
['*a*::*real-inner* ⇒ *real*, '*a*, '*a*] ⇒ *bool*
((*GDERIV* (-)/ (-)/ :> (-)) [*1000*, *1000*, *60*] *60*)
where
GDERIV f x :> *D* ⟷ *FDERIV f x* :> (λ*h*. *inner h D*)

lemma *deriv-fderiv*: *DERIV f x* :> *D* ⟷ *FDERIV f x* :> (λ*h*. *h* ∗ *D*)
⟨*proof*⟩

lemma *gderiv-deriv* [*simp*]: *GDERIV f x* :> *D* ⟷ *DERIV f x* :> *D*
⟨*proof*⟩

lemma *GDERIV-DERIV-compose*:
⟦*GDERIV f x* :> *df*; *DERIV g* (*f x*) :> *dg*⟧

$\Longrightarrow$ *GDERIV* $(\lambda x.\ g\ (f\ x))\ x :> scaleR\ dg\ df$
⟨*proof*⟩

lemma *FDERIV-subst*: $[\![FDERIV\ f\ x :> df;\ df = d]\!] \Longrightarrow FDERIV\ f\ x :> d$
⟨*proof*⟩

lemma *GDERIV-subst*: $[\![GDERIV\ f\ x :> df;\ df = d]\!] \Longrightarrow GDERIV\ f\ x :> d$
⟨*proof*⟩

lemma *GDERIV-const*: $GDERIV\ (\lambda x.\ k)\ x :> 0$
⟨*proof*⟩

lemma *GDERIV-add*:
$[\![GDERIV\ f\ x :> df;\ GDERIV\ g\ x :> dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ f\ x + g\ x)\ x :> df + dg$
⟨*proof*⟩

lemma *GDERIV-minus*:
$GDERIV\ f\ x :> df \Longrightarrow GDERIV\ (\lambda x.\ -\ f\ x)\ x :> -\ df$
⟨*proof*⟩

lemma *GDERIV-diff*:
$[\![GDERIV\ f\ x :> df;\ GDERIV\ g\ x :> dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ f\ x - g\ x)\ x :> df - dg$
⟨*proof*⟩

lemma *GDERIV-scaleR*:
$[\![DERIV\ f\ x :> df;\ GDERIV\ g\ x :> dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ scaleR\ (f\ x)\ (g\ x))\ x$
$:> (scaleR\ (f\ x)\ dg + scaleR\ df\ (g\ x))$
⟨*proof*⟩

lemma *GDERIV-mult*:
$[\![GDERIV\ f\ x :> df;\ GDERIV\ g\ x :> dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ f\ x * g\ x)\ x :> scaleR\ (f\ x)\ dg + scaleR\ (g\ x)\ df$
⟨*proof*⟩

lemma *GDERIV-inverse*:
$[\![GDERIV\ f\ x :> df;\ f\ x \neq 0]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ inverse\ (f\ x))\ x :> -\ (inverse\ (f\ x))^2 *_R df$
⟨*proof*⟩

lemma *GDERIV-norm*:
assumes $x \neq 0$ **shows** $GDERIV\ (\lambda x.\ norm\ x)\ x :> sgn\ x$
⟨*proof*⟩

lemmas *FDERIV-norm* = *GDERIV-norm* [*unfolded gderiv-def*]

end

19 Product-plus: Additive group operations on product types

theory *Product-plus*
imports *Main*
begin

19.1 Operations

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

definition *zero-prod-def*: $0 = (0, 0)$

instance ⟨*proof*⟩
end

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

definition *plus-prod-def*:
$x + y = (fst\ x + fst\ y, snd\ x + snd\ y)$

instance ⟨*proof*⟩
end

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

definition *minus-prod-def*:
$x - y = (fst\ x - fst\ y, snd\ x - snd\ y)$

instance ⟨*proof*⟩
end

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

definition *uminus-prod-def*:
$- x = (- fst\ x, - snd\ x)$

instance ⟨*proof*⟩
end

lemma *fst-zero* [*simp*]: $fst\ 0 = 0$
⟨*proof*⟩

lemma *snd-zero* [*simp*]: *snd 0 = 0*
⟨*proof*⟩

lemma *fst-add* [*simp*]: *fst* $(x + y) =$ *fst* $x +$ *fst* y
⟨*proof*⟩

lemma *snd-add* [*simp*]: *snd* $(x + y) =$ *snd* $x +$ *snd* y
⟨*proof*⟩

lemma *fst-diff* [*simp*]: *fst* $(x - y) =$ *fst* $x -$ *fst* y
⟨*proof*⟩

lemma *snd-diff* [*simp*]: *snd* $(x - y) =$ *snd* $x -$ *snd* y
⟨*proof*⟩

lemma *fst-uminus* [*simp*]: *fst* $(- x) = -$ *fst* x
⟨*proof*⟩

lemma *snd-uminus* [*simp*]: *snd* $(- x) = -$ *snd* x
⟨*proof*⟩

lemma *add-Pair* [*simp*]: $(a, b) + (c, d) = (a + c, b + d)$
⟨*proof*⟩

lemma *diff-Pair* [*simp*]: $(a, b) - (c, d) = (a - c, b - d)$
⟨*proof*⟩

lemma *uminus-Pair* [*simp*, *code*]: $- (a, b) = (- a, - b)$
⟨*proof*⟩

lemmas *expand-prod-eq* = *Pair-fst-snd-eq*

19.2 Class instances

instance $*$ :: (*semigroup-add*, *semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance $*$ :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance $*$ :: (*monoid-add*, *monoid-add*) *monoid-add*
⟨*proof*⟩

instance $*$ :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance $*$ ::
(*cancel-semigroup-add*, *cancel-semigroup-add*) *cancel-semigroup-add*

⟨*proof*⟩

instance ∗ ::
(*cancel-ab-semigroup-add*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance ∗ ::
(*cancel-comm-monoid-add*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
⟨*proof*⟩

instance ∗ :: (*group-add*, *group-add*) *group-add*
⟨*proof*⟩

instance ∗ :: (*ab-group-add*, *ab-group-add*) *ab-group-add*
⟨*proof*⟩

lemma *fst-setsum*: *fst* (∑ *x*∈*A*. *f x*) = (∑ *x*∈*A*. *fst* (*f x*))
⟨*proof*⟩

lemma *snd-setsum*: *snd* (∑ *x*∈*A*. *f x*) = (∑ *x*∈*A*. *snd* (*f x*))
⟨*proof*⟩

end

20 Product-Vector: Cartesian Products as Vector Spaces

theory *Product-Vector*
imports *Inner-Product Product-plus*
begin

20.1 Product is a real vector space

instantiation ∗ :: (*real-vector*, *real-vector*) *real-vector*
begin

definition *scaleR-prod-def*:
scaleR r A = (*scaleR r* (*fst A*), *scaleR r* (*snd A*))

lemma *fst-scaleR* [*simp*]: *fst* (*scaleR r A*) = *scaleR r* (*fst A*)
⟨*proof*⟩

lemma *snd-scaleR* [*simp*]: *snd* (*scaleR r A*) = *scaleR r* (*snd A*)
⟨*proof*⟩

lemma *scaleR-Pair* [*simp*]: *scaleR r* (*a*, *b*) = (*scaleR r a*, *scaleR r b*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

20.2 Product is a topological space

instantiation
∗ :: (*topological-space*, *topological-space*) *topological-space*
begin

definition *open-prod-def*:
open (*S* :: ('*a* × '*b*) *set*) ⟷
(∀ *x*∈*S*. ∃ *A B*. *open A* ∧ *open B* ∧ *x* ∈ *A* × *B* ∧ *A* × *B* ⊆ *S*)

lemma *open-prod-elim*:
assumes *open S* **and** *x* ∈ *S*
obtains *A B* **where** *open A* **and** *open B* **and** *x* ∈ *A* × *B* **and** *A* × *B* ⊆ *S*
⟨*proof*⟩

lemma *open-prod-intro*:
assumes ⋀*x*. *x* ∈ *S* ⟹ ∃ *A B*. *open A* ∧ *open B* ∧ *x* ∈ *A* × *B* ∧ *A* × *B* ⊆ *S*
shows *open S*
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *open-Times*: *open S* ⟹ *open T* ⟹ *open* (*S* × *T*)
⟨*proof*⟩

lemma *fst-vimage-eq-Times*: *fst* −‘ *S* = *S* × *UNIV*
⟨*proof*⟩

lemma *snd-vimage-eq-Times*: *snd* −‘ *S* = *UNIV* × *S*
⟨*proof*⟩

lemma *open-vimage-fst*: *open S* ⟹ *open* (*fst* −‘ *S*)
⟨*proof*⟩

lemma *open-vimage-snd*: *open S* ⟹ *open* (*snd* −‘ *S*)
⟨*proof*⟩

lemma *closed-vimage-fst*: *closed S* ⟹ *closed* (*fst* −‘ *S*)
⟨*proof*⟩

lemma *closed-vimage-snd*: *closed S* ⟹ *closed* (*snd* −‘ *S*)
⟨*proof*⟩

lemma *closed-Times*: *closed* $S \Longrightarrow$ *closed* $T \Longrightarrow$ *closed* $(S \times T)$
⟨*proof*⟩

lemma *openI*:
assumes $\bigwedge x.\ x \in S \Longrightarrow \exists\, T.\ open\ T \wedge x \in T \wedge T \subseteq S$
shows *open* S
⟨*proof*⟩

lemma *subset-fst-imageI*: $A \times B \subseteq S \Longrightarrow y \in B \Longrightarrow A \subseteq fst\ `\ S$
⟨*proof*⟩

lemma *subset-snd-imageI*: $A \times B \subseteq S \Longrightarrow x \in A \Longrightarrow B \subseteq snd\ `\ S$
⟨*proof*⟩

lemma *open-image-fst*: **assumes** *open* S **shows** *open* $(fst\ `\ S)$
⟨*proof*⟩

lemma *open-image-snd*: **assumes** *open* S **shows** *open* $(snd\ `\ S)$
⟨*proof*⟩

20.3 Product is a metric space

instantiation
$*$ :: (*metric-space*, *metric-space*) *metric-space*
begin

definition *dist-prod-def*:
$dist\ (x::'a \times 'b)\ y = sqrt\ ((dist\ (fst\ x)\ (fst\ y))^2 + (dist\ (snd\ x)\ (snd\ y))^2)$

lemma *dist-Pair-Pair*: $dist\ (a,\ b)\ (c,\ d) = sqrt\ ((dist\ a\ c)^2 + (dist\ b\ d)^2)$
⟨*proof*⟩

lemma *dist-fst-le*: $dist\ (fst\ x)\ (fst\ y) \leq dist\ x\ y$
⟨*proof*⟩

lemma *dist-snd-le*: $dist\ (snd\ x)\ (snd\ y) \leq dist\ x\ y$
⟨*proof*⟩

instance ⟨*proof*⟩

end

20.4 Continuity of operations

lemma *tendsto-fst* [*tendsto-intros*]:
assumes $(f \longrightarrow a)\ net$
shows $((\lambda x.\ fst\ (f\ x)) \longrightarrow fst\ a)\ net$
⟨*proof*⟩

lemma *tendsto-snd* [*tendsto-intros*]:
assumes $(f \longrightarrow a)$ *net*
shows $((\lambda x.\ snd\ (f\ x)) \longrightarrow snd\ a)$ *net*
⟨*proof*⟩

lemma *tendsto-Pair* [*tendsto-intros*]:
assumes $(f \longrightarrow a)$ *net* **and** $(g \longrightarrow b)$ *net*
shows $((\lambda x.\ (f\ x,\ g\ x)) \longrightarrow (a,\ b))$ *net*
⟨*proof*⟩

lemma *Cauchy-fst*: $Cauchy\ X \Longrightarrow Cauchy\ (\lambda n.\ fst\ (X\ n))$
⟨*proof*⟩

lemma *Cauchy-snd*: $Cauchy\ X \Longrightarrow Cauchy\ (\lambda n.\ snd\ (X\ n))$
⟨*proof*⟩

lemma *Cauchy-Pair*:
assumes *Cauchy X* **and** *Cauchy Y*
shows $Cauchy\ (\lambda n.\ (X\ n,\ Y\ n))$
⟨*proof*⟩

lemma *isCont-Pair* [*simp*]:
$[\![isCont\ f\ x;\ isCont\ g\ x]\!] \Longrightarrow isCont\ (\lambda x.\ (f\ x,\ g\ x))\ x$
⟨*proof*⟩

20.5 Product is a complete metric space

instance $*$ :: (*complete-space*, *complete-space*) *complete-space*
⟨*proof*⟩

20.6 Product is a normed vector space

instantiation
$*$ :: (*real-normed-vector*, *real-normed-vector*) *real-normed-vector*
begin

definition *norm-prod-def*:
$norm\ x = sqrt\ ((norm\ (fst\ x))^2 + (norm\ (snd\ x))^2)$

definition *sgn-prod-def*:
$sgn\ (x::'a \times 'b) = scaleR\ (inverse\ (norm\ x))\ x$

lemma *norm-Pair*: $norm\ (a,\ b) = sqrt\ ((norm\ a)^2 + (norm\ b)^2)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance $*$ :: (*banach*, *banach*) *banach* ⟨*proof*⟩

20.7 Product is an inner product space

instantiation $*$ :: (*real-inner*, *real-inner*) *real-inner*
begin

definition *inner-prod-def*:
inner x y = *inner* (*fst x*) (*fst y*) + *inner* (*snd x*) (*snd y*)

lemma *inner-Pair* [*simp*]: *inner* (*a*, *b*) (*c*, *d*) = *inner a c* + *inner b d*
⟨*proof*⟩

instance ⟨*proof*⟩

end

20.8 Pair operations are linear

interpretation *fst*: *bounded-linear fst*
⟨*proof*⟩

interpretation *snd*: *bounded-linear snd*
⟨*proof*⟩

TODO: move to NthRoot

lemma *sqrt-add-le-add-sqrt*:
assumes *x*: $0 \leq x$ **and** *y*: $0 \leq y$
shows *sqrt* $(x + y) \leq$ *sqrt x* + *sqrt y*
⟨*proof*⟩

lemma *bounded-linear-Pair*:
assumes *f*: *bounded-linear f*
assumes *g*: *bounded-linear g*
shows *bounded-linear* $(\lambda x.\ (f\ x,\ g\ x))$
⟨*proof*⟩

20.9 Frechet derivatives involving pairs

lemma *FDERIV-Pair*:
assumes *f*: *FDERIV f x* :> *f'* **and** *g*: *FDERIV g x* :> *g'*
shows *FDERIV* $(\lambda x.\ (f\ x,\ g\ x))\ x :> (\lambda h.\ (f'\ h,\ g'\ h))$
⟨*proof*⟩

end

21 Convex: Convexity in real vector spaces

theory *Convex*
imports *Product-Vector*
begin

21.1 Convexity.

definition
convex :: *'a::real-vector set* ⇒ *bool* **where**
convex s ⟷ (∀ *x*∈*s*. ∀ *y*∈*s*. ∀ *u*≥*0*. ∀ *v*≥*0*. *u* + *v* = *1* ⟶ *u* *$_R$ *x* + *v* *$_R$ *y* ∈ *s*)

lemma *convex-alt*:
convex s ⟷ (∀ *x*∈*s*. ∀ *y*∈*s*. ∀ *u*. *0* ≤ *u* ∧ *u* ≤ *1* ⟶ ((*1* − *u*) *$_R$ *x* + *u* *$_R$ *y*) ∈ *s*)
(**is** - ⟷ *?alt*)
⟨*proof*⟩

lemma *mem-convex*:
assumes *convex s a* ∈ *s b* ∈ *s* *0* ≤ *u u* ≤ *1*
shows ((*1* − *u*) *$_R$ *a* + *u* *$_R$ *b*) ∈ *s*
⟨*proof*⟩

lemma *convex-empty*[*intro*]: *convex* {}
⟨*proof*⟩

lemma *convex-singleton*[*intro*]: *convex* {*a*}
⟨*proof*⟩

lemma *convex-UNIV*[*intro*]: *convex UNIV*
⟨*proof*⟩

lemma *convex-Inter*: (∀ *s*∈*f*. *convex s*) ==> *convex*(⋂ *f*)
⟨*proof*⟩

lemma *convex-Int*: *convex s* ⟹ *convex t* ⟹ *convex* (*s* ∩ *t*)
⟨*proof*⟩

lemma *convex-halfspace-le*: *convex* {*x*. *inner a x* ≤ *b*}
⟨*proof*⟩

lemma *convex-halfspace-ge*: *convex* {*x*. *inner a x* ≥ *b*}
⟨*proof*⟩

lemma *convex-hyperplane*: *convex* {*x*. *inner a x* = *b*}
⟨*proof*⟩

lemma *convex-halfspace-lt*: *convex* {*x*. *inner a x* < *b*}
⟨*proof*⟩

lemma *convex-halfspace-gt*: *convex* {*x*. *inner a x* > *b*}
⟨*proof*⟩

lemma *convex-real-interval*:
fixes *a b* :: *real*

shows *convex* $\{a..\}$ **and** *convex* $\{..b\}$
and *convex* $\{a<..\}$ **and** *convex* $\{..<b\}$
and *convex* $\{a..b\}$ **and** *convex* $\{a<..b\}$
and *convex* $\{a..<b\}$ **and** *convex* $\{a<..<b\}$
⟨*proof*⟩

21.2 Explicit expressions for convexity in terms of arbitrary sums.

lemma *convex-setsum*:
fixes C :: $'a$::*real-vector set*
assumes *finite* s **and** *convex* C **and** $(\sum\ i \in s.\ a\ i) = 1$
assumes $\bigwedge i.\ i \in s \Longrightarrow a\ i \geq 0$ **and** $\bigwedge i.\ i \in s \Longrightarrow y\ i \in C$
shows $(\sum\ j \in s.\ a\ j *_R y\ j) \in C$
⟨*proof*⟩

lemma *convex*:
shows *convex* $s \longleftrightarrow (\forall (k::nat)\ u\ x.\ (\forall i.\ 1 \leq i \wedge i \leq k \longrightarrow 0 \leq u\ i \wedge x\ i \in s) \wedge$
$(setsum\ u\ \{1..k\} = 1)$
$\longrightarrow setsum\ (\lambda i.\ u\ i *_R x\ i)\ \{1..k\} \in s)$
⟨*proof*⟩

lemma *convex-explicit*:
fixes s :: $'a$::*real-vector set*
shows *convex* $s \longleftrightarrow$
$(\forall t\ u.\ finite\ t \wedge t \subseteq s \wedge (\forall x \in t.\ 0 \leq u\ x) \wedge setsum\ u\ t = 1 \longrightarrow setsum\ (\lambda x.\ u$
$x *_R x)\ t \in s)$
⟨*proof*⟩

lemma *convex-finite*: **assumes** *finite* s
shows *convex* $s \longleftrightarrow (\forall u.\ (\forall x \in s.\ 0 \leq u\ x) \wedge setsum\ u\ s = 1$
$\longrightarrow setsum\ (\lambda x.\ u\ x *_R x)\ s \in s)$
⟨*proof*⟩

definition
convex-on :: $'a$::*real-vector set* ⇒ ($'a$ ⇒ *real*) ⇒ *bool* **where**
convex-on $s\ f \longleftrightarrow$
$(\forall x \in s.\ \forall y \in s.\ \forall u \geq 0.\ \forall v \geq 0.\ u + v = 1 \longrightarrow f\ (u *_R x + v *_R y) \leq u * f\ x +$
$v * f\ y)$

lemma *convex-on-subset*: *convex-on* $t\ f \Longrightarrow s \subseteq t \Longrightarrow$ *convex-on* $s\ f$
⟨*proof*⟩

lemma *convex-add*[*intro*]:
assumes *convex-on* $s\ f$ *convex-on* $s\ g$
shows *convex-on* $s\ (\lambda x.\ f\ x + g\ x)$
⟨*proof*⟩

lemma *convex-cmul*[*intro*]:
assumes $0 \leq (c::real)$ *convex-on* $s\ f$
shows *convex-on* $s\ (\lambda x.\ c * f\ x)$
⟨*proof*⟩

lemma *convex-lower*:
assumes *convex-on* $s\ f\ \ x{\in}s\ \ y \in s\ \ 0 \leq u\ \ 0 \leq v\ \ u + v = 1$
shows $f\ (u *_R x + v *_R y) \leq max\ (f\ x)\ (f\ y)$
⟨*proof*⟩

lemma *convex-distance*[*intro*]:
fixes $s :: 'a::real\text{-}normed\text{-}vector\ set$
shows *convex-on* $s\ (\lambda x.\ dist\ a\ x)$
⟨*proof*⟩

21.3 Arithmetic operations on sets preserve convexity.

lemma *convex-scaling*:
assumes *convex* s
shows*convex* $((\lambda x.\ c *_R x)\ `\ s)$
⟨*proof*⟩

lemma *convex-negations*: *convex* $s \Longrightarrow$ *convex* $((\lambda x.\ -x)`\ s)$
⟨*proof*⟩

lemma *convex-sums*:
assumes *convex* s *convex* t
shows *convex* $\{x + y |\ x\ y.\ x \in s \wedge y \in t\}$
⟨*proof*⟩

lemma *convex-differences*:
assumes *convex* s *convex* t
shows *convex* $\{x - y |\ x\ y.\ x \in s \wedge y \in t\}$
⟨*proof*⟩

lemma *convex-translation*: **assumes** *convex* s **shows** *convex* $((\lambda x.\ a + x)\ `\ s)$
⟨*proof*⟩

lemma *convex-affinity*: **assumes** *convex* s **shows** *convex* $((\lambda x.\ a + c *_R x)\ `\ s)$
⟨*proof*⟩

lemma *convex-linear-image*:
assumes c:*convex* s **and** l:*bounded-linear* f
shows *convex*$(f\ `\ s)$
⟨*proof*⟩

lemma *pos-is-convex*:
shows *convex* $\{0 :: real <..\}$

⟨*proof*⟩

lemma *convex-on-setsum*:
fixes $a :: 'a \Rightarrow real$
fixes $y :: 'a \Rightarrow 'b$::*real-vector*
fixes $f :: 'b \Rightarrow real$
assumes *finite* s $s \neq \{\}$
assumes *convex-on* C f
assumes *convex* C
assumes $(\sum\ i \in s.\ a\ i) = 1$
assumes $\bigwedge i.\ i \in s \Longrightarrow a\ i \geq 0$
assumes $\bigwedge i.\ i \in s \Longrightarrow y\ i \in C$
shows $f\ (\sum\ i \in s.\ a\ i *_R y\ i) \leq (\sum\ i \in s.\ a\ i * f\ (y\ i))$
⟨*proof*⟩

lemma *convex-on-alt*:
fixes $C :: 'a$::*real-vector set*
assumes *convex* C
shows *convex-on* C f =
$(\forall\ x \in C.\ \forall\ y \in C.\ \forall\ \mu :: real.\ \mu \geq 0 \wedge \mu \leq 1$
$\longrightarrow f\ (\mu *_R x + (1 - \mu) *_R y) \leq \mu * f\ x + (1 - \mu) * f\ y)$
⟨*proof*⟩

lemma *pos-convex-function*:
fixes $f :: real \Rightarrow real$
assumes *convex* C
assumes *leq*: $\bigwedge x\ y.\ [\![x \in C\ ;\ y \in C]\!] \Longrightarrow f'\ x * (y - x) \leq f\ y - f\ x$
shows *convex-on* C f
⟨*proof*⟩

lemma *atMostAtLeast-subset-convex*:
fixes $C :: real\ set$
assumes *convex* C
assumes $x \in C\ y \in C\ x < y$
shows $\{x\ ..\ y\} \subseteq C$
⟨*proof*⟩

lemma *f''-imp-f'*:
fixes $f :: real \Rightarrow real$
assumes *convex* C
assumes f': $\bigwedge x.\ x \in C \Longrightarrow DERIV\ f\ x :> (f'\ x)$
assumes f'': $\bigwedge x.\ x \in C \Longrightarrow DERIV\ f'\ x :> (f''\ x)$
assumes *pos*: $\bigwedge x.\ x \in C \Longrightarrow f''\ x \geq 0$
assumes $x \in C\ y \in C$
shows $f'\ x * (y - x) \leq f\ y - f\ x$
⟨*proof*⟩

lemma *f''-ge0-imp-convex*:

fixes $f :: real \Rightarrow real$
assumes *conv*: *convex C*
assumes f': $\bigwedge x.\ x \in C \Longrightarrow DERIV\ f\ x :> (f'\ x)$
assumes f'': $\bigwedge x.\ x \in C \Longrightarrow DERIV\ f'\ x :> (f''\ x)$
assumes *pos*: $\bigwedge x.\ x \in C \Longrightarrow f''\ x \geq 0$
shows *convex-on C f*
⟨*proof*⟩

lemma *minus-log-convex*:
fixes $b :: real$
assumes $b > 1$
shows *convex-on* $\{0 <..\}$ $(\lambda\ x.\ -\ log\ b\ x)$
⟨*proof*⟩

end

22 Nat-Bijection: Bijections between natural numbers and other types

theory *Nat-Bijection*
imports *Main Parity*
begin

22.1 Type $nat \times nat$

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

definition
triangle :: $nat \Rightarrow nat$
where
$triangle\ n = n * Suc\ n\ div\ 2$

lemma *triangle-0* [*simp*]: $triangle\ 0 = 0$
⟨*proof*⟩

lemma *triangle-Suc* [*simp*]: $triangle\ (Suc\ n) = triangle\ n + Suc\ n$
⟨*proof*⟩

definition
prod-encode :: $nat \times nat \Rightarrow nat$
where
$prod\text{-}encode = (\lambda(m,\ n).\ triangle\ (m + n) + m)$

In this auxiliary function, $triangle\ k + m$ is an invariant.

fun
prod-decode-aux :: $nat \Rightarrow nat \Rightarrow nat \times nat$
where
$prod\text{-}decode\text{-}aux\ k\ m =$

(*if* $m \leq k$ *then* $(m, k - m)$ *else prod-decode-aux* (*Suc* k) $(m - Suc\ k)$)

declare *prod-decode-aux.simps* [*simp del*]

definition
prod-decode :: $nat \Rightarrow nat \times nat$
where
prod-decode = *prod-decode-aux 0*

lemma *prod-encode-prod-decode-aux*:
prod-encode (*prod-decode-aux k m*) = *triangle k* + *m*
⟨*proof*⟩

lemma *prod-decode-inverse* [*simp*]: *prod-encode* (*prod-decode n*) = *n*
⟨*proof*⟩

lemma *prod-decode-triangle-add*:
prod-decode (*triangle k* + *m*) = *prod-decode-aux k m*
⟨*proof*⟩

lemma *prod-encode-inverse* [*simp*]: *prod-decode* (*prod-encode x*) = *x*
⟨*proof*⟩

lemma *inj-prod-encode*: *inj-on prod-encode A*
⟨*proof*⟩

lemma *inj-prod-decode*: *inj-on prod-decode A*
⟨*proof*⟩

lemma *surj-prod-encode*: *surj prod-encode*
⟨*proof*⟩

lemma *surj-prod-decode*: *surj prod-decode*
⟨*proof*⟩

lemma *bij-prod-encode*: *bij prod-encode*
⟨*proof*⟩

lemma *bij-prod-decode*: *bij prod-decode*
⟨*proof*⟩

lemma *prod-encode-eq*: *prod-encode x* = *prod-encode y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *prod-decode-eq*: *prod-decode x* = *prod-decode y* ⟷ *x* = *y*
⟨*proof*⟩

Ordering properties

lemma *le-prod-encode-1*: *a* ≤ *prod-encode* (*a*, *b*)

⟨*proof*⟩

lemma *le-prod-encode-2*: $b \leq$ *prod-encode* (a, b)
⟨*proof*⟩

22.2 Type *nat* + *nat*

definition
sum-encode :: $nat + nat \Rightarrow nat$
where
sum-encode $x =$ (*case* x *of Inl* $a \Rightarrow 2 * a \mid Inr\ b \Rightarrow Suc\ (2 * b)$)

definition
sum-decode :: $nat \Rightarrow nat + nat$
where
sum-decode $n =$ (*if even* n *then Inl* $(n\ div\ 2)$ *else Inr* $(n\ div\ 2)$)

lemma *sum-encode-inverse* [*simp*]: *sum-decode* (*sum-encode* x) $= x$
⟨*proof*⟩

lemma *sum-decode-inverse* [*simp*]: *sum-encode* (*sum-decode* n) $= n$
⟨*proof*⟩

lemma *inj-sum-encode*: *inj-on sum-encode A*
⟨*proof*⟩

lemma *inj-sum-decode*: *inj-on sum-decode A*
⟨*proof*⟩

lemma *surj-sum-encode*: *surj sum-encode*
⟨*proof*⟩

lemma *surj-sum-decode*: *surj sum-decode*
⟨*proof*⟩

lemma *bij-sum-encode*: *bij sum-encode*
⟨*proof*⟩

lemma *bij-sum-decode*: *bij sum-decode*
⟨*proof*⟩

lemma *sum-encode-eq*: *sum-encode* $x =$ *sum-encode* $y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *sum-decode-eq*: *sum-decode* $x =$ *sum-decode* $y \longleftrightarrow x = y$
⟨*proof*⟩

22.3 Type *int*

definition

int-encode :: *int* ⇒ *nat*
where
int-encode i = *sum-encode* (*if* 0 ≤ *i then Inl* (*nat i*) *else Inr* (*nat* (− *i* − 1)))

definition
int-decode :: *nat* ⇒ *int*
where
int-decode n = (*case sum-decode n of Inl a* ⇒ *int a* | *Inr b* ⇒ − *int b* − 1)

lemma *int-encode-inverse* [*simp*]: *int-decode* (*int-encode x*) = *x*
⟨*proof*⟩

lemma *int-decode-inverse* [*simp*]: *int-encode* (*int-decode n*) = *n*
⟨*proof*⟩

lemma *inj-int-encode*: *inj-on int-encode A*
⟨*proof*⟩

lemma *inj-int-decode*: *inj-on int-decode A*
⟨*proof*⟩

lemma *surj-int-encode*: *surj int-encode*
⟨*proof*⟩

lemma *surj-int-decode*: *surj int-decode*
⟨*proof*⟩

lemma *bij-int-encode*: *bij int-encode*
⟨*proof*⟩

lemma *bij-int-decode*: *bij int-decode*
⟨*proof*⟩

lemma *int-encode-eq*: *int-encode x* = *int-encode y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *int-decode-eq*: *int-decode x* = *int-decode y* ⟷ *x* = *y*
⟨*proof*⟩

22.4 Type *nat list*

fun
list-encode :: *nat list* ⇒ *nat*
where
list-encode [] = 0
| *list-encode* (*x* # *xs*) = *Suc* (*prod-encode* (*x*, *list-encode xs*))

function
list-decode :: *nat* ⇒ *nat list*

where
list-decode 0 = []
| *list-decode (Suc n) = (case prod-decode n of (x, y) ⇒ x # list-decode y)*
⟨*proof*⟩

termination *list-decode*
⟨*proof*⟩

lemma *list-encode-inverse* [*simp*]: *list-decode (list-encode x) = x*
⟨*proof*⟩

lemma *list-decode-inverse* [*simp*]: *list-encode (list-decode n) = n*
⟨*proof*⟩

lemma *inj-list-encode*: *inj-on list-encode A*
⟨*proof*⟩

lemma *inj-list-decode*: *inj-on list-decode A*
⟨*proof*⟩

lemma *surj-list-encode*: *surj list-encode*
⟨*proof*⟩

lemma *surj-list-decode*: *surj list-decode*
⟨*proof*⟩

lemma *bij-list-encode*: *bij list-encode*
⟨*proof*⟩

lemma *bij-list-decode*: *bij list-decode*
⟨*proof*⟩

lemma *list-encode-eq*: *list-encode x = list-encode y ⟷ x = y*
⟨*proof*⟩

lemma *list-decode-eq*: *list-decode x = list-decode y ⟷ x = y*
⟨*proof*⟩

22.5 Finite sets of naturals

22.5.1 Preliminaries

lemma *finite-vimage-Suc-iff*: *finite (Suc −‘ F) ⟷ finite F*
⟨*proof*⟩

lemma *vimage-Suc-insert-0*: *Suc −‘ insert 0 A = Suc −‘ A*
⟨*proof*⟩

lemma *vimage-Suc-insert-Suc*:
Suc −‘ insert (Suc n) A = insert n (Suc −‘ A)

⟨*proof*⟩

lemma *even-nat-Suc-div-2*: *even x* ⟹ *Suc x div 2* = *x div 2*
⟨*proof*⟩

lemma *div2-even-ext-nat*:
⟦*x div 2* = *y div 2*; *even x* = *even y*⟧ ⟹ (*x*::*nat*) = *y*
⟨*proof*⟩

22.5.2 From sets to naturals

definition
set-encode :: *nat set* ⇒ *nat*
where
set-encode = *setsum* (*op* ^ *2*)

lemma *set-encode-empty* [*simp*]: *set-encode* {} = *0*
⟨*proof*⟩

lemma *set-encode-insert* [*simp*]:
⟦*finite A*; *n* ∉ *A*⟧ ⟹ *set-encode* (*insert n A*) = *2*^*n* + *set-encode A*
⟨*proof*⟩

lemma *even-set-encode-iff*: *finite A* ⟹ *even* (*set-encode A*) ⟷ *0* ∉ *A*
⟨*proof*⟩

lemma *set-encode-vimage-Suc*: *set-encode* (*Suc* −‘ *A*) = *set-encode A div 2*
⟨*proof*⟩

lemmas *set-encode-div-2* = *set-encode-vimage-Suc* [*symmetric*]

22.5.3 From naturals to sets

definition
set-decode :: *nat* ⇒ *nat set*
where
set-decode x = {*n*. *odd* (*x div 2* ^ *n*)}

lemma *set-decode-0* [*simp*]: *0* ∈ *set-decode x* ⟷ *odd x*
⟨*proof*⟩

lemma *set-decode-Suc* [*simp*]:
Suc n ∈ *set-decode x* ⟷ *n* ∈ *set-decode* (*x div 2*)
⟨*proof*⟩

lemma *set-decode-zero* [*simp*]: *set-decode 0* = {}
⟨*proof*⟩

lemma *set-decode-div-2*: *set-decode* (*x div 2*) = *Suc* −‘ *set-decode x*
⟨*proof*⟩

lemma *set-decode-plus-power-2*:
$n \notin$ *set-decode* $z \Longrightarrow$ *set-decode* $(2 \hat{} n + z) =$ *insert* n (*set-decode* z)
⟨*proof*⟩

lemma *finite-set-decode* [*simp*]: *finite* (*set-decode* n)
⟨*proof*⟩

22.5.4 Proof of isomorphism

lemma *set-decode-inverse* [*simp*]: *set-encode* (*set-decode* n) $= n$
⟨*proof*⟩

lemma *set-encode-inverse* [*simp*]: *finite* $A \Longrightarrow$ *set-decode* (*set-encode* A) $= A$
⟨*proof*⟩

lemma *inj-on-set-encode*: *inj-on set-encode* (*Collect finite*)
⟨*proof*⟩

lemma *set-encode-eq*:
⟦*finite* A; *finite* B⟧ $\Longrightarrow$ *set-encode* $A =$ *set-encode* $B \longleftrightarrow A = B$
⟨*proof*⟩

end

23 Countable: Encoding (almost) everything into natural numbers

theory *Countable*
imports *Main Rat Nat-Bijection*
begin

23.1 The class of countable types

class *countable* =
assumes *ex-inj*: $\exists$ *to-nat* :: $'a \Rightarrow$ *nat*. *inj to-nat*

lemma *countable-classI*:
fixes f :: $'a \Rightarrow$ *nat*
assumes $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
shows *OFCLASS*($'a$, *countable-class*)
⟨*proof*⟩

23.2 Conversion functions

definition *to-nat* :: $'a$::*countable* $\Rightarrow$ *nat* **where**
to-nat $=$ (*SOME* f. *inj* f)

definition *from-nat* :: *nat* ⇒ *'a*::*countable* **where**
from-nat = *inv* (*to-nat* :: *'a* ⇒ *nat*)

lemma *inj-to-nat* [*simp*]: *inj to-nat*
⟨*proof*⟩

lemma *surj-from-nat* [*simp*]: *surj from-nat*
⟨*proof*⟩

lemma *to-nat-split* [*simp*]: *to-nat x* = *to-nat y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *from-nat-to-nat* [*simp*]:
from-nat (*to-nat x*) = *x*
⟨*proof*⟩

23.3 Countable types

instance *nat* :: *countable*
⟨*proof*⟩

subclass (**in** *finite*) *countable*
⟨*proof*⟩

Pairs

instance ∗ :: (*countable*, *countable*) *countable*
⟨*proof*⟩

Sums

instance +:: (*countable*, *countable*) *countable*
⟨*proof*⟩

Integers

instance *int* :: *countable*
⟨*proof*⟩

Options

instance *option* :: (*countable*) *countable*
⟨*proof*⟩

Lists

instance *list* :: (*countable*) *countable*
⟨*proof*⟩

Functions

instance *fun* :: (*finite*, *countable*) *countable*
⟨*proof*⟩

23.4 The Rationals are Countably Infinite

definition *nat-to-rat-surj* :: *nat* ⇒ *rat* **where**
nat-to-rat-surj n = (*let* (*a*,*b*) = *prod-decode n*
in Fract (*int-decode a*) (*int-decode b*))

lemma *surj-nat-to-rat-surj*: *surj nat-to-rat-surj*
⟨*proof*⟩

lemma *Rats-eq-range-nat-to-rat-surj*: ℚ = *range nat-to-rat-surj*
⟨*proof*⟩

context *field-char-0*
begin

lemma *Rats-eq-range-of-rat-o-nat-to-rat-surj*:
ℚ = *range* (*of-rat o nat-to-rat-surj*)
⟨*proof*⟩

lemma *surj-of-rat-nat-to-rat-surj*:
r∈ℚ ⟹ ∃ *n*. *r* = *of-rat*(*nat-to-rat-surj n*)
⟨*proof*⟩

end

instance *rat* :: *countable*
⟨*proof*⟩

end

24 Diagonalize: A constructive version of Cantor's first diagonalization argument.

theory *Diagonalize*
imports *Main*
begin

24.1 Summation from *0* to *n*

definition *sum* :: *nat* ⇒ *nat* **where**
sum n = *n* ∗ *Suc n div 2*

lemma *sum-0*:
sum 0 = *0*
⟨*proof*⟩

lemma *sum-Suc*:
sum (*Suc n*) = *Suc n* + *sum n*

⟨*proof*⟩

lemma *sum2*:
*2 * sum n = n * Suc n*
⟨*proof*⟩

lemma *sum-strict-mono*:
strict-mono sum
⟨*proof*⟩

lemma *sum-not-less-self*:
n ≤ sum n
⟨*proof*⟩

lemma *sum-rest-aux*:
assumes *q ≤ n*
assumes *sum m ≤ sum n + q*
shows *m ≤ n*
⟨*proof*⟩

lemma *sum-rest*:
assumes *q ≤ n*
shows *sum m ≤ sum n + q ⟷ m ≤ n*
⟨*proof*⟩

24.2 Diagonalization: an injective embedding of two *nat*s to one *nat*

definition *diagonalize* :: *nat ⇒ nat ⇒ nat* **where**
diagonalize m n = sum (m + n) + m

lemma *diagonalize-inject*:
assumes *diagonalize a b = diagonalize c d*
shows *a = c* **and** *b = d*
⟨*proof*⟩

24.3 The reverse diagonalization: reconstruction a pair of *nat*s from one *nat*

The inverse of the *sum* function

definition *tupelize* :: *nat ⇒ nat × nat* **where**
tupelize q = (let d = Max {d. sum d ≤ q}; m = q − sum d
in (m, d − m))

lemma *tupelize-diagonalize*:
tupelize (diagonalize m n) = (m, n)
⟨*proof*⟩

lemma *snd-tupelize*:

snd (*tupelize n*) ≤ *n*
⟨*proof*⟩

end

25 More-List: Operations on lists beyond the standard List theory

theory *More-List*
imports *Main*
begin

hide-const (**open**) *Finite-Set.fold*

Repairing code generator setup

declare (**in** *lattice*) *Inf-fin-set-fold* [*code-unfold del*]
declare (**in** *lattice*) *Sup-fin-set-fold* [*code-unfold del*]
declare (**in** *linorder*) *Min-fin-set-fold* [*code-unfold del*]
declare (**in** *linorder*) *Max-fin-set-fold* [*code-unfold del*]
declare (**in** *complete-lattice*) *Inf-set-fold* [*code-unfold del*]
declare (**in** *complete-lattice*) *Sup-set-fold* [*code-unfold del*]
declare *rev-foldl-cons* [*code del*]

Fold combinator with canonical argument order

primrec *fold* :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*a list* ⇒ '*b* ⇒ '*b* **where**
  fold f [] = *id*
| *fold f* (*x* # *xs*) = *fold f xs* ∘ *f x*

lemma *foldl-fold*:
  foldl f s xs = *fold* (λ*x s*. *f s x*) *xs s*
  ⟨*proof*⟩

lemma *foldr-fold-rev*:
  foldr f xs = *fold f* (*rev xs*)
  ⟨*proof*⟩

lemma *fold-rev-conv* [*code-unfold*]:
  fold f (*rev xs*) = *foldr f xs*
  ⟨*proof*⟩

lemma *fold-cong* [*fundef-cong*, *recdef-cong*]:
  a = *b* ⟹ *xs* = *ys* ⟹ (⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *g x*)
    ⟹ *fold f xs a* = *fold g ys b*
  ⟨*proof*⟩

lemma *fold-id*:
  assumes ⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *id*

shows *fold f xs = id*
⟨*proof*⟩

lemma *fold-apply*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow h \circ g\ x = f\ x \circ h$
shows $h \circ fold\ g\ xs = fold\ f\ xs \circ h$
⟨*proof*⟩

lemma *fold-invariant*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow Q\ x$ **and** $P\ s$
and $\bigwedge x\ s.\ Q\ x \Longrightarrow P\ s \Longrightarrow P\ (f\ x\ s)$
shows $P\ (fold\ f\ xs\ s)$
⟨*proof*⟩

lemma *fold-weak-invariant*:
assumes $P\ s$
and $\bigwedge s\ x.\ x \in set\ xs \Longrightarrow P\ s \Longrightarrow P\ (f\ x\ s)$
shows $P\ (fold\ f\ xs\ s)$
⟨*proof*⟩

lemma *fold-append* [*simp*]:
$fold\ f\ (xs\ @\ ys) = fold\ f\ ys \circ fold\ f\ xs$
⟨*proof*⟩

lemma *fold-map* [*code-unfold*]:
$fold\ g\ (map\ f\ xs) = fold\ (g\ o\ f)\ xs$
⟨*proof*⟩

lemma *fold-rev*:
assumes $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow f\ y \circ f\ x = f\ x \circ f\ y$
shows $fold\ f\ (rev\ xs) = fold\ f\ xs$
⟨*proof*⟩

lemma *foldr-fold*:
assumes $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow f\ y \circ f\ x = f\ x \circ f\ y$
shows $foldr\ f\ xs = fold\ f\ xs$
⟨*proof*⟩

lemma *fold-Cons-rev*:
$fold\ Cons\ xs = append\ (rev\ xs)$
⟨*proof*⟩

lemma *rev-conv-fold* [*code*]:
$rev\ xs = fold\ Cons\ xs\ []$
⟨*proof*⟩

lemma *fold-append-concat-rev*:
$fold\ append\ xss = append\ (concat\ (rev\ xss))$
⟨*proof*⟩

lemma *concat-conv-foldr* [*code*]:
concat xss = *foldr append xss* []
⟨*proof*⟩

lemma *fold-plus-listsum-rev*:
fold plus xs = *plus* (*listsum* (*rev xs*))
⟨*proof*⟩

lemma *listsum-conv-foldr* [*code*]:
listsum xs = *foldr plus xs 0*
⟨*proof*⟩

lemma *sort-key-conv-fold*:
assumes *inj-on f* (*set xs*)
shows *sort-key f xs* = *fold* (*insort-key f*) *xs* []
⟨*proof*⟩

lemma *sort-conv-fold*:
sort xs = *fold insort xs* []
⟨*proof*⟩

Finite-Set.fold and *fold*

lemma (**in** *fun-left-comm*) *fold-set-remdups*:
Finite-Set.fold f y (*set xs*) = *fold f* (*remdups xs*) *y*
⟨*proof*⟩

lemma (**in** *fun-left-comm-idem*) *fold-set*:
Finite-Set.fold f y (*set xs*) = *fold f xs y*
⟨*proof*⟩

lemma (**in** *ab-semigroup-idem-mult*) *fold1-set*:
assumes *xs* ≠ []
shows *Finite-Set.fold1 times* (*set xs*) = *fold times* (*tl xs*) (*hd xs*)
⟨*proof*⟩

lemma (**in** *lattice*) *Inf-fin-set-fold*:
Inf-fin (*set* (*x* # *xs*)) = *fold inf xs x*
⟨*proof*⟩

lemma (**in** *lattice*) *Inf-fin-set-foldr* [*code-unfold*]:
Inf-fin (*set* (*x* # *xs*)) = *foldr inf xs x*
⟨*proof*⟩

lemma (**in** *lattice*) *Sup-fin-set-fold*:
Sup-fin (*set* (*x* # *xs*)) = *fold sup xs x*
⟨*proof*⟩

lemma (**in** *lattice*) *Sup-fin-set-foldr* [*code-unfold*]:

Sup-fin (*set* (*x* # *xs*)) = *foldr sup xs x*
⟨*proof*⟩

lemma (**in** *linorder*) *Min-fin-set-fold*:
Min (*set* (*x* # *xs*)) = *fold min xs x*
⟨*proof*⟩

lemma (**in** *linorder*) *Min-fin-set-foldr* [*code-unfold*]:
Min (*set* (*x* # *xs*)) = *foldr min xs x*
⟨*proof*⟩

lemma (**in** *linorder*) *Max-fin-set-fold*:
Max (*set* (*x* # *xs*)) = *fold max xs x*
⟨*proof*⟩

lemma (**in** *linorder*) *Max-fin-set-foldr* [*code-unfold*]:
Max (*set* (*x* # *xs*)) = *foldr max xs x*
⟨*proof*⟩

lemma (**in** *complete-lattice*) *Inf-set-fold*:
Inf (*set xs*) = *fold inf xs top*
⟨*proof*⟩

lemma (**in** *complete-lattice*) *Inf-set-foldr* [*code-unfold*]:
Inf (*set xs*) = *foldr inf xs top*
⟨*proof*⟩

lemma (**in** *complete-lattice*) *Sup-set-fold*:
Sup (*set xs*) = *fold sup xs bot*
⟨*proof*⟩

lemma (**in** *complete-lattice*) *Sup-set-foldr* [*code-unfold*]:
Sup (*set xs*) = *foldr sup xs bot*
⟨*proof*⟩

lemma (**in** *complete-lattice*) *INFI-set-fold*:
INFI (*set xs*) *f* = *fold* (*inf* ∘ *f*) *xs top*
⟨*proof*⟩

lemma (**in** *complete-lattice*) *SUPR-set-fold*:
SUPR (*set xs*) *f* = *fold* (*sup* ∘ *f*) *xs bot*
⟨*proof*⟩

nth-map

definition *nth-map* :: *nat* ⇒ (*'a* ⇒ *'a*) ⇒ *'a list* ⇒ *'a list* **where**
nth-map n f xs = (*if n* < *length xs then*
take n xs @ [*f* (*xs* ! *n*)] @ *drop* (*Suc n*) *xs*
else xs)

lemma *nth-map-id*:
$n \geq length\ xs \Longrightarrow nth\text{-}map\ n\ f\ xs = xs$
⟨*proof*⟩

lemma *nth-map-unfold*:
$n < length\ xs \Longrightarrow nth\text{-}map\ n\ f\ xs = take\ n\ xs\ @\ [f\ (xs\ !\ n)]\ @\ drop\ (Suc\ n)\ xs$
⟨*proof*⟩

lemma *nth-map-Nil* [*simp*]:
$nth\text{-}map\ n\ f\ [] = []$
⟨*proof*⟩

lemma *nth-map-zero* [*simp*]:
$nth\text{-}map\ 0\ f\ (x\ \#\ xs) = f\ x\ \#\ xs$
⟨*proof*⟩

lemma *nth-map-Suc* [*simp*]:
$nth\text{-}map\ (Suc\ n)\ f\ (x\ \#\ xs) = x\ \#\ nth\text{-}map\ n\ f\ xs$
⟨*proof*⟩

end

26 More-Set: Relating (finite) sets and lists

theory *More-Set*
imports *Main More-List*
begin

26.1 Various additional set functions

definition *is-empty* :: *'a set* ⇒ *bool* **where**
$is\text{-}empty\ A \longleftrightarrow A = \{\}$

definition *remove* :: *'a* ⇒ *'a set* ⇒ *'a set* **where**
$remove\ x\ A = A - \{x\}$

lemma *fun-left-comm-idem-remove*:
fun-left-comm-idem remove
⟨*proof*⟩

lemma *minus-fold-remove*:
assumes *finite A*
shows $B - A = Finite\text{-}Set.fold\ remove\ B\ A$
⟨*proof*⟩

definition *project* :: (*'a* ⇒ *bool*) ⇒ *'a set* ⇒ *'a set* **where**
$project\ P\ A = \{a{\in}A.\ P\ a\}$

26.2 Basic set operations

lemma *is-empty-set*:
is-empty (*set xs*) ⟷ *null xs*
⟨*proof*⟩

lemma *ball-set*:
(∀ *x*∈*set xs*. *P x*) ⟷ *list-all P xs*
⟨*proof*⟩

lemma *bex-set*:
(∃ *x*∈*set xs*. *P x*) ⟷ *list-ex P xs*
⟨*proof*⟩

lemma *empty-set*:
{} = *set* []
⟨*proof*⟩

lemma *insert-set-compl*:
insert x (− *set xs*) = − *set* (*removeAll x xs*)
⟨*proof*⟩

lemma *remove-set-compl*:
remove x (− *set xs*) = − *set* (*List.insert x xs*)
⟨*proof*⟩

lemma *image-set*:
image f (*set xs*) = *set* (*map f xs*)
⟨*proof*⟩

lemma *project-set*:
project P (*set xs*) = *set* (*filter P xs*)
⟨*proof*⟩

26.3 Functorial set operations

lemma *union-set*:
set xs ∪ *A* = *fold Set.insert xs A*
⟨*proof*⟩

lemma *union-set-foldr*:
set xs ∪ *A* = *foldr Set.insert xs A*
⟨*proof*⟩

lemma *minus-set*:
A − *set xs* = *fold remove xs A*
⟨*proof*⟩

lemma *minus-set-foldr*:
A − *set xs* = *foldr remove xs A*

⟨*proof*⟩

26.4 Derived set operations

lemma *member*:
$a \in A \longleftrightarrow (\exists\, x{\in}A.\ a = x)$
⟨*proof*⟩

lemma *subset-eq*:
$A \subseteq B \longleftrightarrow (\forall\, x{\in}A.\ x \in B)$
⟨*proof*⟩

lemma *subset*:
$A \subset B \longleftrightarrow A \subseteq B \wedge \neg\ B \subseteq A$
⟨*proof*⟩

lemma *set-eq*:
$A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
⟨*proof*⟩

lemma *inter*:
$A \cap B = project\ (\lambda x.\ x \in A)\ B$
⟨*proof*⟩

26.5 Various lemmas

lemma *not-set-compl*:
$Not \circ set\ xs = -\ set\ xs$
⟨*proof*⟩

end

27 Fset: Executable finite sets

theory *Fset*
imports *More-Set More-List*
begin

declare *mem-def* [*simp*]

27.1 Lifting

datatype $'a\ fset = Fset\ 'a\ set$

primrec *member* :: $'a\ fset \Rightarrow 'a\ set$ **where**
$member\ (Fset\ A) = A$

lemma *member-inject* [*simp*]:

member A = member B ⟹ A = B
⟨*proof*⟩

lemma *Fset-member* [*simp*]:
Fset (member A) = A
⟨*proof*⟩

definition *Set* :: *'a list ⇒ 'a fset* **where**
Set xs = Fset (set xs)

lemma *member-Set* [*simp*]:
member (Set xs) = set xs
⟨*proof*⟩

definition *Coset* :: *'a list ⇒ 'a fset* **where**
Coset xs = Fset (− set xs)

lemma *member-Coset* [*simp*]:
member (Coset xs) = − set xs
⟨*proof*⟩

code-datatype *Set Coset*

lemma *member-code* [*code*]:
member (Set xs) = List.member xs
member (Coset xs) = Not ∘ List.member xs
⟨*proof*⟩

lemma *member-image-UNIV* [*simp*]:
member ‘ UNIV = UNIV
⟨*proof*⟩

27.2 Lattice instantiation

instantiation *fset* :: (*type*) *boolean-algebra*
begin

definition *less-eq-fset* :: *'a fset ⇒ 'a fset ⇒ bool* **where**
[*simp*]: *A ≤ B ⟷ member A ⊆ member B*

definition *less-fset* :: *'a fset ⇒ 'a fset ⇒ bool* **where**
[*simp*]: *A < B ⟷ member A ⊂ member B*

definition *inf-fset* :: *'a fset ⇒ 'a fset ⇒ 'a fset* **where**
[*simp*]: *inf A B = Fset (member A ∩ member B)*

definition *sup-fset* :: *'a fset ⇒ 'a fset ⇒ 'a fset* **where**
[*simp*]: *sup A B = Fset (member A ∪ member B)*

definition *bot-fset* :: *'a fset* **where**
[*simp*]: *bot* = *Fset* {}

definition *top-fset* :: *'a fset* **where**
[*simp*]: *top* = *Fset UNIV*

definition *uminus-fset* :: *'a fset* ⇒ *'a fset* **where**
[*simp*]: − *A* = *Fset* (− (*member A*))

definition *minus-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **where**
[*simp*]: *A* − *B* = *Fset* (*member A* − *member B*)

instance ⟨*proof*⟩

end

instantiation *fset* :: (*type*) *complete-lattice*
begin

definition *Inf-fset* :: *'a fset set* ⇒ *'a fset* **where**
[*simp*, *code del*]: *Inf-fset As* = *Fset* (*Inf* (*image member As*))

definition *Sup-fset* :: *'a fset set* ⇒ *'a fset* **where**
[*simp*, *code del*]: *Sup-fset As* = *Fset* (*Sup* (*image member As*))

instance ⟨*proof*⟩

end

27.3 Basic operations

definition *is-empty* :: *'a fset* ⇒ *bool* **where**
[*simp*]: *is-empty A* ⟷ *More-Set.is-empty* (*member A*)

lemma *is-empty-Set* [*code*]:
is-empty (*Set xs*) ⟷ *null xs*
⟨*proof*⟩

lemma *empty-Set* [*code*]:
bot = *Set* []
⟨*proof*⟩

lemma *UNIV-Set* [*code*]:
top = *Coset* []
⟨*proof*⟩

definition *insert* :: *'a* ⇒ *'a fset* ⇒ *'a fset* **where**
[*simp*]: *insert x A* = *Fset* (*Set.insert x* (*member A*))

lemma *insert-Set* [*code*]:
insert x (*Set xs*) = *Set* (*List.insert x xs*)
insert x (*Coset xs*) = *Coset* (*removeAll x xs*)
⟨*proof*⟩

definition *remove* :: *'a* ⇒ *'a fset* ⇒ *'a fset* **where**
[*simp*]: *remove x A* = *Fset* (*More-Set.remove x* (*member A*))

lemma *remove-Set* [*code*]:
remove x (*Set xs*) = *Set* (*removeAll x xs*)
remove x (*Coset xs*) = *Coset* (*List.insert x xs*)
⟨*proof*⟩

definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a fset* ⇒ *'b fset* **where**
[*simp*]: *map f A* = *Fset* (*image f* (*member A*))

lemma *map-Set* [*code*]:
map f (*Set xs*) = *Set* (*remdups* (*List.map f xs*))
⟨*proof*⟩

definition *filter* :: (*'a* ⇒ *bool*) ⇒ *'a fset* ⇒ *'a fset* **where**
[*simp*]: *filter P A* = *Fset* (*More-Set.project P* (*member A*))

lemma *filter-Set* [*code*]:
filter P (*Set xs*) = *Set* (*List.filter P xs*)
⟨*proof*⟩

definition *forall* :: (*'a* ⇒ *bool*) ⇒ *'a fset* ⇒ *bool* **where**
[*simp*]: *forall P A* ⟷ *Ball* (*member A*) *P*

lemma *forall-Set* [*code*]:
forall P (*Set xs*) ⟷ *list-all P xs*
⟨*proof*⟩

definition *exists* :: (*'a* ⇒ *bool*) ⇒ *'a fset* ⇒ *bool* **where**
[*simp*]: *exists P A* ⟷ *Bex* (*member A*) *P*

lemma *exists-Set* [*code*]:
exists P (*Set xs*) ⟷ *list-ex P xs*
⟨*proof*⟩

definition *card* :: *'a fset* ⇒ *nat* **where**
[*simp*]: *card A* = *Finite-Set.card* (*member A*)

lemma *card-Set* [*code*]:
card (*Set xs*) = *length* (*remdups xs*)
⟨*proof*⟩

lemma *compl-Set* [*simp*, *code*]:

– *Set xs = Coset xs*
⟨*proof*⟩

lemma *compl-Coset* [*simp*, *code*]:
– *Coset xs = Set xs*
⟨*proof*⟩

27.4 Derived operations

lemma *subfset-eq-forall* [*code*]:
A ≤ B ⟷ forall (member B) A
⟨*proof*⟩

lemma *subfset-subfset-eq* [*code*]:
A < B ⟷ A ≤ B ∧ ¬ B ≤ (A :: 'a fset)
⟨*proof*⟩

lemma *eq-fset-subfset-eq* [*code*]:
eq-class.eq A B ⟷ A ≤ B ∧ B ≤ (A :: 'a fset)
⟨*proof*⟩

27.5 Functorial operations

lemma *inter-project* [*code*]:
inf A (Set xs) = Set (List.filter (member A) xs)
inf A (Coset xs) = foldr remove xs A
⟨*proof*⟩

lemma *subtract-remove* [*code*]:
A – Set xs = foldr remove xs A
A – Coset xs = Set (List.filter (member A) xs)
⟨*proof*⟩

lemma *union-insert* [*code*]:
sup (Set xs) A = foldr insert xs A
sup (Coset xs) A = Coset (List.filter (Not ∘ member A) xs)
⟨*proof*⟩

context *complete-lattice*
begin

definition *Infimum* :: *'a fset ⇒ 'a* **where**
[*simp*]: *Infimum A = Inf (member A)*

lemma *Infimum-inf* [*code*]:
Infimum (Set As) = foldr inf As top
Infimum (Coset []) = bot
⟨*proof*⟩

definition *Supremum* :: *'a fset ⇒ 'a* **where**

[*simp*]: *Supremum A* = *Sup* (*member A*)

lemma *Supremum-sup* [*code*]:
Supremum (*Set As*) = *foldr sup As bot*
Supremum (*Coset* []) = *top*
⟨*proof*⟩

end

27.6 Misc operations

lemma *size-fset* [*code*]:
fset-size f A = *0*
size A = *0*
⟨*proof*⟩

lemma *fset-case-code* [*code*]:
fset-case f A = *f* (*member A*)
⟨*proof*⟩

lemma *fset-rec-code* [*code*]:
fset-rec f A = *f* (*member A*)
⟨*proof*⟩

27.7 Simplified simprules

lemma *is-empty-simp* [*simp*]:
is-empty A ⟷ *member A* = {}
⟨*proof*⟩
declare *is-empty-def* [*simp del*]

lemma *remove-simp* [*simp*]:
remove x A = *Fset* (*member A* − {*x*})
⟨*proof*⟩
declare *remove-def* [*simp del*]

lemma *filter-simp* [*simp*]:
filter P A = *Fset* {*x* ∈ *member A*. *P x*}
⟨*proof*⟩
declare *filter-def* [*simp del*]

declare *mem-def* [*simp del*]

hide-const (**open**) *is-empty insert remove map filter forall exists card Inter Union*

end

28 Dlist: Lists with elements distinct as canonical example for datatype invariants

theory *Dlist*
imports *Main More-List Fset*
begin

29 The type of distinct lists

typedef (**open**) *'a dlist* = {*xs*::*'a list*. *distinct xs*}
  morphisms *list-of-dlist Abs-dlist*
⟨*proof*⟩

lemma *dlist-ext*:
  assumes *list-of-dlist xs* = *list-of-dlist ys*
  shows *xs* = *ys*
  ⟨*proof*⟩

Formal, totalized constructor for *'a dlist*:

definition *Dlist* :: *'a list* ⇒ *'a dlist* **where**
  [*code del*]: *Dlist xs* = *Abs-dlist* (*remdups xs*)

lemma *distinct-list-of-dlist* [*simp*]:
  distinct (*list-of-dlist dxs*)
  ⟨*proof*⟩

lemma *list-of-dlist-Dlist* [*simp*]:
  list-of-dlist (*Dlist xs*) = *remdups xs*
  ⟨*proof*⟩

lemma *Dlist-list-of-dlist* [*simp*, *code abstype*]:
  Dlist (*list-of-dlist dxs*) = *dxs*
  ⟨*proof*⟩

Fundamental operations:

definition *empty* :: *'a dlist* **where**
  empty = *Dlist* []

definition *insert* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
  insert x dxs = *Dlist* (*List.insert x* (*list-of-dlist dxs*))

definition *remove* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
  remove x dxs = *Dlist* (*remove1 x* (*list-of-dlist dxs*))

definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b dlist* **where**
  map f dxs = *Dlist* (*remdups* (*List.map f* (*list-of-dlist dxs*)))

definition *filter* :: (*'a* ⇒ *bool*) ⇒ *'a dlist* ⇒ *'a dlist* **where**

filter P dxs = Dlist (List.filter P (list-of-dlist dxs))

Derived operations:

definition *null :: 'a dlist ⇒ bool* **where**
null dxs = List.null (list-of-dlist dxs)

definition *member :: 'a dlist ⇒ 'a ⇒ bool* **where**
member dxs = List.member (list-of-dlist dxs)

definition *length :: 'a dlist ⇒ nat* **where**
length dxs = List.length (list-of-dlist dxs)

definition *fold :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a dlist ⇒ 'b ⇒ 'b* **where**
fold f dxs = More-List.fold f (list-of-dlist dxs)

definition *foldr :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a dlist ⇒ 'b ⇒ 'b* **where**
foldr f dxs = List.foldr f (list-of-dlist dxs)

30 Executable version obeying invariant

lemma *list-of-dlist-empty* [*simp*, *code abstract*]:
list-of-dlist empty = []
⟨*proof*⟩

lemma *list-of-dlist-insert* [*simp*, *code abstract*]:
list-of-dlist (insert x dxs) = List.insert x (list-of-dlist dxs)
⟨*proof*⟩

lemma *list-of-dlist-remove* [*simp*, *code abstract*]:
list-of-dlist (remove x dxs) = remove1 x (list-of-dlist dxs)
⟨*proof*⟩

lemma *list-of-dlist-map* [*simp*, *code abstract*]:
list-of-dlist (map f dxs) = remdups (List.map f (list-of-dlist dxs))
⟨*proof*⟩

lemma *list-of-dlist-filter* [*simp*, *code abstract*]:
list-of-dlist (filter P dxs) = List.filter P (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*⟩

31 Induction principle and case distinction

lemma *dlist-induct* [*case-names empty insert*, *induct type*: *dlist*]:
assumes *empty*: *P empty*
assumes *insrt*: $\bigwedge x\ dxs.\ \neg\ member\ dxs\ x \Longrightarrow P\ dxs \Longrightarrow P\ (insert\ x\ dxs)$
shows *P dxs*
⟨*proof*⟩

lemma *dlist-case* [*case-names empty insert*, *cases type*: *dlist*]:
assumes *empty*: *dxs* = *empty* ⟹ *P*
assumes *insert*: $\bigwedge x\ dys.\ \neg\ member\ dys\ x \Longrightarrow dxs = insert\ x\ dys \Longrightarrow P$
shows *P*
⟨*proof*⟩

32 Implementation of sets by distinct lists – canonical!

definition *Set* :: *'a dlist* ⇒ *'a fset* **where**
Set dxs = *Fset.Set* (*list-of-dlist dxs*)

definition *Coset* :: *'a dlist* ⇒ *'a fset* **where**
Coset dxs = *Fset.Coset* (*list-of-dlist dxs*)

code-datatype *Set Coset*

declare *member-code* [*code del*]
declare *is-empty-Set* [*code del*]
declare *empty-Set* [*code del*]
declare *UNIV-Set* [*code del*]
declare *insert-Set* [*code del*]
declare *remove-Set* [*code del*]
declare *compl-Set* [*code del*]
declare *compl-Coset* [*code del*]
declare *map-Set* [*code del*]
declare *filter-Set* [*code del*]
declare *forall-Set* [*code del*]
declare *exists-Set* [*code del*]
declare *card-Set* [*code del*]
declare *inter-project* [*code del*]
declare *subtract-remove* [*code del*]
declare *union-insert* [*code del*]
declare *Infimum-inf* [*code del*]
declare *Supremum-sup* [*code del*]

lemma *Set-Dlist* [*simp*]:
Set (*Dlist xs*) = *Fset* (*set xs*)
⟨*proof*⟩

lemma *Coset-Dlist* [*simp*]:
Coset (*Dlist xs*) = *Fset* (− *set xs*)
⟨*proof*⟩

lemma *member-Set* [*simp*]:
Fset.member (*Set dxs*) = *List.member* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *member-Coset* [*simp*]:
Fset.member (*Coset dxs*) = *Not* ∘ *List.member* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *Set-dlist-of-list* [*code*]:
Fset.Set xs = *Set* (*dlist-of-list xs*)
⟨*proof*⟩

lemma *Coset-dlist-of-list* [*code*]:
Fset.Coset xs = *Coset* (*dlist-of-list xs*)
⟨*proof*⟩

lemma *is-empty-Set* [*code*]:
Fset.is-empty (*Set dxs*) ⟷ *null dxs*
⟨*proof*⟩

lemma *bot-code* [*code*]:
bot = *Set empty*
⟨*proof*⟩

lemma *top-code* [*code*]:
top = *Coset empty*
⟨*proof*⟩

lemma *insert-code* [*code*]:
Fset.insert x (*Set dxs*) = *Set* (*insert x dxs*)
Fset.insert x (*Coset dxs*) = *Coset* (*remove x dxs*)
⟨*proof*⟩

lemma *remove-code* [*code*]:
Fset.remove x (*Set dxs*) = *Set* (*remove x dxs*)
Fset.remove x (*Coset dxs*) = *Coset* (*insert x dxs*)
⟨*proof*⟩

lemma *member-code* [*code*]:
Fset.member (*Set dxs*) = *member dxs*
Fset.member (*Coset dxs*) = *Not* ∘ *member dxs*
⟨*proof*⟩

lemma *compl-code* [*code*]:
− *Set dxs* = *Coset dxs*

– Coset dxs = Set dxs
⟨*proof*⟩

lemma *map-code* [*code*]:
Fset.map f (Set dxs) = Set (map f dxs)
⟨*proof*⟩

lemma *filter-code* [*code*]:
Fset.filter f (Set dxs) = Set (filter f dxs)
⟨*proof*⟩

lemma *forall-Set* [*code*]:
Fset.forall P (Set xs) ⟷ list-all P (list-of-dlist xs)
⟨*proof*⟩

lemma *exists-Set* [*code*]:
Fset.exists P (Set xs) ⟷ list-ex P (list-of-dlist xs)
⟨*proof*⟩

lemma *card-code* [*code*]:
Fset.card (Set dxs) = length dxs
⟨*proof*⟩

lemma *inter-code* [*code*]:
inf A (Set xs) = Set (filter (Fset.member A) xs)
inf A (Coset xs) = foldr Fset.remove xs A
⟨*proof*⟩

lemma *subtract-code* [*code*]:
A – Set xs = foldr Fset.remove xs A
A – Coset xs = Set (filter (Fset.member A) xs)
⟨*proof*⟩

lemma *union-code* [*code*]:
sup (Set xs) A = foldr Fset.insert xs A
sup (Coset xs) A = Coset (filter (Not ∘ Fset.member A) xs)
⟨*proof*⟩

context *complete-lattice*
begin

lemma *Infimum-code* [*code*]:
Infimum (Set As) = foldr inf As top
⟨*proof*⟩

lemma *Supremum-code* [*code*]:
Supremum (Set As) = foldr sup As bot
⟨*proof*⟩

end

hide-const (**open**) *member fold foldr empty insert remove map filter null member length fold*

end

33 Efficient-Nat: Implementation of natural numbers by target-language integers

theory *Efficient-Nat*
imports *Code-Integer Main*
begin

When generating code for functions on natural numbers, the canonical representation using *0* and *Suc* is unsuitable for computations involving large numbers. The efficiency of the generated code can be improved drastically by implementing natural numbers by target-language integers. To do this, just include this theory.

33.1 Basic arithmetic

Most standard arithmetic functions on natural numbers are implemented using their counterparts on the integers:

code-datatype *number-nat-inst.number-of-nat*

lemma *zero-nat-code* [*code*, *code-unfold-post*]:
0 = (*Numeral0* :: *nat*)
⟨*proof*⟩

lemma *one-nat-code* [*code*, *code-unfold-post*]:
1 = (*Numeral1* :: *nat*)
⟨*proof*⟩

lemma *Suc-code* [*code*]:
Suc n = *n* + *1*
⟨*proof*⟩

lemma *plus-nat-code* [*code*]:
n + *m* = *nat* (*of-nat n* + *of-nat m*)
⟨*proof*⟩

lemma *minus-nat-code* [*code*]:
n − *m* = *nat* (*of-nat n* − *of-nat m*)
⟨*proof*⟩

lemma *times-nat-code* [*code*]:
n ∗ *m* = *nat* (*of-nat n* ∗ *of-nat m*)
⟨*proof*⟩

Specialized *op div* and *op mod* operations.

definition *divmod-aux* :: *nat* ⇒ *nat* ⇒ *nat* × *nat* **where**
[*code del*]: *divmod-aux* = *divmod-nat*

lemma [*code*]:
divmod-nat n m = (*if m* = *0 then* (*0*, *n*) *else divmod-aux n m*)
⟨*proof*⟩

lemma *divmod-aux-code* [*code*]:
divmod-aux n m = (*nat* (*of-nat n div of-nat m*), *nat* (*of-nat n mod of-nat m*))
⟨*proof*⟩

lemma *eq-nat-code* [*code*]:
eq-class.eq n m ⟷ *eq-class.eq* (*of-nat n* :: *int*) (*of-nat m*)
⟨*proof*⟩

lemma *eq-nat-refl* [*code nbe*]:
eq-class.eq (*n*::*nat*) *n* ⟷ *True*
⟨*proof*⟩

lemma *less-eq-nat-code* [*code*]:
n ≤ *m* ⟷ (*of-nat n* :: *int*) ≤ *of-nat m*
⟨*proof*⟩

lemma *less-nat-code* [*code*]:
n < *m* ⟷ (*of-nat n* :: *int*) < *of-nat m*
⟨*proof*⟩

33.2 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code*, *code-unfold*]:
nat-case = (λ*f g n*. *if n* = *0 then f else g* (*n* − *1*))
⟨*proof*⟩

33.3 Preprocessors

In contrast to *Suc n*, the term *n* + *1* is no longer a constructor term. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a recursion equation or in the arguments of an inductive relation in an introduction rule) must be eliminated. This can be accomplished by applying the following transformation rules:

lemma *Suc-if-eq*: (⋀*n*. *f* (*Suc n*) ≡ *h n*) ⟹ *f 0* ≡ *g* ⟹

$f\ n \equiv if\ n = 0\ then\ g\ else\ h\ (n - 1)$
⟨*proof*⟩

lemma *Suc-clause*: $(\bigwedge n.\ P\ n\ (Suc\ n)) \Longrightarrow n \neq 0 \Longrightarrow P\ (n - 1)\ n$
⟨*proof*⟩

The rules above are built into a preprocessor that is plugged into the code generator. Since the preprocessor for introduction rules does not know anything about modes, some of the modes that worked for the canonical representation of natural numbers may no longer work.

⟨*ML*⟩

33.4 Target language setup

For ML, we map *nat* to target language integers, where we ensure that values are always non-negative.

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

types-code
nat (*int*)
attach (*term-of*) ⟪
val term-of-nat = HOLogic.mk-number HOLogic.natT;
⟫
attach (*test*) ⟪
fun gen-nat i =
let val n = random-range 0 i
in (*n*, *fn* () => *term-of-nat n*) *end*;
⟫

For Haskell ans Scala we define our own *nat* type. The reason is that we have to distinguish type class instances for *nat* and *int*.

code-include *Haskell Nat* ⟪
newtype Nat = Nat Integer deriving (*Eq*, *Show*, *Read*);

instance Num Nat where {
fromInteger k = Nat (*if k >= 0 then k else 0*);
Nat n + Nat m = Nat (*n + m*);
Nat n - Nat m = fromInteger (*n - m*);
*Nat n * Nat m = Nat* (*n * m*);
abs n = n;
signum - = 1;
negate n = error negate Nat;
};

instance Ord Nat where {
Nat n <= Nat m = n <= m;

```
  Nat n < Nat m = n < m;
};

instance Real Nat where {
  toRational (Nat n) = toRational n;
};

instance Enum Nat where {
  toEnum k = fromInteger (toEnum k);
  fromEnum (Nat n) = fromEnum n;
};

instance Integral Nat where {
  toInteger (Nat n) = n;
  divMod n m = quotRem n m;
  quotRem (Nat n) (Nat m) = (Nat k, Nat l) where (k, l) = quotRem n m;
};
⟫
```

code-reserved *Haskell Nat*

code-include *Scala Nat* ⟪

```
import scala.Math

object Nat {

  def apply(numeral: BigInt): Nat = new Nat(numeral max 0)
  def apply(numeral: Int): Nat = Nat(BigInt(numeral))
  def apply(numeral: String): Nat = Nat(BigInt(numeral))

}

class Nat private(private val value: BigInt) {

  override def hashCode(): Int = this.value.hashCode()

  override def equals(that: Any): Boolean = that match {
    case that: Nat => this equals that
    case - => false
  }

  override def toString(): String = this.value.toString

  def equals(that: Nat): Boolean = this.value == that.value

  def as-BigInt: BigInt = this.value
 def as-Int: Int = if (this.value >= Math.MAX-INT && this.value <= Math.MAX-INT)
      this.value.intValue
    else error(Int value too big: + this.value.toString)
```

```
  def +(that: Nat): Nat = new Nat(this.value + that.value)
  def −(that: Nat): Nat = Nat(this.value − that.value)
  def ∗(that: Nat): Nat = new Nat(this.value ∗ that.value)

  def /%(that: Nat): (Nat, Nat) = if (that.value == 0) (new Nat(0), this)
    else {
      val (k, l) = this.value /% that.value
      (new Nat(k), new Nat(l))
    }

  def <=(that: Nat): Boolean = this.value <= that.value

  def <(that: Nat): Boolean = this.value < that.value

}
⟩⟩
```

code-reserved *Scala Nat*

code-type *nat*
  (*Haskell Nat.Nat*)
  (*Scala Nat.Nat*)

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

Natural numerals.

lemma [*code-unfold-post*]:
  nat (*number-of i*) = *number-nat-inst.number-of-nat i*
  — this interacts as desired with *number-of ?v* = *nat* (*number-of ?v*)
  ⟨*proof*⟩

⟨*ML*⟩

Since natural numbers are implemented using integers in ML, the coercion function *of-nat* of type *nat* ⇒ *int* is simply implemented by the identity function. For the *nat* function for converting an integer to a natural number, we give a specific implementation using an ML function that returns its input value, provided that it is non-negative, and otherwise returns *0*.

definition *int* :: *nat* ⇒ *int* **where**
  [*code del*]: *int* = *of-nat*

lemma *int-code′* [*code*]:
  int (*number-of l*) = (*if neg* (*number-of l* :: *int*) *then 0 else number-of l*)
  ⟨*proof*⟩

lemma *nat-code′* [*code*]:
  nat (*number-of l*) = (*if neg* (*number-of l* :: *int*) *then 0 else number-of l*)

⟨*proof*⟩

lemma *of-nat-int* [*code-unfold-post*]:
of-nat = *int* ⟨*proof*⟩

lemma *of-nat-aux-int* [*code-unfold*]:
of-nat-aux (λ*i*. *i* + *1*) *k 0* = *int k*
⟨*proof*⟩

code-const *int*
(*SML* -)
(*OCaml* -)

consts-code
int ((-))
nat (⟨**module**⟩*nat*)
attach ⟪
fun nat i = if i < 0 then 0 else i;
⟫

code-const *nat*
(*SML IntInf.max*/ (/*0*,/ -))
(*OCaml Big'-int.max'-big'-int*/ *Big'-int.zero'-big'-int*)

For Haskell and Scala, things are slightly different again.

code-const *int* **and** *nat*
(*Haskell toInteger* **and** *fromInteger*)
(*Scala* !-.*as'-BigInt* **and** !*Nat.Nat*((-)))

Conversion from and to indices.

code-const *Code-Numeral.of-nat*
(*SML IntInf.toInt*)
(*OCaml* -)
(*Haskell fromEnum*)
(*Scala* !-.*as'-Int*)

code-const *Code-Numeral.nat-of*
(*SML IntInf.fromInt*)
(*OCaml* -)
(*Haskell toEnum*)
(*Scala* !*Nat.Nat*((-)))

Using target language arithmetic operations whenever appropriate

code-const *op* + :: *nat* ⇒ *nat* ⇒ *nat*
(*SML IntInf.*+ ((-), (-)))
(*OCaml Big'-int.add'-big'-int*)
(*Haskell* **infixl** *6* +)
(*Scala* **infixl** *7* +)

code-const *op −* :: *nat* ⇒ *nat* ⇒ *nat*
(*Haskell* **infixl** *6 −*)
(*Scala* **infixl** *7 −*)

code-const *op ∗* :: *nat* ⇒ *nat* ⇒ *nat*
(*SML IntInf.∗* ((-), (-)))
(*OCaml Big'-int.mult'-big'-int*)
(*Haskell* **infixl** *7 ∗*)
(*Scala* **infixl** *8 ∗*)

code-const *divmod-aux*
(*SML IntInf.divMod/* ((-),/ (-)))
(*OCaml Big'-int.quomod'-big'-int*)
(*Haskell divMod*)
(*Scala* **infixl** *8 /%*)

code-const *divmod-nat*
(*Haskell divMod*)
(*Scala* **infixl** *8 /%*)

code-const *eq-class.eq* :: *nat* ⇒ *nat* ⇒ *bool*
(*SML* !((- : *IntInf.int*) = -))
(*OCaml Big'-int.eq'-big'-int*)
(*Haskell* **infixl** *4 ==*)
(*Scala* **infixl** *5 ==*)

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

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

consts-code
0::*nat* (*0*)
1::*nat* (*1*)
Suc ((- +/ *1*))
op + :: *nat* ⇒ *nat* ⇒ *nat* ((- +/ -))
op ∗ :: *nat* ⇒ *nat* ⇒ *nat* ((- */ -))
op ≤ :: *nat* ⇒ *nat* ⇒ *bool* ((- <=/ -))
op < :: *nat* ⇒ *nat* ⇒ *bool* ((- </ -))

Evaluation

lemma [*code*, *code del*]:

(*Code-Evaluation.term-of* :: *nat* ⇒ *term*) = *Code-Evaluation.term-of* ⟨*proof*⟩

code-const *Code-Evaluation.term-of* :: *nat* ⇒ *term*
(*SML HOLogic.mk'-number*/ *HOLogic.natT*)

Module names

code-modulename *SML*
Efficient-Nat Arith

code-modulename *OCaml*
Efficient-Nat Arith

code-modulename *Haskell*
Efficient-Nat Arith

hide-const *int*

end

34 Enum: Finite types as explicit enumerations

theory *Enum*
imports *Map Main*
begin

34.1 Class *enum*

class *enum* =
fixes *enum* :: *'a list*
assumes *UNIV-enum*: *UNIV* = *set enum*
and *enum-distinct*: *distinct enum*
begin

subclass *finite* ⟨*proof*⟩

lemma *enum-all*: *set enum* = *UNIV* ⟨*proof*⟩

lemma *in-enum* [*intro*]: *x* ∈ *set enum*
⟨*proof*⟩

lemma *enum-eq-I*:
assumes ⋀*x*. *x* ∈ *set xs*
shows *set enum* = *set xs*
⟨*proof*⟩

end

34.2 Equality and order on functions

instantiation *fun* :: (*enum*, *eq*) *eq*
begin

definition
eq-class.eq f g ⟷ (∀ *x* ∈ *set enum*. *f x* = *g x*)

instance ⟨*proof*⟩

end

lemma *order-fun* [*code*]:
fixes *f g* :: *'a*::*enum* ⇒ *'b*::*order*
shows *f* ≤ *g* ⟷ *list-all* (λ*x*. *f x* ≤ *g x*) *enum*
and *f* < *g* ⟷ *f* ≤ *g* ∧ ¬ *list-all* (λ*x*. *f x* = *g x*) *enum*
⟨*proof*⟩

34.3 Quantifiers

lemma *all-code* [*code*]: (∀ *x*. *P x*) ⟷ *list-all P enum*
⟨*proof*⟩

lemma *exists-code* [*code*]: (∃ *x*. *P x*) ⟷ ¬ *list-all* (*Not o P*) *enum*
⟨*proof*⟩

34.4 Default instances

primrec *n-lists* :: *nat* ⇒ *'a list* ⇒ *'a list list* **where**
n-lists 0 xs = [[]]
| *n-lists* (*Suc n*) *xs* = *concat* (*map* (λ*ys*. *map* (λ*y*. *y* # *ys*) *xs*) (*n-lists n xs*))

lemma *n-lists-Nil* [*simp*]: *n-lists n* [] = (*if n* = *0 then* [[]] *else* [])
⟨*proof*⟩

lemma *length-n-lists*: *length* (*n-lists n xs*) = *length xs* ^ *n*
⟨*proof*⟩

lemma *length-n-lists-elem*: *ys* ∈ *set* (*n-lists n xs*) ⟹ *length ys* = *n*
⟨*proof*⟩

lemma *set-n-lists*: *set* (*n-lists n xs*) = {*ys*. *length ys* = *n* ∧ *set ys* ⊆ *set xs*}
⟨*proof*⟩

lemma *distinct-n-lists*:
assumes *distinct xs*
shows *distinct* (*n-lists n xs*)
⟨*proof*⟩

lemma *map-of-zip-enum-is-Some*:

assumes *length ys = length* (*enum* :: *'a*::*enum list*)
shows ∃ *y*. *map-of* (*zip* (*enum* :: *'a*::*enum list*) *ys*) *x* = *Some y*
⟨*proof*⟩

lemma *map-of-zip-enum-inject*:
fixes *xs ys* :: *'b*::*enum list*
assumes *length*: *length xs = length* (*enum* :: *'a*::*enum list*)
length ys = length (*enum* :: *'a*::*enum list*)
and *map-of*: *the* ∘ *map-of* (*zip* (*enum* :: *'a*::*enum list*) *xs*) = *the* ∘ *map-of* (*zip* (*enum* :: *'a*::*enum list*) *ys*)
shows *xs = ys*
⟨*proof*⟩

instantiation *fun* :: (*enum*, *enum*) *enum*
begin

definition
[*code del*]: *enum = map* (λ*ys*. *the o map-of* (*zip* (*enum*::*'a list*) *ys*)) (*n-lists* (*length* (*enum*::*'a*::*enum list*)) *enum*)

instance ⟨*proof*⟩

end

lemma *enum-fun-code* [*code*]: *enum* = (*let enum-a* = (*enum* :: *'a*::{*enum*, *eq*} *list*)
in map (λ*ys*. *the o map-of* (*zip enum-a ys*)) (*n-lists* (*length enum-a*) *enum*))
⟨*proof*⟩

instantiation *unit* :: *enum*
begin

definition
enum = [()]

instance ⟨*proof*⟩

end

instantiation *bool* :: *enum*
begin

definition
enum = [*False*, *True*]

instance ⟨*proof*⟩

end

primrec *product* :: *'a list* ⇒ *'b list* ⇒ (*'a* × *'b*) *list* **where**

product [] - = []
| *product* (*x*#*xs*) *ys* = *map* (*Pair x*) *ys* @ *product xs ys*

lemma *product-list-set*:
set (*product xs ys*) = *set xs* × *set ys*
⟨*proof*⟩

lemma *distinct-product*:
assumes *distinct xs* **and** *distinct ys*
shows *distinct* (*product xs ys*)
⟨*proof*⟩

instantiation ∗ :: (*enum*, *enum*) *enum*
begin

definition
enum = *product enum enum*

instance ⟨*proof*⟩

end

instantiation + :: (*enum*, *enum*) *enum*
begin

definition
enum = *map Inl enum* @ *map Inr enum*

instance ⟨*proof*⟩

end

primrec *sublists* :: *'a list* ⇒ *'a list list* **where**
sublists [] = [[]]
| *sublists* (*x*#*xs*) = (*let xss* = *sublists xs in map* (*Cons x*) *xss* @ *xss*)

lemma *length-sublists*:
length (*sublists xs*) = *Suc* (*Suc* (*0*::*nat*)) ^ *length xs*
⟨*proof*⟩

lemma *sublists-powset*:
set ‘ *set* (*sublists xs*) = *Pow* (*set xs*)
⟨*proof*⟩

lemma *distinct-set-sublists*:
assumes *distinct xs*
shows *distinct* (*map set* (*sublists xs*))
⟨*proof*⟩

instantiation *nibble* :: *enum*
begin

definition
enum = [*Nibble0*, *Nibble1*, *Nibble2*, *Nibble3*, *Nibble4*, *Nibble5*, *Nibble6*, *Nibble7*, *Nibble8*, *Nibble9*, *NibbleA*, *NibbleB*, *NibbleC*, *NibbleD*, *NibbleE*, *NibbleF*]

instance ⟨*proof*⟩

end

instantiation *char* :: *enum*
begin

definition
[*code del*]: *enum* = *map* (*split Char*) (*product enum enum*)

lemma *enum-chars* [*code*]:
enum = *chars*
⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *option* :: (*enum*) *enum*
begin

definition
enum = *None* # *map Some enum*

instance ⟨*proof*⟩

end

end

35 Eval-Witness: Evaluation Oracle with ML witnesses

theory *Eval-Witness*
imports *List Main*
begin

We provide an oracle method similar to "eval", but with the possibility to provide ML values as witnesses for existential statements.

Our oracle can prove statements of the form $\exists\, x.\ P\ x$ where P is an

executable predicate that can be compiled to ML. The oracle generates code for P and applies it to a user-specified ML value. If the evaluation returns true, this is effectively a proof of $\exists\, x.\ P\ x$.

However, this is only sound if for every ML value of the given type there exists a corresponding HOL value, which could be used in an explicit proof. Unfortunately this is not true for function types, since ML functions are not equivalent to the pure HOL functions. Thus, the oracle can only be used on first-order types.

We define a type class to mark types that can be safely used with the oracle.

class *ml-equiv*

Instances of *ml-equiv* should only be declared for those types, where the universe of ML values coincides with the HOL values.

Since this is essentially a statement about ML, there is no logical characterization.

instance *nat* :: *ml-equiv* ⟨*proof*⟩
instance *bool* :: *ml-equiv* ⟨*proof*⟩
instance *list* :: (*ml-equiv*) *ml-equiv* ⟨*proof*⟩

⟨*ML*⟩

35.1 Toy Examples

Note that we must use the generated data structure for the naturals, since ML integers are different.

Since polymorphism is not allowed, we must specify the type explicitly:

lemma ∃ *l*. *length* (*l*::*bool list*) = *3*
⟨*proof*⟩

Multiple witnesses

lemma ∃ *k l*. *length* (*k*::*bool list*) = *length* (*l*::*bool list*)
⟨*proof*⟩

35.2 Discussion

35.2.1 Conflicts

This theory conflicts with EfficientNat, since the *ml-equiv* instance for natural numbers is not valid when they are mapped to ML integers. With that theory loaded, we could use our oracle to prove ∃ *n*. *n* < (*0*::'*a*) by providing ~*1* as a witness.

This shows that *ml-equiv* declarations have to be used with care, taking the configuration of the code generator into account.

35.2.2 Haskell

If we were able to run generated Haskell code, the situation would be much nicer, since Haskell functions are pure and could be used as witnesses for HOL functions: Although Haskell functions are partial, we know that if the evaluation terminates, they are "sufficiently defined" and could be completed arbitrarily to a total (HOL) function.

This would allow us to provide access to very efficient data structures via lookup functions coded in Haskell and provided to HOL as witnesses.

end

36 Executable-Set: A crude implementation of finite sets by lists – avoid using this at any cost!

theory *Executable-Set*
imports *More-Set*
begin

declare *mem-def* [*code del*]
declare *Collect-def* [*code del*]
declare *insert-code* [*code del*]
declare *vimage-code* [*code del*]

36.1 Set representation

⟨*ML*⟩

definition *Set* :: *'a list* ⇒ *'a set* **where**
[*simp*]: *Set* = *set*

definition *Coset* :: *'a list* ⇒ *'a set* **where**
[*simp*]: *Coset xs* = − *set xs*

⟨*ML*⟩

code-datatype *Set Coset*

consts-code
Coset (⟨**module**⟩*Coset*)
Set (⟨**module**⟩*Set*)
attach ⟪
datatype 'a set = *Set of 'a list* | *Coset of 'a list*;
⟫ — This assumes that there won't be a *Coset* without a *Set*

36.2 Basic operations

lemma [*code*]:
set xs = *Set* (*remdups xs*)

⟨*proof*⟩

lemma [*code*]:
x ∈ *Set xs* ⟷ *member xs x*
x ∈ *Coset xs* ⟷ ¬ *member xs x*
⟨*proof*⟩

definition *is-empty* :: *'a set* ⇒ *bool* **where**
[*simp*]: *is-empty A* ⟷ *A* = {}

lemma [*code-unfold*]:
A = {} ⟷ *is-empty A*
⟨*proof*⟩

definition *empty* :: *'a set* **where**
[*simp*]: *empty* = {}

lemma [*code-unfold*]:
{} = *empty*
⟨*proof*⟩

lemma [*code-unfold*, *code-inline del*]:
empty = *Set* []
⟨*proof*⟩

⟨*ML*⟩

lemma *is-empty-Set* [*code*]:
is-empty (*Set xs*) ⟷ *null xs*
⟨*proof*⟩

lemma *empty-Set* [*code*]:
empty = *Set* []
⟨*proof*⟩

lemma *insert-Set* [*code*]:
insert x (*Set xs*) = *Set* (*List.insert x xs*)
insert x (*Coset xs*) = *Coset* (*removeAll x xs*)
⟨*proof*⟩

lemma *remove-Set* [*code*]:
remove x (*Set xs*) = *Set* (*removeAll x xs*)
remove x (*Coset xs*) = *Coset* (*List.insert x xs*)
⟨*proof*⟩

lemma *image-Set* [*code*]:
image f (*Set xs*) = *Set* (*remdups* (*map f xs*))
⟨*proof*⟩

lemma *project-Set* [*code*]:
project P (*Set xs*) = *Set* (*filter P xs*)
⟨*proof*⟩

lemma *Ball-Set* [*code*]:
Ball (*Set xs*) *P* ⟷ *list-all P xs*
⟨*proof*⟩

lemma *Bex-Set* [*code*]:
Bex (*Set xs*) *P* ⟷ *list-ex P xs*
⟨*proof*⟩

lemma *card-Set* [*code*]:
card (*Set xs*) = *length* (*remdups xs*)
⟨*proof*⟩

36.3 Derived operations

definition *set-eq* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
[*simp*]: *set-eq* = *op* =

lemma [*code-unfold*]:
op = = *set-eq*
⟨*proof*⟩

definition *subset-eq* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
[*simp*]: *subset-eq* = *op* ⊆

lemma [*code-unfold*]:
op ⊆ = *subset-eq*
⟨*proof*⟩

definition *subset* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
[*simp*]: *subset* = *op* ⊂

lemma [*code-unfold*]:
op ⊂ = *subset*
⟨*proof*⟩

⟨*ML*⟩

lemma *set-eq-subset-eq* [*code*]:
set-eq A B ⟷ *subset-eq A B* ∧ *subset-eq B A*
⟨*proof*⟩

lemma *subset-eq-forall* [*code*]:
subset-eq A B ⟷ (∀ *x*∈*A*. *x* ∈ *B*)
⟨*proof*⟩

lemma *subset-subset-eq* [*code*]:
subset A B ⟷ *subset-eq A B* ∧ ¬ *subset-eq B A*
⟨*proof*⟩

36.4 Functorial operations

definition *inter* :: *'a set* ⇒ *'a set* ⇒ *'a set* **where**
[*simp*]: *inter* = *op* ∩

lemma [*code-unfold*]:
op ∩ = *inter*
⟨*proof*⟩

definition *subtract* :: *'a set* ⇒ *'a set* ⇒ *'a set* **where**
[*simp*]: *subtract A B* = *B* − *A*

lemma [*code-unfold*]:
B − *A* = *subtract A B*
⟨*proof*⟩

definition *union* :: *'a set* ⇒ *'a set* ⇒ *'a set* **where**
[*simp*]: *union* = *op* ∪

lemma [*code-unfold*]:
op ∪ = *union*
⟨*proof*⟩

definition *Inf* :: *'a*::*complete-lattice set* ⇒ *'a* **where**
[*simp*]: *Inf* = *Complete-Lattice.Inf*

lemma [*code-unfold*]:
Complete-Lattice.Inf = *Inf*
⟨*proof*⟩

definition *Sup* :: *'a*::*complete-lattice set* ⇒ *'a* **where**
[*simp*]: *Sup* = *Complete-Lattice.Sup*

lemma [*code-unfold*]:
Complete-Lattice.Sup = *Sup*
⟨*proof*⟩

definition *Inter* :: *'a set set* ⇒ *'a set* **where**
[*simp*]: *Inter* = *Inf*

lemma [*code-unfold*]:
Inf = *Inter*
⟨*proof*⟩

definition *Union* :: *'a set set* ⇒ *'a set* **where**

[*simp*]: *Union* = *Sup*

lemma [*code-unfold*]:
Sup = *Union*
⟨*proof*⟩

⟨*ML*⟩

lemma *inter-project* [*code*]:
inter A (*Set xs*) = *Set* (*List.filter* (λ*x*. *x* ∈ *A*) *xs*)
inter A (*Coset xs*) = *foldr remove xs A*
⟨*proof*⟩

lemma *subtract-remove* [*code*]:
subtract (*Set xs*) *A* = *foldr remove xs A*
subtract (*Coset xs*) *A* = *Set* (*List.filter* (λ*x*. *x* ∈ *A*) *xs*)
⟨*proof*⟩

lemma *union-insert* [*code*]:
union (*Set xs*) *A* = *foldr insert xs A*
union (*Coset xs*) *A* = *Coset* (*List.filter* (λ*x*. *x* ∉ *A*) *xs*)
⟨*proof*⟩

lemma *Inf-inf* [*code*]:
Inf (*Set xs*) = *foldr inf xs* (*top* :: *'a*::*complete-lattice*)
Inf (*Coset* []) = (*bot* :: *'a*::*complete-lattice*)
⟨*proof*⟩

lemma *Sup-sup* [*code*]:
Sup (*Set xs*) = *foldr sup xs* (*bot* :: *'a*::*complete-lattice*)
Sup (*Coset* []) = (*top* :: *'a*::*complete-lattice*)
⟨*proof*⟩

lemma *Inter-inter* [*code*]:
Inter (*Set xs*) = *foldr inter xs* (*Coset* [])
Inter (*Coset* []) = *empty*
⟨*proof*⟩

lemma *Union-union* [*code*]:
Union (*Set xs*) = *foldr union xs empty*
Union (*Coset* []) = *Coset* []
⟨*proof*⟩

hide-const (**open**) *is-empty empty remove*
set-eq subset-eq subset inter union subtract Inf Sup Inter Union

end

37 Lattice-Algebras: Various algebraic structures combined with a lattice

theory *Lattice-Algebras*
imports *Complex-Main*
begin

class *semilattice-inf-ab-group-add* = *ordered-ab-group-add* + *semilattice-inf*
begin

lemma *add-inf-distrib-left*:
  $a + inf\ b\ c = inf\ (a + b)\ (a + c)$
⟨*proof*⟩

lemma *add-inf-distrib-right*:
  $inf\ a\ b + c = inf\ (a + c)\ (b + c)$
⟨*proof*⟩

end

class *semilattice-sup-ab-group-add* = *ordered-ab-group-add* + *semilattice-sup*
begin

lemma *add-sup-distrib-left*:
  $a + sup\ b\ c = sup\ (a + b)\ (a + c)$
⟨*proof*⟩

lemma *add-sup-distrib-right*:
  $sup\ a\ b + c = sup\ (a+c)\ (b+c)$
⟨*proof*⟩

end

class *lattice-ab-group-add* = *ordered-ab-group-add* + *lattice*
begin

subclass *semilattice-inf-ab-group-add* ⟨*proof*⟩
subclass *semilattice-sup-ab-group-add* ⟨*proof*⟩

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

lemma *inf-eq-neg-sup*: $inf\ a\ b = -\ sup\ (-a)\ (-b)$
⟨*proof*⟩

lemma *sup-eq-neg-inf*: $sup\ a\ b = -\ inf\ (-a)\ (-b)$
⟨*proof*⟩

lemma *neg-inf-eq-sup*: $-\ inf\ a\ b = sup\ (-a)\ (-b)$

⟨*proof*⟩

lemma *neg-sup-eq-inf*: $- \; sup \; a \; b = inf \; (-a) \; (-b)$
⟨*proof*⟩

lemma *add-eq-inf-sup*: $a + b = sup \; a \; b + inf \; a \; b$
⟨*proof*⟩

37.1 Positive Part, Negative Part, Absolute Value

definition
$nprt :: 'a \Rightarrow 'a$ **where**
$nprt \; x = inf \; x \; 0$

definition
$pprt :: 'a \Rightarrow 'a$ **where**
$pprt \; x = sup \; x \; 0$

lemma *pprt-neg*: $pprt \; (- \; x) = - \; nprt \; x$
⟨*proof*⟩

lemma *nprt-neg*: $nprt \; (- \; x) = - \; pprt \; x$
⟨*proof*⟩

lemma *prts*: $a = pprt \; a + nprt \; a$
⟨*proof*⟩

lemma *zero-le-pprt*[*simp*]: $0 \leq pprt \; a$
⟨*proof*⟩

lemma *nprt-le-zero*[*simp*]: $nprt \; a \leq 0$
⟨*proof*⟩

lemma *le-eq-neg*: $a \leq - \; b \longleftrightarrow a + b \leq 0$ (**is** $?l = ?r$)
⟨*proof*⟩

lemma *pprt-0*[*simp*]: $pprt \; 0 = 0$ ⟨*proof*⟩
lemma *nprt-0*[*simp*]: $nprt \; 0 = 0$ ⟨*proof*⟩

lemma *pprt-eq-id* [*simp*, *no-atp*]: $0 \leq x \Longrightarrow pprt \; x = x$
⟨*proof*⟩

lemma *nprt-eq-id* [*simp*, *no-atp*]: $x \leq 0 \Longrightarrow nprt \; x = x$
⟨*proof*⟩

lemma *pprt-eq-0* [*simp*, *no-atp*]: $x \leq 0 \Longrightarrow pprt \; x = 0$
⟨*proof*⟩

lemma *nprt-eq-0* [*simp*, *no-atp*]: $0 \leq x \Longrightarrow nprt \; x = 0$

⟨*proof*⟩

lemma *sup-0-imp-0*: $sup\ a\ (-\ a) = 0 \Longrightarrow a = 0$
⟨*proof*⟩

lemma *inf-0-imp-0*: $inf\ a\ (-a) = 0 \Longrightarrow a = 0$
⟨*proof*⟩

lemma *inf-0-eq-0* [*simp*, *no-atp*]: $inf\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *sup-0-eq-0* [*simp*, *no-atp*]: $sup\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]:
$0 \leq a + a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *double-zero* [*simp*]:
$a + a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *zero-less-double-add-iff-zero-less-single-add* [*simp*]:
$0 < a + a \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]:
$a + a \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]:
$a + a < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

declare *neg-inf-eq-sup* [*simp*] *neg-sup-eq-inf* [*simp*]

lemma *le-minus-self-iff*: $a \leq -\ a \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *minus-le-self-iff*: $-\ a \leq a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow nprt\ a = 0$
⟨*proof*⟩

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow pprt\ a = 0$
⟨*proof*⟩

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow pprt\ a = a$

⟨*proof*⟩

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow nprt\ a = a$
⟨*proof*⟩

lemma *pprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow pprt\ a \leq pprt\ b$
⟨*proof*⟩

lemma *nprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow nprt\ a \leq nprt\ b$
⟨*proof*⟩

end

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

class *lattice-ab-group-add-abs* = *lattice-ab-group-add* + *abs* +
  assumes *abs-lattice*: $|a| = sup\ a\ (-\ a)$
begin

lemma *abs-prts*: $|a| = pprt\ a - nprt\ a$
⟨*proof*⟩

subclass *ordered-ab-group-add-abs*
⟨*proof*⟩

end

lemma *sup-eq-if*:
  fixes $a :: 'a::\{lattice\text{-}ab\text{-}group\text{-}add, linorder\}$
  shows $sup\ a\ (-\ a) = (if\ a < 0\ then\ -\ a\ else\ a)$
⟨*proof*⟩

lemma *abs-if-lattice*:
  fixes $a :: 'a::\{lattice\text{-}ab\text{-}group\text{-}add\text{-}abs, linorder\}$
  shows $|a| = (if\ a < 0\ then\ -\ a\ else\ a)$
⟨*proof*⟩

lemma *estimate-by-abs*:
  $a + b <= (c::'a::lattice\text{-}ab\text{-}group\text{-}add\text{-}abs) \Longrightarrow a <= c + abs\ b$
⟨*proof*⟩

class *lattice-ring* = *ordered-ring* + *lattice-ab-group-add-abs*
begin

subclass *semilattice-inf-ab-group-add* ⟨*proof*⟩
subclass *semilattice-sup-ab-group-add* ⟨*proof*⟩

end

lemma *abs-le-mult*: *abs* (*a* ∗ *b*) ≤ (*abs a*) ∗ (*abs* (*b*::′*a*::*lattice-ring*))
⟨*proof*⟩

instance *lattice-ring* ⊆ *ordered-ring-abs*
⟨*proof*⟩

lemma *mult-le-prts*:
assumes
a1 <= (*a*::′*a*::*lattice-ring*)
a <= *a2*
b1 <= *b*
b <= *b2*
shows
a ∗ *b* <= *pprt a2* ∗ *pprt b2* + *pprt a1* ∗ *nprt b2* + *nprt a2* ∗ *pprt b1* + *nprt a1* ∗ *nprt b1*
⟨*proof*⟩

lemma *mult-ge-prts*:
assumes
a1 <= (*a*::′*a*::*lattice-ring*)
a <= *a2*
b1 <= *b*
b <= *b2*
shows
a ∗ *b* >= *nprt a1* ∗ *pprt b2* + *nprt a2* ∗ *nprt b2* + *pprt a1* ∗ *pprt b1* + *pprt a2* ∗ *nprt b1*
⟨*proof*⟩

instance *int* :: *lattice-ring*
⟨*proof*⟩

instance *real* :: *lattice-ring*
⟨*proof*⟩

end

38 Float: Floating-Point Numbers

theory *Float*
imports *Complex-Main Lattice-Algebras*
begin

definition
pow2 :: *int* ⇒ *real* **where**
[*simp*]: *pow2 a* = (*if* (*0* <= *a*) *then* (*2*^(*nat a*)) *else* (*inverse* (*2*^(*nat* (−*a*)))))

datatype *float* = *Float int int*

primrec *of-float* :: *float* ⇒ *real* **where**
of-float (*Float a b*) = *real a* ∗ *pow2 b*

defs (**overloaded**)
real-of-float-def [*code-unfold*]: *real* == *of-float*

primrec *mantissa* :: *float* ⇒ *int* **where**
mantissa (*Float a b*) = *a*

primrec *scale* :: *float* ⇒ *int* **where**
scale (*Float a b*) = *b*

instantiation *float* :: *zero* **begin**
definition *zero-float* **where** *0* = *Float 0 0*
instance ⟨*proof*⟩
end

instantiation *float* :: *one* **begin**
definition *one-float* **where** *1* = *Float 1 0*
instance ⟨*proof*⟩
end

instantiation *float* :: *number* **begin**
definition *number-of-float* **where** *number-of n* = *Float n 0*
instance ⟨*proof*⟩
end

lemma *number-of-float-Float* [*code-unfold-post*]:
number-of k = *Float* (*number-of k*) *0*
⟨*proof*⟩

lemma *real-of-float-simp*[*simp*]: *real* (*Float a b*) = *real a* ∗ *pow2 b*
⟨*proof*⟩

lemma *real-of-float-neg-exp*: *e* < *0* ⟹ *real* (*Float m e*) = *real m* ∗ *inverse* (*2^nat* (−*e*)) ⟨*proof*⟩
lemma *real-of-float-nge0-exp*: ¬ *0* ≤ *e* ⟹ *real* (*Float m e*) = *real m* ∗ *inverse* (*2^nat* (−*e*)) ⟨*proof*⟩
lemma *real-of-float-ge0-exp*: *0* ≤ *e* ⟹ *real* (*Float m e*) = *real m* ∗ (*2^nat e*) ⟨*proof*⟩

lemma *Float-num*[*simp*]: **shows**
real (*Float 1 0*) = *1* **and** *real* (*Float 1 1*) = *2* **and** *real* (*Float 1 2*) = *4* **and**
real (*Float 1 −1*) = *1/2* **and** *real* (*Float 1 −2*) = *1/4* **and** *real* (*Float 1 −3*) = *1/8* **and**
real (*Float −1 0*) = *−1* **and** *real* (*Float* (*number-of n*) *0*) = *number-of n*
⟨*proof*⟩

lemma *float-number-of*[*simp*]: *real* (*number-of x* :: *float*) = *number-of x*
⟨*proof*⟩

lemma *float-number-of-int*[*simp*]: *real* (*Float n 0*) = *real n*
⟨*proof*⟩

lemma *pow2-0*[*simp*]: *pow2 0* = *1* ⟨*proof*⟩
lemma *pow2-1*[*simp*]: *pow2 1* = *2* ⟨*proof*⟩
lemma *pow2-neg*: *pow2 x* = *inverse* (*pow2* (−*x*)) ⟨*proof*⟩

declare *pow2-def*[*simp del*]

lemma *pow2-add1*: *pow2* (*1* + *a*) = *2* ∗ (*pow2 a*)
⟨*proof*⟩

lemma *pow2-add*: *pow2* (*a*+*b*) = (*pow2 a*) ∗ (*pow2 b*)
⟨*proof*⟩

lemma *float-components*[*simp*]: *Float* (*mantissa f*) (*scale f*) = *f* ⟨*proof*⟩

lemma *float-split*: ∃ *a b*. *x* = *Float a b* ⟨*proof*⟩

lemma *float-split2*: (∀ *a b*. *x* ≠ *Float a b*) = *False* ⟨*proof*⟩

lemma *float-zero*[*simp*]: *real* (*Float 0 e*) = *0* ⟨*proof*⟩

lemma *abs-div-2-less*: *a* ≠ *0* ⟹ *a* ≠ −*1* ⟹ *abs*((*a*::*int*) *div 2*) < *abs a*
⟨*proof*⟩

function *normfloat* :: *float* ⇒ *float* **where**
normfloat (*Float a b*) = (*if a* ≠ *0* ∧ *even a then normfloat* (*Float* (*a div 2*) (*b*+*1*))
else if a=*0 then Float 0 0 else Float a b*)
⟨*proof*⟩
termination ⟨*proof*⟩
declare *normfloat.simps*[*simp del*]

theorem *normfloat*[*symmetric*, *simp*]: *real f* = *real* (*normfloat f*)
⟨*proof*⟩

lemma *pow2-neq-zero*[*simp*]: *pow2 x* ≠ *0*
⟨*proof*⟩

lemma *pow2-int*: *pow2* (*int c*) = *2^c*
⟨*proof*⟩

lemma *zero-less-pow2*[*simp*]:
0 < *pow2 x*
⟨*proof*⟩

lemma *normfloat-imp-odd-or-zero*: *normfloat f = Float a b ⟹ odd a ∨ (a = 0 ∧ b = 0)*
⟨*proof*⟩

lemma *float-eq-odd-helper*:
assumes *odd*: *odd a′*
and *floateq*: *real (Float a b) = real (Float a′ b′)*
shows *b ≤ b′*
⟨*proof*⟩

lemma *float-eq-odd*:
assumes *odd1*: *odd a*
and *odd2*: *odd a′*
and *floateq*: *real (Float a b) = real (Float a′ b′)*
shows *a = a′ ∧ b = b′*
⟨*proof*⟩

theorem *normfloat-unique*:
assumes *real-of-float-eq*: *real f = real g*
shows *normfloat f = normfloat g*
⟨*proof*⟩

instantiation *float* :: *plus* **begin**
fun *plus-float* **where**
[*simp del*]: *(Float a-m a-e) + (Float b-m b-e) =*
*(if a-e ≤ b-e then Float (a-m + b-m * 2^(nat(b-e − a-e))) a-e*
*else Float (a-m * 2^(nat (a-e − b-e)) + b-m) b-e)*
instance ⟨*proof*⟩
end

instantiation *float* :: *uminus* **begin**
primrec *uminus-float* **where** [*simp del*]: *uminus-float (Float m e) = Float (−m) e*
instance ⟨*proof*⟩
end

instantiation *float* :: *minus* **begin**
definition *minus-float* **where** [*simp del*]: *(z::float) − w = z + (− w)*
instance ⟨*proof*⟩
end

instantiation *float* :: *times* **begin**
fun *times-float* **where** [*simp del*]: *(Float a-m a-e) * (Float b-m b-e) = Float (a-m * b-m) (a-e + b-e)*
instance ⟨*proof*⟩
end

primrec *float-pprt* :: *float ⇒ float* **where**

float-pprt (*Float a e*) = (*if 0 <= a then* (*Float a e*) *else 0*)

primrec *float-nprt* :: *float* ⇒ *float* **where**
float-nprt (*Float a e*) = (*if 0 <= a then 0 else* (*Float a e*))

instantiation *float* :: *ord* **begin**
definition *le-float-def*: $z \leq (w :: float) \equiv real\ z \leq real\ w$
definition *less-float-def*: $z < (w :: float) \equiv real\ z < real\ w$
instance ⟨*proof*⟩
end

lemma *real-of-float-add*[*simp*]: *real* (*a* + *b*) = *real a* + *real* (*b* :: *float*)
⟨*proof*⟩

lemma *real-of-float-minus*[*simp*]: *real* (− *a*) = − *real* (*a* :: *float*)
⟨*proof*⟩

lemma *real-of-float-sub*[*simp*]: *real* (*a* − *b*) = *real a* − *real* (*b* :: *float*)
⟨*proof*⟩

lemma *real-of-float-mult*[*simp*]: *real* (*a*∗*b*) = *real a* ∗ *real* (*b* :: *float*)
⟨*proof*⟩

lemma *real-of-float-0*[*simp*]: *real* (*0* :: *float*) = *0*
⟨*proof*⟩

lemma *real-of-float-1*[*simp*]: *real* (*1* :: *float*) = *1*
⟨*proof*⟩

lemma *zero-le-float*:
(*0 <= real* (*Float a b*)) = (*0 <= a*)
⟨*proof*⟩

lemma *float-le-zero*:
(*real* (*Float a b*) *<= 0*) = (*a <= 0*)
⟨*proof*⟩

declare *real-of-float-simp*[*simp del*]

lemma *real-of-float-pprt*[*simp*]: *real* (*float-pprt a*) = *pprt* (*real a*)
⟨*proof*⟩

lemma *real-of-float-nprt*[*simp*]: *real* (*float-nprt a*) = *nprt* (*real a*)
⟨*proof*⟩

instance *float* :: *ab-semigroup-add*
⟨*proof*⟩

instance *float* :: *comm-monoid-mult*

⟨*proof*⟩

lemma *0 + Float 0 1 = 0 + Float 0 2*
⟨*proof*⟩

instance *float* :: *comm-semiring*
⟨*proof*⟩

instance *float* :: *zero-neq-one*
⟨*proof*⟩

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

lemma *float-less-simp*: $((x::float) < y) = (0 < y - x)$
⟨*proof*⟩

lemma *real-of-float-min*: $real\ (min\ x\ y :: float) = min\ (real\ x)\ (real\ y)$ ⟨*proof*⟩
lemma *real-of-float-max*: $real\ (max\ a\ b :: float) = max\ (real\ a)\ (real\ b)$ ⟨*proof*⟩

lemma *float-power*: $real\ (x \hat{}\ n :: float) = real\ x \hat{}\ n$
⟨*proof*⟩

lemma *zero-le-pow2*[*simp*]: $0 \leq pow2\ s$
⟨*proof*⟩

lemma *pow2-less-0-eq-False*[*simp*]: $(pow2\ s < 0) = False$
⟨*proof*⟩

lemma *pow2-le-0-eq-False*[*simp*]: $(pow2\ s \leq 0) = False$
⟨*proof*⟩

lemma *float-pos-m-pos*: $0 < Float\ m\ e \Longrightarrow 0 < m$
⟨*proof*⟩

lemma *float-pos-less1-e-neg*: **assumes** $0 < Float\ m\ e$ **and** $Float\ m\ e < 1$ **shows** $e < 0$
⟨*proof*⟩

lemma *float-less1-mantissa-bound*: **assumes** $0 < Float\ m\ e\ Float\ m\ e < 1$ **shows** $m < 2\hat{}(nat\ (-e))$
⟨*proof*⟩

function *bitlen* :: *int* ⇒ *int* **where**
bitlen 0 = 0 |
bitlen −1 = 1 |

$0 < x \Longrightarrow bitlen\ x = 1 + (bitlen\ (x\ div\ 2))$ |
$x < -1 \Longrightarrow bitlen\ x = 1 + (bitlen\ (x\ div\ 2))$
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *bitlen-ge0*: $0 \leq bitlen\ x$ ⟨*proof*⟩
lemma *bitlen-ge1*: $x \neq 0 \Longrightarrow 1 \leq bitlen\ x$ ⟨*proof*⟩

lemma *bitlen-bounds'*: **assumes** $0 < x$ **shows** $2\hat{}nat\ (bitlen\ x - 1) \leq x \wedge x + 1 \leq 2\hat{}nat\ (bitlen\ x)$ (**is** $?P\ x$)
⟨*proof*⟩

lemma *bitlen-bounds*: **assumes** $0 < x$ **shows** $2\hat{}nat\ (bitlen\ x - 1) \leq x \wedge x < 2\hat{}nat\ (bitlen\ x)$
⟨*proof*⟩

lemma *bitlen-div*: **assumes** $0 < m$ **shows** $1 \leq real\ m\ /\ 2\hat{}nat\ (bitlen\ m - 1)$ **and** $real\ m\ /\ 2\hat{}nat\ (bitlen\ m - 1) < 2$
⟨*proof*⟩

lemma *float-gt1-scale*: **assumes** $1 \leq Float\ m\ e$
shows $0 \leq e + (bitlen\ m - 1)$
⟨*proof*⟩

lemma *normalized-float*: **assumes** $m \neq 0$ **shows** $real\ (Float\ m\ (-\ (bitlen\ m - 1))) = real\ m\ /\ 2\hat{}nat\ (bitlen\ m - 1)$
⟨*proof*⟩

lemma *bitlen-Pls*: $bitlen\ (Int.Pls) = Int.Pls$ ⟨*proof*⟩

lemma *bitlen-Min*: $bitlen\ (Int.Min) = Int.Bit1\ Int.Pls$ ⟨*proof*⟩

lemma *bitlen-B0*: $bitlen\ (Int.Bit0\ b) = (if\ iszero\ b\ then\ Int.Pls\ else\ Int.succ\ (bitlen\ b))$
⟨*proof*⟩

lemma *bitlen-B1*: $bitlen\ (Int.Bit1\ b) = (if\ iszero\ (Int.succ\ b)\ then\ Int.Bit1\ Int.Pls\ else\ Int.succ\ (bitlen\ b))$
⟨*proof*⟩

lemma *bitlen-number-of*: $bitlen\ (number\text{-}of\ w) = number\text{-}of\ (bitlen\ w)$
⟨*proof*⟩

lemma [*code*]: $bitlen\ x =$
$(if\ x = 0\ \ then\ 0$
$\ else\ if\ x = -1\ then\ 1$
$\ \ \ \ \ \ \ \ else\ (1 + (bitlen\ (x\ div\ 2))))$
⟨*proof*⟩

definition *lapprox-posrat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
lapprox-posrat prec x y =
(*let*
l = *nat* (*int prec* + *bitlen y* − *bitlen x*) ;
d = (*x* ∗ *2^l*) *div y*
in normfloat (*Float d* (− (*int l*))))

lemma *pow2-minus*: *pow2* (−*x*) = *inverse* (*pow2 x*)
⟨*proof*⟩

lemma *lapprox-posrat*:
assumes *x*: *0* ≤ *x*
and *y*: *0* < *y*
shows *real* (*lapprox-posrat prec x y*) ≤ *real x* / *real y*
⟨*proof*⟩

lemma *real-of-int-div-mult*:
fixes *x y c* :: *int* **assumes** *0* < *y* **and** *0* < *c*
shows *real* (*x div y*) ≤ *real* (*x* ∗ *c div y*) ∗ *inverse* (*real c*)
⟨*proof*⟩

lemma *lapprox-posrat-bottom*: **assumes** *0* < *y*
shows *real* (*x div y*) ≤ *real* (*lapprox-posrat n x y*)
⟨*proof*⟩

lemma *lapprox-posrat-nonneg*: **assumes** *0* ≤ *x* **and** *0* < *y*
shows *0* ≤ *real* (*lapprox-posrat n x y*)
⟨*proof*⟩

definition *rapprox-posrat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
rapprox-posrat prec x y = (*let*
l = *nat* (*int prec* + *bitlen y* − *bitlen x*) ;
X = *x* ∗ *2^l* ;
d = *X div y* ;
m = *X mod y*
in normfloat (*Float* (*d* + (*if m* = *0 then 0 else 1*)) (− (*int l*))))

lemma *rapprox-posrat*:
assumes *x*: *0* ≤ *x*
and *y*: *0* < *y*
shows *real x* / *real y* ≤ *real* (*rapprox-posrat prec x y*)
⟨*proof*⟩

lemma *rapprox-posrat-le1*: **assumes** *0* ≤ *x* **and** *0* < *y* **and** *x* ≤ *y*
shows *real* (*rapprox-posrat n x y*) ≤ *1*
⟨*proof*⟩

lemma *zdiv-greater-zero*: **fixes** $a\ b :: int$ **assumes** $0 < a$ **and** $a \leq b$
shows $0 < b\ div\ a$
⟨*proof*⟩

lemma *rapprox-posrat-less1*: **assumes** $0 \leq x$ **and** $0 < y$ **and** $2 * x < y$ **and** $0 < n$
shows $real\ (rapprox\text{-}posrat\ n\ x\ y) < 1$
⟨*proof*⟩

lemma *approx-rat-pattern*: **fixes** P **and** $ps :: nat * int * int$
assumes Y: $\bigwedge y\ prec\ x.\ [\![y = 0;\ ps = (prec,\ x,\ 0)]\!] \Longrightarrow P$
and A: $\bigwedge x\ y\ prec.\ [\![0 \leq x;\ 0 < y;\ ps = (prec,\ x,\ y)]\!] \Longrightarrow P$
and B: $\bigwedge x\ y\ prec.\ [\![x < 0;\ 0 < y;\ ps = (prec,\ x,\ y)]\!] \Longrightarrow P$
and C: $\bigwedge x\ y\ prec.\ [\![x < 0;\ y < 0;\ ps = (prec,\ x,\ y)]\!] \Longrightarrow P$
and D: $\bigwedge x\ y\ prec.\ [\![0 \leq x;\ y < 0;\ ps = (prec,\ x,\ y)]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

function *lapprox-rat* :: $nat \Rightarrow int \Rightarrow int \Rightarrow float$
where
$y = 0 \Longrightarrow lapprox\text{-}rat\ prec\ x\ y = 0$
| $0 \leq x \Longrightarrow 0 < y \Longrightarrow lapprox\text{-}rat\ prec\ x\ y = lapprox\text{-}posrat\ prec\ x\ y$
| $x < 0 \Longrightarrow 0 < y \Longrightarrow lapprox\text{-}rat\ prec\ x\ y = -\ (rapprox\text{-}posrat\ prec\ (-x)\ y)$
| $x < 0 \Longrightarrow y < 0 \Longrightarrow lapprox\text{-}rat\ prec\ x\ y = lapprox\text{-}posrat\ prec\ (-x)\ (-y)$
| $0 \leq x \Longrightarrow y < 0 \Longrightarrow lapprox\text{-}rat\ prec\ x\ y = -\ (rapprox\text{-}posrat\ prec\ x\ (-y))$
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *compute-lapprox-rat*[*code*]:
lapprox-rat prec x y = (if y = 0 then 0 else if 0 ≤ x then (if 0 < y then lapprox-posrat prec x y else − (rapprox-posrat prec x (−y))) else (if 0 < y then − (rapprox-posrat prec (−x) y) else lapprox-posrat prec (−x) (−y)))
⟨*proof*⟩

lemma *lapprox-rat*: $real\ (lapprox\text{-}rat\ prec\ x\ y) \leq real\ x\ /\ real\ y$
⟨*proof*⟩

lemma *lapprox-rat-bottom*: **assumes** $0 \leq x$ **and** $0 < y$
shows $real\ (x\ div\ y) \leq real\ (lapprox\text{-}rat\ n\ x\ y)$
⟨*proof*⟩

function *rapprox-rat* :: $nat \Rightarrow int \Rightarrow int \Rightarrow float$
where
$y = 0 \Longrightarrow rapprox\text{-}rat\ prec\ x\ y = 0$
| $0 \leq x \Longrightarrow 0 < y \Longrightarrow rapprox\text{-}rat\ prec\ x\ y = rapprox\text{-}posrat\ prec\ x\ y$
| $x < 0 \Longrightarrow 0 < y \Longrightarrow rapprox\text{-}rat\ prec\ x\ y = -\ (lapprox\text{-}posrat\ prec\ (-x)\ y)$
| $x < 0 \Longrightarrow y < 0 \Longrightarrow rapprox\text{-}rat\ prec\ x\ y = rapprox\text{-}posrat\ prec\ (-x)\ (-y)$
| $0 \leq x \Longrightarrow y < 0 \Longrightarrow rapprox\text{-}rat\ prec\ x\ y = -\ (lapprox\text{-}posrat\ prec\ x\ (-y))$

⟨*proof*⟩
termination ⟨*proof*⟩

lemma *compute-rapprox-rat*[*code*]:
rapprox-rat prec x y = (*if y* = *0 then 0 else if 0* ≤ *x then* (*if 0* < *y then rapprox-posrat prec x y else* − (*lapprox-posrat prec x* (−*y*))) *else* (*if 0* < *y then* − (*lapprox-posrat prec* (−*x*) *y*) *else rapprox-posrat prec* (−*x*) (−*y*)))
⟨*proof*⟩

lemma *rapprox-rat*: *real x* / *real y* ≤ *real* (*rapprox-rat prec x y*)
⟨*proof*⟩

lemma *rapprox-rat-le1*: **assumes** *0* ≤ *x* **and** *0* < *y* **and** *x* ≤ *y*
shows *real* (*rapprox-rat n x y*) ≤ *1*
⟨*proof*⟩

lemma *rapprox-rat-neg*: **assumes** *x* < *0* **and** *0* < *y*
shows *real* (*rapprox-rat n x y*) ≤ *0*
⟨*proof*⟩

lemma *rapprox-rat-nonneg-neg*: **assumes** *0* ≤ *x* **and** *y* < *0*
shows *real* (*rapprox-rat n x y*) ≤ *0*
⟨*proof*⟩

lemma *rapprox-rat-nonpos-pos*: **assumes** *x* ≤ *0* **and** *0* < *y*
shows *real* (*rapprox-rat n x y*) ≤ *0*
⟨*proof*⟩

fun *float-divl* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float*
where
float-divl prec (*Float m1 s1*) (*Float m2 s2*) =
(*let*
l = *lapprox-rat prec m1 m2*;
f = *Float 1* (*s1* − *s2*)
in
f ∗ *l*)

lemma *float-divl*: *real* (*float-divl prec x y*) ≤ *real x* / *real y*
⟨*proof*⟩

lemma *float-divl-lower-bound*: **assumes** *0* ≤ *x* **and** *0* < *y* **shows** *0* ≤ *float-divl prec x y*
⟨*proof*⟩

lemma *float-divl-pos-less1-bound*: **assumes** *0* < *x* **and** *x* < *1* **and** *0* < *prec*
shows *1* ≤ *float-divl prec 1 x*
⟨*proof*⟩

fun *float-divr* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float*
where
float-divr prec (*Float m1 s1*) (*Float m2 s2*) =
(*let*
r = *rapprox-rat prec m1 m2*;
f = *Float 1* (*s1* − *s2*)
in
f ∗ *r*)

lemma *float-divr*: *real x* / *real y* ≤ *real* (*float-divr prec x y*)
⟨*proof*⟩

lemma *float-divr-pos-less1-lower-bound*: **assumes** *0* < *x* **and** *x* < *1* **shows** *1* ≤ *float-divr prec 1 x*
⟨*proof*⟩

lemma *float-divr-nonpos-pos-upper-bound*: **assumes** *x* ≤ *0* **and** *0* < *y* **shows** *float-divr prec x y* ≤ *0*
⟨*proof*⟩

lemma *float-divr-nonneg-neg-upper-bound*: **assumes** *0* ≤ *x* **and** *y* < *0* **shows** *float-divr prec x y* ≤ *0*
⟨*proof*⟩

primrec *round-down* :: *nat* ⇒ *float* ⇒ *float* **where**
round-down prec (*Float m e*) = (*let d* = *bitlen m* − *int prec in*
if 0 < *d then let P* = *2^nat d* ; *n* = *m div P in Float n* (*e* + *d*)
else Float m e)

primrec *round-up* :: *nat* ⇒ *float* ⇒ *float* **where**
round-up prec (*Float m e*) = (*let d* = *bitlen m* − *int prec in*
if 0 < *d then let P* = *2^nat d* ; *n* = *m div P* ; *r* = *m mod P in Float* (*n* + (*if r* = *0 then 0 else 1*)) (*e* + *d*)
else Float m e)

lemma *round-up*: *real x* ≤ *real* (*round-up prec x*)
⟨*proof*⟩

lemma *round-down*: *real* (*round-down prec x*) ≤ *real x*
⟨*proof*⟩

definition *lb-mult* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **where**
lb-mult prec x y = (*case normfloat* (*x* ∗ *y*) *of Float m e* ⇒ *let*
l = *bitlen m* − *int prec*
in if l > *0 then Float* (*m div* (*2^nat l*)) (*e* + *l*)
else Float m e)

definition *ub-mult* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **where**
ub-mult prec x y = (*case normfloat* (*x* ∗ *y*) *of Float m e* ⇒ *let*

$l = bitlen\ m - int\ prec$
in if $l > 0$ *then Float* $(m\ div\ (2\hat{}nat\ l) + 1)\ (e + l)$
else Float $m\ e$)

lemma *lb-mult*: $real\ (lb\text{-}mult\ prec\ x\ y) \leq real\ (x * y)$
⟨*proof*⟩

lemma *ub-mult*: $real\ (x * y) \leq real\ (ub\text{-}mult\ prec\ x\ y)$
⟨*proof*⟩

primrec *float-abs* :: *float* ⇒ *float* **where**
float-abs (*Float* $m\ e$) = *Float* $|m|\ e$

instantiation *float* :: *abs* **begin**
definition *abs-float-def*: $|x| = float\text{-}abs\ x$
instance ⟨*proof*⟩
end

lemma *real-of-float-abs*: $real\ |x :: float| = |real\ x|$
⟨*proof*⟩

primrec *floor-fl* :: *float* ⇒ *float* **where**
floor-fl (*Float* $m\ e$) = (*if* $0 \leq e$ *then Float* $m\ e$
else Float $(m\ div\ (2\ \hat{}\ (nat\ (-e))))\ 0$)

lemma *floor-fl*: $real\ (floor\text{-}fl\ x) \leq real\ x$
⟨*proof*⟩

lemma *floor-pos-exp*: **assumes** *floor*: *Float* $m\ e = floor\text{-}fl\ x$ **shows** $0 \leq e$
⟨*proof*⟩

declare *floor-fl.simps*[*simp del*]

primrec *ceiling-fl* :: *float* ⇒ *float* **where**
ceiling-fl (*Float* $m\ e$) = (*if* $0 \leq e$ *then Float* $m\ e$
else Float $(m\ div\ (2\ \hat{}\ (nat\ (-e))) + 1)\ 0$)

lemma *ceiling-fl*: $real\ x \leq real\ (ceiling\text{-}fl\ x)$
⟨*proof*⟩

declare *ceiling-fl.simps*[*simp del*]

definition *lb-mod* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* ⇒ *float* **where**
$lb\text{-}mod\ prec\ x\ ub\ lb = x - ceiling\text{-}fl\ (float\text{-}divr\ prec\ x\ lb) * ub$

definition *ub-mod* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* ⇒ *float* **where**
$ub\text{-}mod\ prec\ x\ ub\ lb = x - floor\text{-}fl\ (float\text{-}divl\ prec\ x\ ub) * lb$

lemma *lb-mod*: **fixes** k :: *int* **assumes** $0 \leq real\ x$ **and** $real\ k * y \leq real\ x$ (**is** *?k*

$* y \leq ?x$)
assumes $0 < real\ lb\ real\ lb \leq y$ (**is** $?lb \leq y$) $y \leq real\ ub$ (**is** $y \leq ?ub$)
shows $real\ (lb\text{-}mod\ prec\ x\ ub\ lb) \leq ?x - ?k * y$
⟨*proof*⟩

lemma *ub-mod*: **fixes** $k :: int$ **and** $x :: float$ **assumes** $0 \leq real\ x$ **and** $real\ x \leq real\ k * y$ (**is** $?x \leq ?k * y$)
assumes $0 < real\ lb\ real\ lb \leq y$ (**is** $?lb \leq y$) $y \leq real\ ub$ (**is** $y \leq ?ub$)
shows $?x - ?k * y \leq real\ (ub\text{-}mod\ prec\ x\ ub\ lb)$
⟨*proof*⟩

lemma *le-float-def'*: $f \leq g = (case\ f - g\ of\ Float\ a\ b \Rightarrow a \leq 0)$
⟨*proof*⟩

lemma *float-less-zero*:
$(real\ (Float\ a\ b) < 0) = (a < 0)$
⟨*proof*⟩

lemma *less-float-def'*: $f < g = (case\ f - g\ of\ Float\ a\ b \Rightarrow a < 0)$
⟨*proof*⟩

end

39 Formal-Power-Series: A formalization of formal power series

theory *Formal-Power-Series*
imports *Complex-Main Binomial*
begin

39.1 The type of formal power series

typedef (**open**) $'a\ fps = \{f :: nat \Rightarrow 'a.\ True\}$
morphisms *fps-nth Abs-fps*
⟨*proof*⟩

notation *fps-nth* (**infixl** $ 75)

lemma *expand-fps-eq*: $p = q \longleftrightarrow (\forall n.\ p\ \$\ n = q\ \$\ n)$
⟨*proof*⟩

lemma *fps-ext*: $(\bigwedge n.\ p\ \$\ n = q\ \$\ n) \Longrightarrow p = q$
⟨*proof*⟩

lemma *fps-nth-Abs-fps* [*simp*]: $Abs\text{-}fps\ f\ \$\ n = f\ n$
⟨*proof*⟩

Definition of the basic elements 0 and 1 and the basic operations of

addition, negation and multiplication

instantiation *fps* :: (*zero*) *zero*
begin

definition *fps-zero-def*:
0 = Abs-fps (λn. 0)

instance ⟨*proof*⟩
end

lemma *fps-zero-nth* [*simp*]: *0 $ n = 0*
⟨*proof*⟩

instantiation *fps* :: ({*one*, *zero*}) *one*
begin

definition *fps-one-def*:
1 = Abs-fps (λn. if n = 0 then 1 else 0)

instance ⟨*proof*⟩
end

lemma *fps-one-nth* [*simp*]: *1 $ n = (if n = 0 then 1 else 0)*
⟨*proof*⟩

instantiation *fps* :: (*plus*) *plus*
begin

definition *fps-plus-def*:
op + = (λf g. Abs-fps (λn. f $ n + g $ n))

instance ⟨*proof*⟩
end

lemma *fps-add-nth* [*simp*]: *(f + g) $ n = f $ n + g $ n*
⟨*proof*⟩

instantiation *fps* :: (*minus*) *minus*
begin

definition *fps-minus-def*:
op − = (λf g. Abs-fps (λn. f $ n − g $ n))

instance ⟨*proof*⟩
end

lemma *fps-sub-nth* [*simp*]: *(f − g) $ n = f $ n − g $ n*
⟨*proof*⟩

instantiation *fps* :: (*uminus*) *uminus*
begin

definition *fps-uminus-def*:
uminus = (λ*f*. *Abs-fps* (λ*n*. − (*f* $ *n*)))

instance ⟨*proof*⟩
end

lemma *fps-neg-nth* [*simp*]: (− *f*) $ *n* = − (*f* $ *n*)
⟨*proof*⟩

instantiation *fps* :: ({*comm-monoid-add*, *times*}) *times*
begin

definition *fps-times-def*:
op ∗ = (λ*f g*. *Abs-fps* (λ*n*. ∑ *i*=*0*..*n*. *f* $ *i* ∗ *g* $ (*n* − *i*)))

instance ⟨*proof*⟩
end

lemma *fps-mult-nth*: (*f* ∗ *g*) $ *n* = (∑ *i*=*0*..*n*. *f*$*i* ∗ *g*$(*n* − *i*))
⟨*proof*⟩

declare *atLeastAtMost-iff*[*presburger*]
declare *Bex-def*[*presburger*]
declare *Ball-def*[*presburger*]

lemma *mult-delta-left*:
fixes *x y* :: *'a*::*mult-zero*
shows (*if b then x else 0*) ∗ *y* = (*if b then x* ∗ *y else 0*)
⟨*proof*⟩

lemma *mult-delta-right*:
fixes *x y* :: *'a*::*mult-zero*
shows *x* ∗ (*if b then y else 0*) = (*if b then x* ∗ *y else 0*)
⟨*proof*⟩

lemma *cond-value-iff*: *f* (*if b then x else y*) = (*if b then f x else f y*)
⟨*proof*⟩
lemma *cond-application-beta*: (*if b then f else g*) *x* = (*if b then f x else g x*)
⟨*proof*⟩

39.2 Formal power series form a commutative ring with unity, if the range of sequences they represent is a commutative ring with unity

instance *fps* :: (*semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance *fps* :: (*ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

lemma *fps-mult-assoc-lemma*:
fixes k :: *nat* **and** f :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a$::*comm-monoid-add*
shows $(\sum j{=}0..k.\ \sum i{=}0..j.\ f\ i\ (j - i)\ (n - j)) =$
$(\sum j{=}0..k.\ \sum i{=}0..k - j.\ f\ j\ i\ (n - j - i))$
⟨*proof*⟩

instance *fps* :: (*semiring-0*) *semigroup-mult*
⟨*proof*⟩

lemma *fps-mult-commute-lemma*:
fixes n :: *nat* **and** f :: $nat \Rightarrow nat \Rightarrow 'a$::*comm-monoid-add*
shows $(\sum i{=}0..n.\ f\ i\ (n - i)) = (\sum i{=}0..n.\ f\ (n - i)\ i)$
⟨*proof*⟩

instance *fps* :: (*comm-semiring-0*) *ab-semigroup-mult*
⟨*proof*⟩

instance *fps* :: (*monoid-add*) *monoid-add*
⟨*proof*⟩

instance *fps* :: (*comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *fps* :: (*semiring-1*) *monoid-mult*
⟨*proof*⟩

instance *fps* :: (*cancel-semigroup-add*) *cancel-semigroup-add*
⟨*proof*⟩

instance *fps* :: (*cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance *fps* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add* ⟨*proof*⟩

instance *fps* :: (*group-add*) *group-add*
⟨*proof*⟩

instance *fps* :: (*ab-group-add*) *ab-group-add*
⟨*proof*⟩

instance *fps* :: (*zero-neq-one*) *zero-neq-one*
⟨*proof*⟩

instance *fps* :: (*semiring-0*) *semiring*
⟨*proof*⟩

instance *fps* :: (*semiring-0*) *semiring-0*
⟨*proof*⟩

instance *fps* :: (*semiring-0-cancel*) *semiring-0-cancel* ⟨*proof*⟩

39.3 Selection of the nth power of the implicit variable in the infinite sum

lemma *fps-nonzero-nth*: $f \neq 0 \longleftrightarrow (\exists\ n.\ f\ \$n \neq 0)$
⟨*proof*⟩

lemma *fps-nonzero-nth-minimal*:
$f \neq 0 \longleftrightarrow (\exists n.\ f\ \$\ n \neq 0 \wedge (\forall m{<}n.\ f\ \$\ m = 0))$
⟨*proof*⟩

lemma *fps-eq-iff*: $f = g \longleftrightarrow (\forall n.\ f\ \$\ n = g\ \$n)$
⟨*proof*⟩

lemma *fps-setsum-nth*: $(setsum\ f\ S)\ \$\ n = setsum\ (\lambda k.\ (f\ k)\ \$\ n)\ S$
⟨*proof*⟩

39.4 Injection of the basic ring elements and multiplication by scalars

definition
$fps\text{-}const\ c = Abs\text{-}fps\ (\lambda n.\ if\ n = 0\ then\ c\ else\ 0)$

lemma *fps-nth-fps-const* [*simp*]: $fps\text{-}const\ c\ \$\ n = (if\ n = 0\ then\ c\ else\ 0)$
⟨*proof*⟩

lemma *fps-const-0-eq-0* [*simp*]: $fps\text{-}const\ 0 = 0$
⟨*proof*⟩

lemma *fps-const-1-eq-1* [*simp*]: $fps\text{-}const\ 1 = 1$
⟨*proof*⟩

lemma *fps-const-neg* [*simp*]: $-\ (fps\text{-}const\ (c::'a::ring)) = fps\text{-}const\ (-\ c)$
⟨*proof*⟩

lemma *fps-const-add* [*simp*]: $fps\text{-}const\ (c::'a::monoid\text{-}add) + fps\text{-}const\ d = fps\text{-}const\ (c + d)$
⟨*proof*⟩
lemma *fps-const-sub* [*simp*]: $fps\text{-}const\ (c::'a::group\text{-}add) - fps\text{-}const\ d = fps\text{-}const\ (c - d)$
⟨*proof*⟩
lemma *fps-const-mult*[*simp*]: $fps\text{-}const\ (c::'a::ring) * fps\text{-}const\ d = fps\text{-}const\ (c * d)$
⟨*proof*⟩

lemma *fps-const-add-left*: *fps-const* (*c*::'*a*::*monoid-add*) + *f* = *Abs-fps* (λ*n*. *if n* = *0 then c* + *f*$*0 else f*$*n*)
⟨*proof*⟩

lemma *fps-const-add-right*: *f* + *fps-const* (*c*::'*a*::*monoid-add*) = *Abs-fps* (λ*n*. *if n* = *0 then f*$*0* + *c else f*$*n*)
⟨*proof*⟩

lemma *fps-const-mult-left*: *fps-const* (*c*::'*a*::*semiring-0*) ∗ *f* = *Abs-fps* (λ*n*. *c* ∗ *f*$*n*)
⟨*proof*⟩

lemma *fps-const-mult-right*: *f* ∗ *fps-const* (*c*::'*a*::*semiring-0*) = *Abs-fps* (λ*n*. *f*$*n* ∗ *c*)
⟨*proof*⟩

lemma *fps-mult-left-const-nth* [*simp*]: (*fps-const* (*c*::'*a*::*semiring-1*) ∗ *f*)$*n* = *c*∗ *f*$*n*
⟨*proof*⟩

lemma *fps-mult-right-const-nth* [*simp*]: (*f* ∗ *fps-const* (*c*::'*a*::*semiring-1*))$*n* = *f*$*n* ∗ *c*
⟨*proof*⟩

39.5 Formal power series form an integral domain

instance *fps* :: (*ring*) *ring* ⟨*proof*⟩

instance *fps* :: (*ring-1*) *ring-1*
⟨*proof*⟩

instance *fps* :: (*comm-ring-1*) *comm-ring-1*
⟨*proof*⟩

instance *fps* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors*
⟨*proof*⟩

instance *fps* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors* ⟨*proof*⟩

instance *fps* :: (*idom*) *idom* ⟨*proof*⟩

instantiation *fps* :: (*comm-ring-1*) *number-ring*
begin
definition *number-of-fps-def*: (*number-of k*::'*a fps*) = *of-int k*

instance ⟨*proof*⟩
end

lemma *number-of-fps-const*: (*number-of k*::('*a*::*comm-ring-1*) *fps*) = *fps-const* (*of-int k*)

⟨*proof*⟩

39.6 The eXtractor series X

lemma *minus-one-power-iff*: (− (*1*::'*a* :: {*comm-ring-1*})) ^ *n* = (*if even n then 1 else* − *1*)
⟨*proof*⟩

definition *X* = *Abs-fps* (λ*n*. *if n* = *1 then 1 else 0*)
lemma *X-mult-nth*[*simp*]: (*X* ∗ (*f* :: ('*a*::*semiring-1*) *fps*)) $*n* = (*if n* = *0 then 0 else f* $ (*n* − *1*))
⟨*proof*⟩

lemma *X-mult-right-nth*[*simp*]: ((*f* :: ('*a*::*comm-semiring-1*) *fps*) ∗ *X*) $*n* = (*if n* = *0 then 0 else f* $ (*n* − *1*))
⟨*proof*⟩

lemma *X-power-iff*: *X*^*k* = *Abs-fps* (λ*n*. *if n* = *k then* (*1*::'*a*::*comm-ring-1*) *else 0*)
⟨*proof*⟩

lemma *X-power-mult-nth*: (*X*^*k* ∗ (*f* :: ('*a*::*comm-ring-1*) *fps*)) $*n* = (*if n* < *k then 0 else f* $ (*n* − *k*))
⟨*proof*⟩

lemma *X-power-mult-right-nth*: ((*f* :: ('*a*::*comm-ring-1*) *fps*) ∗ *X*^*k*) $*n* = (*if n* < *k then 0 else f* $ (*n* − *k*))
⟨*proof*⟩

39.7 Formal Power series form a metric space

definition (**in** *dist*) *ball-def*: *ball x r* = {*y*. *dist y x* < *r*}
instantiation *fps* :: (*comm-ring-1*) *dist*
begin

definition *dist-fps-def*: *dist* (*a*::'*a fps*) *b* = (*if* (∃ *n*. *a*$*n* ≠ *b*$*n*) *then inverse* (*2* ^ *The* (*leastP* (λ*n*. *a*$*n* ≠ *b*$*n*))) *else 0*)

lemma *dist-fps-ge0*: *dist* (*a*::'*a fps*) *b* ≥ *0*
⟨*proof*⟩

lemma *dist-fps-sym*: *dist* (*a*::'*a fps*) *b* = *dist b a*
⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *fps-nonzero-least-unique*: **assumes** *a0*: *a* ≠ *0*

shows $\exists!\ n.\ leastP\ (\lambda n.\ a\$n \neq 0)\ n$
⟨*proof*⟩

lemma *fps-eq-least-unique*: **assumes** *ab*: $(a::('a::ab\text{-}group\text{-}add)\ fps) \neq b$
shows $\exists!\ n.\ leastP\ (\lambda n.\ a\$n \neq b\$n)\ n$
⟨*proof*⟩

instantiation *fps* :: (*comm-ring-1*) *metric-space*
begin

definition *open-fps-def*: $open\ (S :: 'a\ fps\ set) = (\forall a \in S.\ \exists r.\ r > 0 \wedge ball\ a\ r \subseteq S)$

instance
⟨*proof*⟩

end

The infinite sums and justification of the notation in textbooks

lemma *reals-power-lt-ex*: **assumes** *xp*: $x > 0$ **and** *y1*: $(y::real) > 1$
shows $\exists k{>}0.\ (1/y)\hat{}k < x$
⟨*proof*⟩
lemma *X-nth*[*simp*]: $X\$n = (if\ n = 1\ then\ 1\ else\ 0)$ ⟨*proof*⟩
lemma *X-power-nth*[*simp*]: $(X\hat{}k)\ \$n = (if\ n = k\ then\ 1\ else\ (0::'a::comm\text{-}ring\text{-}1))$
⟨*proof*⟩

lemma *fps-sum-rep-nth*: $(setsum\ (\%i.\ fps\text{-}const(a\$i){*}X\hat{}i)\ \{0..m\})\$n = (if\ n \leq m\ then\ a\$n\ else\ (0::'a::comm\text{-}ring\text{-}1))$
⟨*proof*⟩

lemma *fps-notation*:
$(\%n.\ setsum\ (\%i.\ fps\text{-}const(a\$i) * X\hat{}i)\ \{0..n\}) \longrightarrow a$ (**is** *?s* $\longrightarrow a$)
⟨*proof*⟩

39.8 Inverses of formal power series

declare *setsum-cong*[*fundef-cong*]

instantiation *fps* :: ({*comm-monoid-add*, *inverse*, *times*, *uminus*}) *inverse*
begin

fun *natfun-inverse*:: $'a\ fps \Rightarrow nat \Rightarrow 'a$ **where**
natfun-inverse f 0 = *inverse* (*f*$*0*)
| *natfun-inverse f n* = − *inverse* (*f*$*0*) ∗ *setsum* (λ*i*. *f*$*i* ∗ *natfun-inverse f* (*n* − *i*)) {*1*..*n*}

definition *fps-inverse-def*:
inverse f = (*if f* $ *0* = *0 then 0 else Abs-fps* (*natfun-inverse f*))

definition *fps-divide-def*: *divide* = (λ(*f*::'*a fps*) *g*. *f* ∗ *inverse g*)

instance ⟨*proof*⟩

end

lemma *fps-inverse-zero*[*simp*]:
inverse (*0* :: '*a*::{*comm-monoid-add*,*inverse*, *times*, *uminus*} *fps*) = *0*
⟨*proof*⟩

lemma *fps-inverse-one*[*simp*]: *inverse* (*1* :: '*a*::{*division-ring*,*zero-neq-one*} *fps*) = *1*
⟨*proof*⟩

lemma *inverse-mult-eq-1* [*intro*]: **assumes** *f0*: *f*$*0* ≠ (*0*::'*a*::*field*)
shows *inverse f* ∗ *f* = *1*
⟨*proof*⟩

lemma *fps-inverse-0-iff*[*simp*]: (*inverse f*)$*0* = (*0*::'*a*::*division-ring*) ⟷ *f*$*0* = *0*
⟨*proof*⟩

lemma *fps-inverse-eq-0-iff*[*simp*]: *inverse f* = (*0*:: ('*a*::*field*) *fps*) ⟷ *f* $*0* = *0*
⟨*proof*⟩

lemma *fps-inverse-idempotent*[*intro*]: **assumes** *f0*: *f*$*0* ≠ (*0*::'*a*::*field*)
shows *inverse* (*inverse f*) = *f*
⟨*proof*⟩

lemma *fps-inverse-unique*: **assumes** *f0*: *f*$*0* ≠ (*0*::'*a*::*field*) **and** *fg*: *f*∗*g* = *1*
shows *inverse f* = *g*
⟨*proof*⟩

lemma *fps-inverse-gp*: *inverse* (*Abs-fps*(λ*n*. (*1*::'*a*::*field*)))
= *Abs-fps* (λ*n*. *if n*= *0 then 1 else if n*=*1 then* − *1 else 0*)
⟨*proof*⟩

39.9 Formal Derivatives, and the MacLaurin theorem around 0

definition *fps-deriv f* = *Abs-fps* (λ*n*. *of-nat* (*n* + *1*) ∗ *f* $ (*n* + *1*))

lemma *fps-deriv-nth*[*simp*]: *fps-deriv f* $ *n* = *of-nat* (*n* +*1*) ∗ *f* $ (*n*+*1*) ⟨*proof*⟩

lemma *fps-deriv-linear*[*simp*]: *fps-deriv* (*fps-const* (*a*::'*a*::*comm-semiring-1*) ∗ *f* + *fps-const b* ∗ *g*) = *fps-const a* ∗ *fps-deriv f* + *fps-const b* ∗ *fps-deriv g*
⟨*proof*⟩

lemma *fps-deriv-mult*[*simp*]:

fixes f :: ($'a$ :: *comm-ring-1*) *fps*
shows *fps-deriv* (f $*$ g) = f $*$ *fps-deriv* g + *fps-deriv* f $*$ g
⟨*proof*⟩

lemma *fps-deriv-X*[*simp*]: *fps-deriv* X = *1*
⟨*proof*⟩

lemma *fps-deriv-neg*[*simp*]: *fps-deriv* (− (f:: ($'a$:: *comm-ring-1*) *fps*)) = − (*fps-deriv* f)
⟨*proof*⟩
lemma *fps-deriv-add*[*simp*]: *fps-deriv* ((f:: ($'a$::*comm-ring-1*) *fps*) + g) = *fps-deriv* f + *fps-deriv* g
⟨*proof*⟩

lemma *fps-deriv-sub*[*simp*]: *fps-deriv* ((f:: ($'a$::*comm-ring-1*) *fps*) − g) = *fps-deriv* f − *fps-deriv* g
⟨*proof*⟩

lemma *fps-deriv-const*[*simp*]: *fps-deriv* (*fps-const* c) = *0*
⟨*proof*⟩

lemma *fps-deriv-mult-const-left*[*simp*]: *fps-deriv* (*fps-const* (c::$'a$::*comm-ring-1*) $*$ f) = *fps-const* c $*$ *fps-deriv* f
⟨*proof*⟩

lemma *fps-deriv-0*[*simp*]: *fps-deriv* *0* = *0*
⟨*proof*⟩

lemma *fps-deriv-1*[*simp*]: *fps-deriv* *1* = *0*
⟨*proof*⟩

lemma *fps-deriv-mult-const-right*[*simp*]: *fps-deriv* (f $*$ *fps-const* (c::$'a$::*comm-ring-1*)) = *fps-deriv* f $*$ *fps-const* c
⟨*proof*⟩

lemma *fps-deriv-setsum*: *fps-deriv* (*setsum* f S) = *setsum* (λi. *fps-deriv* (f i :: ($'a$::*comm-ring-1*) *fps*)) S
⟨*proof*⟩

lemma *fps-deriv-eq-0-iff*[*simp*]: *fps-deriv* f = *0* ⟷ (f = *fps-const* ($f$$*0* :: $'a$::{*idom*,*semiring-char-0*}))
⟨*proof*⟩

lemma *fps-deriv-eq-iff*:
fixes f:: ($'a$::{*idom*,*semiring-char-0*}) *fps*
shows *fps-deriv* f = *fps-deriv* g ⟷ (f = *fps-const*($f$$*0* − $g$$*0*) + g)
⟨*proof*⟩

lemma *fps-deriv-eq-iff-ex*: (*fps-deriv* f = *fps-deriv* g) ⟷ (∃ (c::$'a$::{*idom*,*semiring-char-0*}). f = *fps-const* c + g)

⟨*proof*⟩

fun *fps-nth-deriv* :: *nat* ⇒ ('*a*::*semiring-1*) *fps* ⇒ '*a fps* **where**
fps-nth-deriv 0 f = *f*
| *fps-nth-deriv* (*Suc n*) *f* = *fps-nth-deriv n* (*fps-deriv f*)

lemma *fps-nth-deriv-commute*: *fps-nth-deriv* (*Suc n*) *f* = *fps-deriv* (*fps-nth-deriv n f*)
⟨*proof*⟩

lemma *fps-nth-deriv-linear*[*simp*]: *fps-nth-deriv n* (*fps-const* (*a*::'*a*::*comm-semiring-1*) ∗ *f* + *fps-const b* ∗ *g*) = *fps-const a* ∗ *fps-nth-deriv n f* + *fps-const b* ∗ *fps-nth-deriv n g*
⟨*proof*⟩

lemma *fps-nth-deriv-neg*[*simp*]: *fps-nth-deriv n* (− (*f*:: ('*a*:: *comm-ring-1*) *fps*)) = − (*fps-nth-deriv n f*)
⟨*proof*⟩

lemma *fps-nth-deriv-add*[*simp*]: *fps-nth-deriv n* ((*f*:: ('*a*::*comm-ring-1*) *fps*) + *g*) = *fps-nth-deriv n f* + *fps-nth-deriv n g*
⟨*proof*⟩

lemma *fps-nth-deriv-sub*[*simp*]: *fps-nth-deriv n* ((*f*:: ('*a*::*comm-ring-1*) *fps*) − *g*) = *fps-nth-deriv n f* − *fps-nth-deriv n g*
⟨*proof*⟩

lemma *fps-nth-deriv-0*[*simp*]: *fps-nth-deriv n 0* = *0*
⟨*proof*⟩

lemma *fps-nth-deriv-1*[*simp*]: *fps-nth-deriv n 1* = (*if n* = *0 then 1 else 0*)
⟨*proof*⟩

lemma *fps-nth-deriv-const*[*simp*]: *fps-nth-deriv n* (*fps-const c*) = (*if n* = *0 then fps-const c else 0*)
⟨*proof*⟩

lemma *fps-nth-deriv-mult-const-left*[*simp*]: *fps-nth-deriv n* (*fps-const* (*c*::'*a*::*comm-ring-1*) ∗ *f*) = *fps-const c* ∗ *fps-nth-deriv n f*
⟨*proof*⟩

lemma *fps-nth-deriv-mult-const-right*[*simp*]: *fps-nth-deriv n* (*f* ∗ *fps-const* (*c*::'*a*::*comm-ring-1*)) = *fps-nth-deriv n f* ∗ *fps-const c*
⟨*proof*⟩

lemma *fps-nth-deriv-setsum*: *fps-nth-deriv n* (*setsum f S*) = *setsum* (λ*i*. *fps-nth-deriv n* (*f i* :: ('*a*::*comm-ring-1*) *fps*)) *S*
⟨*proof*⟩

lemma *fps-deriv-maclauren-0*: (*fps-nth-deriv k* (*f*:: ('*a*::*comm-semiring-1*) *fps*)) $ *0* = *of-nat* (*fact k*) ∗ *f*$(*k*)
⟨*proof*⟩

39.10 Powers

lemma *fps-power-zeroth-eq-one*: *a*$*0* =*1* ⟹ *a* ^ *n* $ *0* = (*1*::'*a*::*semiring-1*)
⟨*proof*⟩

lemma *fps-power-first-eq*: (*a*:: '*a*::*comm-ring-1 fps*)$*0* =*1* ⟹ *a* ^ *n* $ *1* = *of-nat n* ∗ *a*$*1*
⟨*proof*⟩

lemma *startsby-one-power*:*a* $ *0* = (*1*::'*a*::*comm-ring-1*) ⟹ *a* ^ *n* $ *0* = *1*
⟨*proof*⟩

lemma *startsby-zero-power*:*a* $*0* = (*0*::'*a*::*comm-ring-1*) ⟹ *n* > *0* ⟹ *a* ^ *n* $*0* = *0*
⟨*proof*⟩

lemma *startsby-power*:*a* $*0* = (*v*::'*a*::{*comm-ring-1*}) ⟹ *a* ^ *n* $*0* = *v* ^ *n*
⟨*proof*⟩

lemma *startsby-zero-power-iff* [*simp*]:
a ^ *n* $*0* = (*0*::'*a*::{*idom*}) ⟷ (*n* ≠ *0* ∧ *a*$*0* = *0*)
⟨*proof*⟩

lemma *startsby-zero-power-prefix*:
assumes *a0*: *a* $*0* = (*0*::'*a*::*idom*)
shows ∀ *n* < *k*. *a* ^ *k* $ *n* = *0*
⟨*proof*⟩

lemma *startsby-zero-setsum-depends*:
assumes *a0*: *a* $*0* = (*0*::'*a*::*idom*) **and** *kn*: *n* ≥ *k*
shows *setsum* (λ*i*. (*a* ^ *i*)$*k*) {*0* .. *n*} = *setsum* (λ*i*. (*a* ^ *i*)$*k*) {*0* .. *k*}
⟨*proof*⟩

lemma *startsby-zero-power-nth-same*: **assumes** *a0*: *a*$*0* = (*0*::'*a*::{*idom*})
shows *a* ^ *n* $ *n* = (*a*$*1*) ^ *n*
⟨*proof*⟩

lemma *fps-inverse-power*:
fixes *a* :: ('*a*::{*field*}) *fps*
shows *inverse* (*a* ^ *n*) = *inverse a* ^ *n*
⟨*proof*⟩

lemma *fps-deriv-power*: *fps-deriv* (*a* ^ *n*) = *fps-const* (*of-nat n* :: '*a*:: *comm-ring-1*) ∗ *fps-deriv a* ∗ *a* ^ (*n* − *1*)

⟨*proof*⟩

lemma *fps-inverse-deriv*:
fixes *a*:: ('*a* :: *field*) *fps*
assumes *a0*: *a*$*0* ≠ *0*
shows *fps-deriv* (*inverse a*) = − *fps-deriv a* ∗ *inverse a* ^ *2*
⟨*proof*⟩

lemma *fps-inverse-mult*:
fixes *a*::('*a* :: *field*) *fps*
shows *inverse* (*a* ∗ *b*) = *inverse a* ∗ *inverse b*
⟨*proof*⟩

lemma *fps-inverse-deriv'*:
fixes *a*:: ('*a* :: *field*) *fps*
assumes *a0*: *a*$*0* ≠ *0*
shows *fps-deriv* (*inverse a*) = − *fps-deriv a* / *a* ^ *2*
⟨*proof*⟩

lemma *inverse-mult-eq-1'*: **assumes** *f0*: *f*$*0* ≠ (*0*::'*a*::*field*)
shows *f* ∗ *inverse f* = *1*
⟨*proof*⟩

lemma *fps-divide-deriv*: **fixes** *a*:: ('*a* :: *field*) *fps*
assumes *a0*: *b*$*0* ≠ *0*
shows *fps-deriv* (*a* / *b*) = (*fps-deriv a* ∗ *b* − *a* ∗ *fps-deriv b*) / *b* ^ *2*
⟨*proof*⟩

lemma *fps-inverse-gp'*: *inverse* (*Abs-fps*(λ*n*. (*1*::'*a*::*field*)))
= *1* − *X*
⟨*proof*⟩

lemma *fps-nth-deriv-X*[*simp*]: *fps-nth-deriv n X* = (*if n* = *0 then X else if n*=*1 then 1 else 0*)
⟨*proof*⟩

lemma *fps-inverse-X-plus1*:
inverse (*1* + *X*) = *Abs-fps* (λ*n*. (− (*1*::'*a*::{*field*})) ^ *n*) (**is** - = *?r*)
⟨*proof*⟩

39.11 Integration

definition
fps-integral :: '*a*::*field-char-0 fps* ⇒ '*a* ⇒ '*a fps* **where**
fps-integral a a0 = *Abs-fps* (λ*n*. *if n* = *0 then a0 else* (*a*$(*n* − *1*) / *of-nat n*))

lemma *fps-deriv-fps-integral*: *fps-deriv* (*fps-integral a a0*) = *a*

⟨*proof*⟩

lemma *fps-integral-linear*:
fps-integral (*fps-const a* ∗ *f* + *fps-const b* ∗ *g*) (*a*∗*a0* + *b*∗*b0*) =
fps-const a ∗ *fps-integral f a0* + *fps-const b* ∗ *fps-integral g b0*
(**is** *?l* = *?r*)
⟨*proof*⟩

39.12 Composition of FPSs

definition *fps-compose* :: ('*a*::*semiring-1*) *fps* ⇒ '*a fps* ⇒ '*a fps* (**infixl** *oo* 55)
where
fps-compose-def: *a oo b* = *Abs-fps* (λ*n*. *setsum* (λ*i*. *a*$*i* ∗ (*b* ^ *i*$*n*)) {0..*n*})

lemma *fps-compose-nth*: (*a oo b*)$*n* = *setsum* (λ*i*. *a*$*i* ∗ (*b* ^ *i*$*n*)) {0..*n*} ⟨*proof*⟩

lemma *fps-compose-X*[*simp*]: *a oo X* = (*a* :: ('*a* :: *comm-ring-1*) *fps*)
⟨*proof*⟩

lemma *fps-const-compose*[*simp*]:
fps-const (*a*::'*a*::{*comm-ring-1*}) *oo b* = *fps-const* (*a*)
⟨*proof*⟩

lemma *number-of-compose*[*simp*]: (*number-of k*::('*a*::{*comm-ring-1*}) *fps*) *oo b* = *number-of k*
⟨*proof*⟩

lemma *X-fps-compose-startby0*[*simp*]: *a*$0 = 0 ⟹ *X oo a* = (*a* :: ('*a* :: *comm-ring-1*) *fps*)
⟨*proof*⟩

39.13 Rules from Herbert Wilf's Generatingfunctionology

39.13.1 Rule 1

lemma *fps-power-mult-eq-shift*:
X ^ *Suc k* ∗ *Abs-fps* (λ*n*. *a* (*n* + *Suc k*)) = *Abs-fps a* − *setsum* (λ*i*. *fps-const* (*a i* :: '*a*:: *comm-ring-1*) ∗ *X* ^ *i*) {0 .. *k*} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

39.13.2 Rule 2

definition *XD* = *op* ∗ *X o fps-deriv*

lemma *XD-add*[*simp*]:*XD* (*a* + *b*) = *XD a* + *XD* (*b* :: ('*a*::*comm-ring-1*) *fps*)
⟨*proof*⟩

lemma *XD-mult-const*[*simp*]:*XD* (*fps-const* (*c*::'*a*::*comm-ring-1*) ∗ *a*) = *fps-const c* ∗ *XD a*
⟨*proof*⟩

lemma *XD-linear*[*simp*]: *XD* (*fps-const c* ∗ *a* + *fps-const d* ∗ *b*) = *fps-const c* ∗ *XD a* + *fps-const d* ∗ *XD* (*b* :: ('*a*::*comm-ring-1*) *fps*)
⟨*proof*⟩

lemma *XDN-linear*:
(*XD* ^^ *n*) (*fps-const c* ∗ *a* + *fps-const d* ∗ *b*) = *fps-const c* ∗ (*XD* ^^ *n*) *a* + *fps-const d* ∗ (*XD* ^^ *n*) (*b* :: ('*a*::*comm-ring-1*) *fps*)
⟨*proof*⟩

lemma *fps-mult-X-deriv-shift*: *X*∗ *fps-deriv a* = *Abs-fps* (λ*n*. *of-nat n*∗ *a*$*n*)
⟨*proof*⟩

lemma *fps-mult-XD-shift*:
(*XD* ^^ *k*) (*a*:: ('*a*::{*comm-ring-1*}) *fps*) = *Abs-fps* (λ*n*. (*of-nat n* ^ *k*) ∗ *a*$*n*)
⟨*proof*⟩

39.13.3 Rule 3 is trivial and is given by *fps-times-def*

39.13.4 Rule 5 — summation and "division" by (1 - X)

lemma *fps-divide-X-minus1-setsum-lemma*:
a = ((*1*::('*a*::*comm-ring-1*) *fps*) − *X*) ∗ *Abs-fps* (λ*n*. *setsum* (λ*i*. *a* $ *i*) {*0*..*n*})
⟨*proof*⟩

lemma *fps-divide-X-minus1-setsum*:
a /((*1*::('*a*::*field*) *fps*) − *X*) = *Abs-fps* (λ*n*. *setsum* (λ*i*. *a* $ *i*) {*0*..*n*})
⟨*proof*⟩

39.13.5 Rule 4 in its more general form: generalizes Rule 3 for an arbitrary finite product of FPS, also the relvant instance of powers of a FPS

definition *natpermute n k* = {*l*:: *nat list*. *length l* = *k* ∧ *foldl op* + *0 l* = *n*}

lemma *natlist-trivial-1*: *natpermute n 1* = {[*n*]}
⟨*proof*⟩

lemma *foldl-add-start0*:
foldl op + *x xs* = *x* + *foldl op* + (*0*::*nat*) *xs*
⟨*proof*⟩

lemma *foldl-add-append*: *foldl op* + (*x*::*nat*) (*xs*@*ys*) = *foldl op* + *x xs* + *foldl op* + *0 ys*
⟨*proof*⟩

lemma *foldl-add-setsum*: *foldl op* + (*x*::*nat*) *xs* = *x* + *setsum* (*nth xs*) {*0*..<*length xs*}
⟨*proof*⟩

lemma *append-natpermute-less-eq*:
assumes *h*: *xs@ys* ∈ *natpermute n k* **shows** *foldl op + 0 xs* ≤ *n* **and** *foldl op + 0 ys* ≤ *n*
⟨*proof*⟩

lemma *natpermute-split*:
assumes *mn*: *h* ≤ *k*
shows *natpermute n k* = (⋃ *m* ∈{*0..n*}. {*l1* @ *l2* |*l1 l2*. *l1* ∈ *natpermute m h* ∧ *l2* ∈ *natpermute* (*n* − *m*) (*k* − *h*)}) (**is** *?L* = *?R* **is** *?L* = (⋃ *m* ∈{*0..n*}. *?S m*))
⟨*proof*⟩

lemma *natpermute-0*: *natpermute n 0* = (*if n* = *0 then* {[]} *else* {})
⟨*proof*⟩
lemma *natpermute-0'*[*simp*]: *natpermute 0 k* = (*if k* = *0 then* {[]} *else* {*replicate k 0*})
⟨*proof*⟩

lemma *natpermute-finite*: *finite* (*natpermute n k*)
⟨*proof*⟩

lemma *natpermute-contain-maximal*:
{*xs* ∈ *natpermute n* (*k*+*1*). *n* ∈ *set xs*} = *UNION* {*0* .. *k*} (λ*i*. {(*replicate* (*k*+*1*) *0*) [*i*:=*n*]})
(**is** *?A* = *?B*)
⟨*proof*⟩

lemma *fps-setprod-nth*:
fixes *m* :: *nat* **and** *a* :: *nat* ⇒ ('*a*::*comm-ring-1*) *fps*
shows (*setprod a* {*0* .. *m*})$*n* = *setsum* (λ*v*. *setprod* (λ*j*. (*a j*) $ (*v*!*j*)) {*0..m*}) (*natpermute n* (*m*+*1*))
(**is** *?P m n*)
⟨*proof*⟩

The special form for powers

lemma *fps-power-nth-Suc*:
fixes *m* :: *nat* **and** *a* :: ('*a*::*comm-ring-1*) *fps*
shows (*a* ^ *Suc m*)$*n* = *setsum* (λ*v*. *setprod* (λ*j*. *a* $ (*v*!*j*)) {*0..m*}) (*natpermute n* (*m*+*1*))
⟨*proof*⟩
lemma *fps-power-nth*:
fixes *m* :: *nat* **and** *a* :: ('*a*::*comm-ring-1*) *fps*
shows (*a* ^*m*)$*n* = (*if m*=*0 then 1*$*n else setsum* (λ*v*. *setprod* (λ*j*. *a* $ (*v*!*j*)) {*0..m* − *1*}) (*natpermute n m*))
⟨*proof*⟩

lemma *fps-nth-power-0*:

fixes $m :: nat$ **and** $a :: ('a::\{comm\text{-}ring\text{-}1\})\ fps$
shows $(a\ \hat{}\ m)\$0 = (a\$0)\ \hat{}\ m$
⟨*proof*⟩

lemma *fps-compose-inj-right*:
assumes *a0*: $a\$0 = (0::'a::\{idom\})$
and *a1*: $a\$1 \neq 0$
shows $(b\ oo\ a = c\ oo\ a) \longleftrightarrow b = c$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

39.14 Radicals

declare *setprod-cong*[*fundef-cong*]
function *radical* :: $(nat \Rightarrow 'a \Rightarrow 'a) \Rightarrow nat \Rightarrow ('a::\{field\})\ fps \Rightarrow nat \Rightarrow 'a$ **where**
radical r 0 a 0 = 1
| *radical r 0 a (Suc n) = 0*
| *radical r (Suc k) a 0 = r (Suc k) (a$0)*
| *radical r (Suc k) a (Suc n) = (a$ Suc n − setsum (λxs. setprod (λj. radical r (Suc k) a (xs ! j)) {0..k}) {xs. xs ∈ natpermute (Suc n) (Suc k) ∧ Suc n ∉ set xs}) / (of-nat (Suc k) ∗ (radical r (Suc k) a 0) ^k)*
⟨*proof*⟩

termination *radical*
⟨*proof*⟩

definition *fps-radical r n a = Abs-fps (radical r n a)*

lemma *fps-radical0*[*simp*]: *fps-radical r 0 a = 1*
⟨*proof*⟩

lemma *fps-radical-nth-0*[*simp*]: *fps-radical r n a $ 0 = (if n=0 then 1 else r n (a$0))*
⟨*proof*⟩

lemma *fps-radical-power-nth*[*simp*]:
assumes *r*: $(r\ k\ (a\$0))\ \hat{}\ k = a\$0$
shows *fps-radical r k a ^ k $ 0 = (if k = 0 then 1 else a$0)*
⟨*proof*⟩

lemma *natpermute-max-card*: **assumes** *n0*: $n \neq 0$
shows *card {xs ∈ natpermute n (k+1). n ∈ set xs} = k+1*
⟨*proof*⟩

lemma *power-radical*:
fixes *a*:: *'a*::*field-char-0 fps*
assumes *a0*: *a$0 ≠ 0*
shows *(r (Suc k) (a$0)) ^ Suc k = a$0 ⟷ (fps-radical r (Suc k) a) ^ (Suc k) = a*
⟨*proof*⟩

lemma *eq-divide-imp'*: **assumes** *c0*: (*c*::'*a*::*field*) ~= *0* **and** *eq*: *a* * *c* = *b*
shows *a* = *b* / *c*
⟨*proof*⟩

lemma *radical-unique*:
assumes *r0*: (*r* (*Suc k*) (*b*$*0*)) ^ *Suc k* = *b*$*0*
and *a0*: *r* (*Suc k*) (*b*$*0* ::'*a*::*field-char-0*) = *a*$*0* **and** *b0*: *b*$*0* ≠ *0*
shows *a*^(*Suc k*) = *b* ⟷ *a* = *fps-radical r* (*Suc k*) *b*
⟨*proof*⟩

lemma *radical-power*:
assumes *r0*: *r* (*Suc k*) ((*a*$*0*) ^ *Suc k*) = *a*$*0*
and *a0*: (*a*$*0* ::'*a*::*field-char-0*) ≠ *0*
shows (*fps-radical r* (*Suc k*) (*a* ^ *Suc k*)) = *a*
⟨*proof*⟩

lemma *fps-deriv-radical*:
fixes *a*:: '*a*::*field-char-0 fps*
assumes *r0*: (*r* (*Suc k*) (*a*$*0*)) ^ *Suc k* = *a*$*0* **and** *a0*: *a*$*0* ≠ *0*
shows *fps-deriv* (*fps-radical r* (*Suc k*) *a*) = *fps-deriv a* / (*fps-const* (*of-nat* (*Suc k*)) * (*fps-radical r* (*Suc k*) *a*) ^ *k*)
⟨*proof*⟩

lemma *radical-mult-distrib*:
fixes *a*:: '*a*::*field-char-0 fps*
assumes
k: *k* > *0*
and *ra0*: *r k* (*a* $ *0*) ^ *k* = *a* $ *0*
and *rb0*: *r k* (*b* $ *0*) ^ *k* = *b* $ *0*
and *a0*: *a*$*0* ≠ *0*
and *b0*: *b*$*0* ≠ *0*
shows *r k* ((*a* * *b*) $ *0*) = *r k* (*a* $ *0*) * *r k* (*b* $ *0*) ⟷ *fps-radical r* (*k*) (*a**b*) = *fps-radical r* (*k*) *a* * *fps-radical r* (*k*) (*b*)
⟨*proof*⟩

lemma *fps-divide-1*[*simp*]: (*a*:: ('*a*::*field*) *fps*) / *1* = *a*
⟨*proof*⟩

lemma *radical-divide*:
fixes *a* :: '*a*::*field-char-0 fps*
assumes
kp: *k*>*0*
and *ra0*: (*r k* (*a* $ *0*)) ^ *k* = *a* $ *0*
and *rb0*: (*r k* (*b* $ *0*)) ^ *k* = *b* $ *0*

and *a0*: $a\$0 \neq 0$
and *b0*: $b\$0 \neq 0$
shows $r\ k\ ((a\ \$\ 0)\ /\ (b\$0)) = r\ k\ (a\$0)\ /\ r\ k\ (b\ \$\ 0) \longleftrightarrow$ *fps-radical r k* (a/b) $=$ *fps-radical r k a / fps-radical r k b* (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *radical-inverse*:
fixes a :: $'a$::*field-char-0 fps*
assumes
k: $k>0$
and *ra0*: $r\ k\ (a\ \$\ 0)\ \hat{}\ k = a\ \$\ 0$
and *r1*: $(r\ k\ 1)\hat{}k = 1$
and *a0*: $a\$0 \neq 0$
shows $r\ k$ (*inverse* $(a\ \$\ 0)) = r\ k\ 1\ /\ (r\ k\ (a\ \$\ 0)) \longleftrightarrow$ *fps-radical r k* (*inverse* a) $=$ *fps-radical r k 1 / fps-radical r k a*
⟨*proof*⟩

39.15 Derivative of composition

lemma *fps-compose-deriv*:
fixes a:: ($'a$::*idom*) *fps*
assumes *b0*: $b\$0 = 0$
shows *fps-deriv* $(a\ oo\ b) = ((fps\text{-}deriv\ a)\ oo\ b) * (fps\text{-}deriv\ b)$
⟨*proof*⟩

lemma *fps-mult-X-plus-1-nth*:
$((1+X)*a)\ \$n = (if\ n = 0\ then\ (a\$n :: 'a::comm\text{-}ring\text{-}1)\ else\ a\$n + a\$(n - 1))$
⟨*proof*⟩

39.16 Finite FPS (i.e. polynomials) and X

lemma *fps-poly-sum-X*:
assumes *z*: $\forall i > n.\ a\$i = (0::'a::comm\text{-}ring\text{-}1)$
shows $a =$ *setsum* $(\lambda i.\ fps\text{-}const\ (a\$i) * X\hat{}i)\ \{0..n\}$ (**is** $a = ?r$)
⟨*proof*⟩

39.17 Compositional inverses

fun *compinv* :: $'a$ *fps* $\Rightarrow$ *nat* $\Rightarrow$ $'a$::{*field*} **where**
compinv a 0 $= X\$0$
| *compinv a* (*Suc n*) $= (X\$\ Suc\ n - setsum\ (\lambda i.\ (compinv\ a\ i) * (a\hat{}i)\$Suc\ n)\ \{0\ ..\ n\})\ /\ (a\$1)\ \hat{}\ Suc\ n$

definition *fps-inv a* $=$ *Abs-fps* (*compinv a*)

lemma *fps-inv*: **assumes** *a0*: $a\$0 = 0$ **and** *a1*: $a\$1 \neq 0$
shows *fps-inv a oo a* $= X$
⟨*proof*⟩

fun *gcompinv* :: *'a fps ⇒ 'a fps ⇒ nat ⇒ 'a*::{*field*} **where**
  gcompinv b a 0 = b$0
| *gcompinv b a (Suc n) = (b$ Suc n − setsum (λi. (gcompinv b a i) * (a^i)$Suc n) {0 .. n}) / (a$1) ^ Suc n*

definition *fps-ginv b a = Abs-fps (gcompinv b a)*

lemma *fps-ginv*: **assumes** *a0*: *a$0 = 0* **and** *a1*: *a$1 ≠ 0*
  shows *fps-ginv b a oo a = b*
⟨*proof*⟩

lemma *fps-inv-ginv*: *fps-inv = fps-ginv X*
  ⟨*proof*⟩

lemma *fps-compose-1*[*simp*]: *1 oo a = 1*
  ⟨*proof*⟩

lemma *fps-compose-0*[*simp*]: *0 oo a = 0*
  ⟨*proof*⟩

lemma *fps-compose-0-right*[*simp*]: *a oo 0 = fps-const (a$0)*
  ⟨*proof*⟩

lemma *fps-compose-add-distrib*: *(a + b) oo c = (a oo c) + (b oo c)*
  ⟨*proof*⟩

lemma *fps-compose-setsum-distrib*: *(setsum f S) oo a = setsum (λi. f i oo a) S*
⟨*proof*⟩

lemma *convolution-eq*:
  *setsum (%i. a (i :: nat) * b (n − i)) {0 .. n} = setsum (%(i,j). a i * b j) {(i,j). i <= n ∧ j ≤ n ∧ i + j = n}*
  ⟨*proof*⟩

lemma *product-composition-lemma*:
  assumes *c0*: *c$0 = (0::'a::idom)* **and** *d0*: *d$0 = 0*
  shows *((a oo c) * (b oo d))$n = setsum (%(k,m). a$k * b$m * (c^k * d^m) $ n) {(k,m). k + m ≤ n}* (**is** *?l = ?r*)
⟨*proof*⟩

lemma *product-composition-lemma'*:
  assumes *c0*: *c$0 = (0::'a::idom)* **and** *d0*: *d$0 = 0*
  shows *((a oo c) * (b oo d))$n = setsum (%k. setsum (%m. a$k * b$m * (c^k * d^m) $ n) {0..n}) {0..n}* (**is** *?l = ?r*)
  ⟨*proof*⟩

lemma *setsum-pair-less-iff*:
  *setsum (%((k::nat),m). a k * b m * c (k + m)) {(k,m). k + m ≤ n} = setsum*

(%*s*. *setsum* (%*i*. *a i* ∗ *b* (*s* − *i*) ∗ *c s*) {*0*..*s*}) {*0*..*n*} (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-compose-mult-distrib-lemma*:
assumes *c0*: *c*$*0* = (*0*::*'a*::*idom*)
shows ((*a oo c*) ∗ (*b oo c*))$*n* = *setsum* (%*s*. *setsum* (%*i*. *a*$*i* ∗ *b*$(*s* − *i*) ∗ (*c*^*s*) $ *n*) {*0*..*s*}) {*0*..*n*} (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-compose-mult-distrib*:
assumes *c0*: *c*$*0* = (*0*::*'a*::*idom*)
shows (*a* ∗ *b*) *oo c* = (*a oo c*) ∗ (*b oo c*) (**is** *?l* = *?r*)
⟨*proof*⟩
lemma *fps-compose-setprod-distrib*:
assumes *c0*: *c*$*0* = (*0*::*'a*::*idom*)
shows (*setprod a S*) *oo c* = *setprod* (%*k*. *a k oo c*) *S* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-compose-power*: **assumes** *c0*: *c*$*0* = (*0*::*'a*::*idom*)
shows (*a oo c*)^*n* = *a*^*n oo c* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-compose-uminus*: − (*a*::*'a*::*ring-1 fps*) *oo c* = − (*a oo c*)
⟨*proof*⟩

lemma *fps-compose-sub-distrib*:
shows (*a* − *b*) *oo* (*c*::*'a*::*ring-1 fps*) = (*a oo c*) − (*b oo c*)
⟨*proof*⟩

lemma *X-fps-compose*:*X oo a* = *Abs-fps* (λ*n*. *if n* = *0 then* (*0*::*'a*::*comm-ring-1*) *else a*$*n*)
⟨*proof*⟩

lemma *fps-inverse-compose*:
assumes *b0*: (*b*$*0* :: *'a*::*field*) = *0* **and** *a0*: *a*$*0* ≠ *0*
shows *inverse a oo b* = *inverse* (*a oo b*)
⟨*proof*⟩

lemma *fps-divide-compose*:
assumes *c0*: (*c*$*0* :: *'a*::*field*) = *0* **and** *b0*: *b*$*0* ≠ *0*
shows (*a*/*b*) *oo c* = (*a oo c*) / (*b oo c*)
⟨*proof*⟩

lemma *gp*: **assumes** *a0*: *a*$*0* = (*0*::*'a*::*field*)
shows (*Abs-fps* (λ*n*. *1*)) *oo a* = *1*/(*1* − *a*) (**is** *?one oo a* = -)
⟨*proof*⟩

lemma *fps-const-power*[*simp*]: *fps-const* (*c*::*'a*::*ring-1*) ^ *n* = *fps-const* (*c*^*n*)

⟨*proof*⟩

lemma *fps-compose-radical*:
assumes *b0*: *b$0* = (*0*::*'a*::*field-char-0*)
and *ra0*: *r* (*Suc k*) (*a$0*) ^ *Suc k* = *a$0*
and *a0*: *a$0* ≠ *0*
shows *fps-radical r* (*Suc k*) *a oo b* = *fps-radical r* (*Suc k*) (*a oo b*)
⟨*proof*⟩

lemma *fps-const-mult-apply-left*:
fps-const c ∗ (*a oo b*) = (*fps-const c* ∗ *a*) *oo b*
⟨*proof*⟩

lemma *fps-const-mult-apply-right*:
(*a oo b*) ∗ *fps-const* (*c*::*'a*::*comm-semiring-1*) = (*fps-const c* ∗ *a*) *oo b*
⟨*proof*⟩

lemma *fps-compose-assoc*:
assumes *c0*: *c$0* = (*0*::*'a*::*idom*) **and** *b0*: *b$0* = *0*
shows *a oo* (*b oo c*) = *a oo b oo c* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-X-power-compose*:
assumes *a0*: *a$0*=*0* **shows** *X*^*k oo a* = (*a*::(*'a*::*idom fps*))^*k* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-inv-right*: **assumes** *a0*: *a$0* = *0* **and** *a1*: *a$1* ≠ *0*
shows *a oo fps-inv a* = *X*
⟨*proof*⟩

lemma *fps-inv-deriv*:
assumes *a0*:*a$0* = (*0*::*'a*::{*field*}) **and** *a1*: *a$1* ≠ *0*
shows *fps-deriv* (*fps-inv a*) = *inverse* (*fps-deriv a oo fps-inv a*)
⟨*proof*⟩

lemma *fps-inv-idempotent*:
assumes *a0*: *a$0* = *0* **and** *a1*: *a$1* ≠ *0*
shows *fps-inv* (*fps-inv a*) = *a*
⟨*proof*⟩

lemma *fps-ginv-ginv*:
assumes *a0*: *a$0* = *0* **and** *a1*: *a$1* ≠ *0*
and *c0*: *c$0* = *0* **and** *c1*: *c$1* ≠ *0*
shows *fps-ginv b* (*fps-ginv c a*) = *b oo a oo fps-inv c*
⟨*proof*⟩

lemma *fps-ginv-deriv*:
assumes *a0*:*a$0* = (*0*::*'a*::{*field*}) **and** *a1*: *a$1* ≠ *0*

shows *fps-deriv* (*fps-ginv b a*) = (*fps-deriv b* / *fps-deriv a*) *oo fps-ginv X a*
⟨*proof*⟩

39.18 Elementary series

39.18.1 Exponential series

definition *E x* = *Abs-fps* (λ*n*. *x*^*n* / *of-nat* (*fact n*))

lemma *E-deriv*[*simp*]: *fps-deriv* (*E a*) = *fps-const* (*a*::'*a*::*field-char-0*) * *E a* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *E-unique-ODE*:
fps-deriv a = *fps-const c* * *a* ⟷ *a* = *fps-const* (*a*$*0*) * *E* (*c* :: '*a*::*field-char-0*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *E-add-mult*: *E* (*a* + *b*) = *E* (*a*::'*a*::*field-char-0*) * *E b* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *E-nth*[*simp*]: *E a* $ *n* = *a*^*n* / *of-nat* (*fact n*)
⟨*proof*⟩

lemma *E0*[*simp*]: *E* (*0*::'*a*::{*field*}) = *1*
⟨*proof*⟩

lemma *E-neg*: *E* (− *a*) = *inverse* (*E* (*a*::'*a*::*field-char-0*))
⟨*proof*⟩

lemma *E-nth-deriv*[*simp*]: *fps-nth-deriv n* (*E* (*a*::'*a*::*field-char-0*)) = (*fps-const a*)^*n* * (*E a*)
⟨*proof*⟩

lemma *X-compose-E*[*simp*]: *X oo E* (*a*::'*a*::{*field*}) = *E a* − *1*
⟨*proof*⟩

lemma *LE-compose*:
assumes *a*: *a*≠*0*
shows *fps-inv* (*E a* − *1*) *oo* (*E a* − *1*) = *X*
and (*E a* − *1*) *oo fps-inv* (*E a* − *1*) = *X*
⟨*proof*⟩

lemma *fps-const-inverse*:
a ≠ *0* ⟹ *inverse* (*fps-const* (*a*::'*a*::*field*)) = *fps-const* (*inverse a*)
⟨*proof*⟩

lemma *inverse-one-plus-X*:
inverse (*1* + *X*) = *Abs-fps* (λ*n*. (− *1* ::'*a*::{*field*})^*n*)

(**is** *inverse ?l = ?r*)
⟨*proof*⟩

lemma *E-power-mult*: (*E* (*c*::'*a*::*field-char-0*)) ^ *n* = *E* (*of-nat n* * *c*)
⟨*proof*⟩

lemma *radical-E*:
assumes *r*: *r* (*Suc k*) *1* = *1*
shows *fps-radical r* (*Suc k*) (*E* (*c*::'*a*::{*field-char-0*})) = *E* (*c* / *of-nat* (*Suc k*))
⟨*proof*⟩

lemma *Ec-E1-eq*:
E (*1*::'*a*::{*field-char-0*}) *oo* (*fps-const c* * *X*) = *E c*
⟨*proof*⟩

The generalized binomial theorem as a consequence of *E* (*?a* + *?b*) = *E ?a* * *E ?b*

lemma *gbinomial-theorem*:
((*a*::'*a*::{*field-char-0*, *field-inverse-zero*})+*b*) ^ *n* = (∑ *k*=*0*..*n*. *of-nat* (*n choose k*) * *a*^*k* * *b*^(*n*−*k*))
⟨*proof*⟩

And the nat-form – also available from Binomial.thy

lemma *binomial-theorem*: (*a*+*b*) ^ *n* = (∑ *k*=*0*..*n*. (*n choose k*) * *a*^*k* * *b*^(*n*−*k*))
⟨*proof*⟩

39.18.2 Logarithmic series

lemma *Abs-fps-if-0*:
Abs-fps(%*n*. *if n*=*0 then* (*v*::'*a*::*ring-1*) *else f n*) = *fps-const v* + *X* * *Abs-fps* (%*n*. *f* (*Suc n*))
⟨*proof*⟩

definition *L*:: '*a*::{*field-char-0*} ⇒ '*a fps* **where**
L c ≡ *fps-const* (*1*/*c*) * *Abs-fps* (λ*n*. *if n* = *0 then 0 else* (− *1*) ^ (*n* − *1*) / *of-nat n*)

lemma *fps-deriv-L*: *fps-deriv* (*L c*) = *fps-const* (*1*/*c*) * *inverse* (*1* + *X*)
⟨*proof*⟩

lemma *L-nth*: *L c* $ *n* = (*if n*=*0 then 0 else 1*/*c* * ((− *1*) ^ (*n* − *1*) / *of-nat n*))
⟨*proof*⟩

lemma *L-0*[*simp*]: *L c* $ *0* = *0* ⟨*proof*⟩
lemma *L-E-inv*:
assumes *a*: *a*≠ (*0*::'*a*::{*field-char-0*})
shows *L a* = *fps-inv* (*E a* − *1*) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *L-mult-add*:
assumes *c0*: $c \neq 0$ **and** *d0*: $d \neq 0$
shows $L\ c + L\ d = fps\text{-}const\ (c+d) * L\ (c*d)$
(**is** *?r* = *?l*)
⟨*proof*⟩

39.18.3 Binomial series

definition $fps\text{-}binomial\ a = Abs\text{-}fps\ (\lambda n.\ a\ gchoose\ n)$

lemma *fps-binomial-nth*[*simp*]: $fps\text{-}binomial\ a\ \$\ n = a\ gchoose\ n$
⟨*proof*⟩

lemma *fps-binomial-ODE-unique*:
fixes *c* :: *'a*::*field-char-0*
shows $fps\text{-}deriv\ a = (fps\text{-}const\ c * a)\ /\ (1 + X) \longleftrightarrow a = fps\text{-}const\ (a\$0) * fps\text{-}binomial\ c$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *fps-binomial-deriv*: $fps\text{-}deriv\ (fps\text{-}binomial\ c) = fps\text{-}const\ c * fps\text{-}binomial\ c\ /\ (1 + X)$
⟨*proof*⟩

lemma *fps-binomial-add-mult*: $fps\text{-}binomial\ (c+d) = fps\text{-}binomial\ c * fps\text{-}binomial\ d$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-minomial-minus-one*: $fps\text{-}binomial\ (-1) = inverse\ (1 + X)$
(**is** *?l* = *inverse ?r*)
⟨*proof*⟩

Vandermonde's Identity as a consequence

lemma *gbinomial-Vandermonde*: $setsum\ (\lambda k.\ (a\ gchoose\ k) * (b\ gchoose\ (n - k)))\ \{0..n\} = (a + b)\ gchoose\ n$
⟨*proof*⟩

lemma *binomial-Vandermonde*: $setsum\ (\lambda k.\ (a\ choose\ k) * (b\ choose\ (n - k)))\ \{0..n\} = (a + b)\ choose\ n$
⟨*proof*⟩

lemma *binomial-Vandermonde-same*: $setsum\ (\lambda k.\ (n\ choose\ k)\ \hat{}\ 2)\ \{0..n\} = (2*n)\ choose\ n$
⟨*proof*⟩

lemma *Vandermonde-pochhammer-lemma*:
fixes *a* :: *'a*::*field-char-0*
assumes *b*: $\forall\ j \in \{0\ ..<n\}.\ b \neq of\text{-}nat\ j$
shows $setsum\ (\%k.\ (pochhammer\ (-a)\ k * pochhammer\ (-(of\text{-}nat\ n))\ k)\ /\ (of\text{-}nat\ (fact\ k) * pochhammer\ (b - of\text{-}nat\ n + 1)\ k))\ \{0..n\} = pochhammer\ (-$

$(a+\ b))\ n\ /\ pochhammer\ (-\ b)\ n$ (**is** *?l = ?r*)
⟨*proof*⟩

lemma *Vandermonde-pochhammer*:
fixes *a* :: *'a*::*field-char-0*
assumes *c*: *ALL i* : {*0*..< *n*}. *c* ≠ − *of-nat i*
shows *setsum* (%*k*. (*pochhammer a k* ∗ *pochhammer* (− (*of-nat n*)) *k*) / (*of-nat* (*fact k*) ∗ *pochhammer c k*)) {*0*..*n*} = *pochhammer* (*c* − *a*) *n* / *pochhammer c n*
⟨*proof*⟩

39.18.4 Formal trigonometric functions

definition *fps-sin* (*c*::*'a*::*field-char-0*) =
Abs-fps (λ*n*. *if even n then 0 else* (− *1*) ^((*n* − *1*) *div 2*) ∗ *c*^*n* /(*of-nat* (*fact n*)))

definition *fps-cos* (*c*::*'a*::*field-char-0*) =
Abs-fps (λ*n*. *if even n then* (− *1*) ^ (*n div 2*) ∗ *c*^*n* / (*of-nat* (*fact n*)) *else 0*)

lemma *fps-sin-deriv*:
fps-deriv (*fps-sin c*) = *fps-const c* ∗ *fps-cos c*
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *fps-cos-deriv*:
fps-deriv (*fps-cos c*) = *fps-const* (− *c*)∗ (*fps-sin c*)
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *fps-sin-cos-sum-of-squares*:
fps-cos c ^ *2* + *fps-sin c* ^ *2* = *1* (**is** *?lhs = 1*)
⟨*proof*⟩

lemma *divide-eq-iff*: *a* ≠ (*0*::*'a*::*field*) ⟹ *x* / *a* = *y* ⟷ *x* = *y* ∗ *a*
⟨*proof*⟩

lemma *eq-divide-iff*: *a* ≠ (*0*::*'a*::*field*) ⟹ *x* = *y* / *a* ⟷ *x* ∗ *a* = *y*
⟨*proof*⟩

lemma *fps-sin-nth-0* [*simp*]: *fps-sin c* $ *0* = *0*
⟨*proof*⟩

lemma *fps-sin-nth-1* [*simp*]: *fps-sin c* $ *1* = *c*
⟨*proof*⟩

lemma *fps-sin-nth-add-2*:
fps-sin c $ (*n* + *2*) = − (*c* ∗ *c* ∗ *fps-sin c* $ *n* / (*of-nat*(*n*+*1*) ∗ *of-nat*(*n*+*2*)))
⟨*proof*⟩

lemma *fps-cos-nth-0* [*simp*]: *fps-cos c $ 0 = 1*
⟨*proof*⟩

lemma *fps-cos-nth-1* [*simp*]: *fps-cos c $ 1 = 0*
⟨*proof*⟩

lemma *fps-cos-nth-add-2*:
*fps-cos c $ (n + 2) = − (c * c * fps-cos c $ n / (of-nat(n+1) * of-nat(n+2)))*
⟨*proof*⟩

lemma *nat-induct2*:
⟦P 0; P 1; ⋀n. P n ⟹ P (n + 2)⟧ ⟹ P (n::nat)
⟨*proof*⟩

lemma *nat-add-1-add-1*: *(n::nat) + 1 + 1 = n + 2*
⟨*proof*⟩

lemma *eq-fps-sin*:
assumes *0*: *a $ 0 = 0* **and** *1*: *a $ 1 = c*
and *2*: *fps-deriv (fps-deriv a) = − (fps-const c * fps-const c * a)*
shows *a = fps-sin c*
⟨*proof*⟩

lemma *eq-fps-cos*:
assumes *0*: *a $ 0 = 1* **and** *1*: *a $ 1 = 0*
and *2*: *fps-deriv (fps-deriv a) = − (fps-const c * fps-const c * a)*
shows *a = fps-cos c*
⟨*proof*⟩

lemma *mult-nth-0* [*simp*]: *(a * b) $ 0 = a $ 0 * b $ 0*
⟨*proof*⟩

lemma *mult-nth-1* [*simp*]: *(a * b) $ 1 = a $ 0 * b $ 1 + a $ 1 * b $ 0*
⟨*proof*⟩

lemma *fps-sin-add*:
*fps-sin (a + b) = fps-sin a * fps-cos b + fps-cos a * fps-sin b*
⟨*proof*⟩

lemma *fps-cos-add*:
*fps-cos (a + b) = fps-cos a * fps-cos b − fps-sin a * fps-sin b*
⟨*proof*⟩

lemma *fps-sin-even*: *fps-sin (− c) = − fps-sin c*
⟨*proof*⟩

lemma *fps-cos-odd*: *fps-cos (− c) = fps-cos c*
⟨*proof*⟩

definition *fps-tan c = fps-sin c / fps-cos c*

lemma *fps-tan-deriv*: *fps-deriv(fps-tan c) = fps-const c/ (fps-cos c ^ 2)*
⟨*proof*⟩

Connection to E c over the complex numbers — Euler and De Moivre

lemma *Eii-sin-cos*:
*E (ii * c) = fps-cos c + fps-const ii * fps-sin c*
(**is** *?l = ?r*)
⟨*proof*⟩

lemma *E-minus-ii-sin-cos*: *E (- (ii * c)) = fps-cos c - fps-const ii * fps-sin c*
⟨*proof*⟩

lemma *fps-const-minus*: *fps-const (c::'a::group-add) - fps-const d = fps-const (c - d)*
⟨*proof*⟩

lemma *fps-number-of-fps-const*: *number-of i = fps-const (number-of i :: 'a:: {comm-ring-1, number-ring})*
⟨*proof*⟩

lemma *fps-cos-Eii*:
*fps-cos c = (E (ii * c) + E (- ii * c)) / fps-const 2*
⟨*proof*⟩

lemma *fps-sin-Eii*:
*fps-sin c = (E (ii * c) - E (- ii * c)) / fps-const (2*ii)*
⟨*proof*⟩

lemma *fps-tan-Eii*:
*fps-tan c = (E (ii * c) - E (- ii * c)) / (fps-const ii * (E (ii * c) + E (- ii * c)))*
⟨*proof*⟩

lemma *fps-demoivre*: *(fps-cos a + fps-const ii * fps-sin a) ^n = fps-cos (of-nat n * a) + fps-const ii * fps-sin (of-nat n * a)*
⟨*proof*⟩

39.19 Hypergeometric series

definition *F as bs (c::'a::{field-char-0, field-inverse-zero}) = Abs-fps (%n. (foldl (%r a. r* pochhammer a n) 1 as * c^n)/ (foldl (%r b. r * pochhammer b n) 1 bs * of-nat (fact n)))*

lemma *F-nth[simp]*: *F as bs c $ n = (foldl (%r a. r* pochhammer a n) 1 as * c^n)/ (foldl (%r b. r * pochhammer b n) 1 bs * of-nat (fact n))*
⟨*proof*⟩

lemma *foldl-mult-start*:
foldl (%*r x*. *r* ∗ *f x*) (*v*::′*a*::*comm-ring-1*) *as* ∗ *x* = *foldl* (%*r x*. *r* ∗ *f x*) (*v* ∗ *x*) *as*
⟨*proof*⟩

lemma *foldr-mult-foldl*: *foldr* (%*x r*. *r* ∗ *f x*) *as v* = *foldl* (%*r x*. *r* ∗ *f x*) (*v* :: ′*a*::*comm-ring-1*) *as*
⟨*proof*⟩

lemma *F-nth-alt*: *F as bs c* $ *n* = *foldr* (λ*a r*. *r* ∗ *pochhammer a n*) *as* (*c* ^ *n*) / *foldr* (λ*b r*. *r* ∗ *pochhammer b n*) *bs* (*of-nat* (*fact n*))
⟨*proof*⟩

lemma *F-E*[*simp*]: *F* [] [] *c* = *E c*
⟨*proof*⟩

lemma *F-1-0*[*simp*]: *F* [*1*] [] *c* = *1*/(*1* − *fps-const c* ∗ *X*)
⟨*proof*⟩

lemma *F-B*[*simp*]: *F* [−*a*] [] (− *1*) = *fps-binomial a*
⟨*proof*⟩

lemma *F-0*[*simp*]: *F as bs c* $*0* = *1*
⟨*proof*⟩

lemma *foldl-prod-prod*: *foldl* (%(*r*::′*b*::*comm-ring-1*) (*x*::′*a*::*comm-ring-1*). *r* ∗ *f x*) *v as* ∗ *foldl* (%*r x*. *r* ∗ *g x*) *w as* = *foldl* (%*r x*. *r* ∗ *f x* ∗ *g x*) (*v*∗*w*) *as*
⟨*proof*⟩

lemma *F-rec*: *F as bs c* $ *Suc n* = ((*foldl* (%*r a*. *r*∗ (*a* + *of-nat n*)) *c as*)/ (*foldl* (%*r b*. *r* ∗ (*b* + *of-nat n*)) (*of-nat* (*Suc n*)) *bs*)) ∗ *F as bs c* $ *n*
⟨*proof*⟩

lemma *XD-nth*[*simp*]: *XD a* $ *n* = (*if n=0 then 0 else of-nat n* ∗ *a*$*n*)
⟨*proof*⟩

lemma *XD-0th*[*simp*]: *XD a* $ *0* = *0* ⟨*proof*⟩
lemma *XD-Suc*[*simp*]: *XD a* $ *Suc n* = *of-nat* (*Suc n*) ∗ *a* $ *Suc n* ⟨*proof*⟩

definition *XDp c a* = *XD a* + *fps-const c* ∗ *a*

lemma *XDp-nth*[*simp*]: *XDp c a* $ *n* = (*c* + *of-nat n*) ∗ *a*$*n*
⟨*proof*⟩

lemma *XDp-commute*:
shows *XDp b o XDp* (*c*::′*a*::*comm-ring-1*) = *XDp c o XDp b*
⟨*proof*⟩

lemma *XDp0*[*simp*]: *XDp 0 = XD*
⟨*proof*⟩

lemma *XDp-fps-integral*[*simp*]:*XDp 0* (*fps-integral a c*) = *X* ∗ *a*
⟨*proof*⟩

lemma *F-minus-nat*:
F [− *of-nat n*] [− *of-nat* (*n* + *m*)] (*c*::′*a*::{*field-char-0*, *field-inverse-zero*}) $ *k*
= (*if k* <= *n then pochhammer* (− *of-nat n*) *k* ∗ *c* ^ *k* /
(*pochhammer* (− *of-nat* (*n* + *m*)) *k* ∗ *of-nat* (*fact k*)) *else 0*)
F [− *of-nat m*] [− *of-nat* (*m* + *n*)] (*c*::′*a*::{*field-char-0*, *field-inverse-zero*}) $ *k*
= (*if k* <= *m then pochhammer* (− *of-nat m*) *k* ∗ *c* ^ *k* /
(*pochhammer* (− *of-nat* (*m* + *n*)) *k* ∗ *of-nat* (*fact k*)) *else 0*)
⟨*proof*⟩

lemma *setsum-eq-if*: *setsum f* {(*n*::*nat*) .. *m*} = (*if m* < *n then 0 else f n* +
setsum f {*n*+*1* .. *m*})
⟨*proof*⟩

lemma *pochhammer-rec-if*:
pochhammer a n = (*if n* = *0 then 1 else a* ∗ *pochhammer* (*a* + *1*) (*n* − *1*))
⟨*proof*⟩

lemma *XDp-foldr-nth*[*simp*]: *foldr* (%*c r*. *XDp c o r*) *cs* (%*c*. *XDp c a*) *c0* $ *n* =

foldr (%*c r*. (*c* + *of-nat n*) ∗ *r*) *cs* (*c0* + *of-nat n*) ∗ *a*$*n*
⟨*proof*⟩

lemma *genric-XDp-foldr-nth*:
assumes
f: *ALL n c a*. *f c a* $ *n* = (*of-nat n* + *k c*) ∗ *a*$*n*

shows *foldr* (%*c r*. *f c o r*) *cs* (%*c*. *g c a*) *c0* $ *n* =
foldr (%*c r*. (*k c* + *of-nat n*) ∗ *r*) *cs* (*g c0 a* $ *n*)
⟨*proof*⟩

end

40 Fraction-Field: A formalization of the fraction field of any integral domain A generalization of Rat.thy from int to any integral domain

theory *Fraction-Field*
imports *Main*
begin

40.1 General fractions construction

40.1.1 Construction of the type of fractions

definition *fractrel* :: $((('a::idom * 'a) * ('a * 'a))\ set$ **where**
fractrel == $\{(x, y).\ snd\ x \neq 0 \wedge snd\ y \neq 0 \wedge fst\ x * snd\ y = fst\ y * snd\ x\}$

lemma *fractrel-iff* [*simp*]:
$(x, y) \in fractrel \longleftrightarrow snd\ x \neq 0 \wedge snd\ y \neq 0 \wedge fst\ x * snd\ y = fst\ y * snd\ x$
⟨*proof*⟩

lemma *refl-fractrel*: *refl-on* $\{x.\ snd\ x \neq 0\}$ *fractrel*
⟨*proof*⟩

lemma *sym-fractrel*: *sym fractrel*
⟨*proof*⟩

lemma *trans-fractrel*: *trans fractrel*
⟨*proof*⟩

lemma *equiv-fractrel*: *equiv* $\{x.\ snd\ x \neq 0\}$ *fractrel*
⟨*proof*⟩

lemmas *UN-fractrel* = *UN-equiv-class* [*OF equiv-fractrel*]
lemmas *UN-fractrel2* = *UN-equiv-class2* [*OF equiv-fractrel equiv-fractrel*]

lemma *equiv-fractrel-iff* [*iff*]:
assumes *snd x* ≠ *0* **and** *snd y* ≠ *0*
shows *fractrel* `` $\{x\}$ = *fractrel* `` $\{y\} \longleftrightarrow (x, y) \in fractrel$
⟨*proof*⟩

typedef *'a fract* = $\{(x::'a \times 'a).\ snd\ x \neq (0::'a::idom)\}$ // *fractrel*
⟨*proof*⟩

lemma *fractrel-in-fract* [*simp*]: *snd x* ≠ *0* ⟹ *fractrel* `` $\{x\} \in fract$
⟨*proof*⟩

declare *Abs-fract-inject* [*simp*] *Abs-fract-inverse* [*simp*]

40.1.2 Representation and basic operations

definition
Fract :: *'a*::*idom* ⇒ *'a* ⇒ *'a fract* **where**
[*code del*]: *Fract a b* = *Abs-fract* (*fractrel* `` $\{$*if b* = *0 then* (*0*, *1*) *else* (*a*, *b*)$\}$)

code-datatype *Fract*

lemma *Fract-cases* [*case-names Fract*, *cases type*: *fract*]:
assumes $\bigwedge a\ b.\ q = Fract\ a\ b \Longrightarrow b \neq 0 \Longrightarrow C$
shows *C*

⟨*proof*⟩

lemma *Fract-induct* [*case-names Fract*, *induct type*: *fract*]:
assumes $\bigwedge a\ b.\ b \neq 0 \Longrightarrow P\ (Fract\ a\ b)$
shows $P\ q$
⟨*proof*⟩

lemma *eq-fract*:
shows $\bigwedge a\ b\ c\ d.\ b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow Fract\ a\ b = Fract\ c\ d \longleftrightarrow a * d = c * b$
and $\bigwedge a.\ Fract\ a\ 0 = Fract\ 0\ 1$
and $\bigwedge a\ c.\ Fract\ 0\ a = Fract\ 0\ c$
⟨*proof*⟩

instantiation *fract* :: (*idom*) {*comm-ring-1*, *power*}
begin

definition
Zero-fract-def [*code*, *code-unfold*]: $0 = Fract\ 0\ 1$

definition
One-fract-def [*code*, *code-unfold*]: $1 = Fract\ 1\ 1$

definition
add-fract-def [*code del*]:
$q + r = Abs\text{-}fract\ (\bigcup x \in Rep\text{-}fract\ q.\ \bigcup y \in Rep\text{-}fract\ r.$
$fractrel\ ``\ \{(fst\ x * snd\ y + fst\ y * snd\ x,\ snd\ x * snd\ y)\})$

lemma *add-fract* [*simp*]:
assumes $b \neq (0::'a::idom)$ **and** $d \neq 0$
shows $Fract\ a\ b + Fract\ c\ d = Fract\ (a * d + c * b)\ (b * d)$
⟨*proof*⟩

definition
minus-fract-def [*code del*]:
$-\ q = Abs\text{-}fract\ (\bigcup x \in Rep\text{-}fract\ q.\ fractrel\ ``\ \{(-\ fst\ x,\ snd\ x)\})$

lemma *minus-fract* [*simp*, *code*]: $-\ Fract\ a\ b = Fract\ (-\ a)\ (b::'a::idom)$
⟨*proof*⟩

lemma *minus-fract-cancel* [*simp*]: $Fract\ (-\ a)\ (-\ b) = Fract\ a\ b$
⟨*proof*⟩

definition
diff-fract-def [*code del*]: $q - r = q + -\ (r::'a\ fract)$

lemma *diff-fract* [*simp*]:
assumes $b \neq 0$ **and** $d \neq 0$
shows $Fract\ a\ b - Fract\ c\ d = Fract\ (a * d - c * b)\ (b * d)$
⟨*proof*⟩

definition
mult-fract-def [*code del*]:
$q * r = Abs\text{-}fract\ (\bigcup x \in Rep\text{-}fract\ q.\ \bigcup y \in Rep\text{-}fract\ r.$
$fractrel``\{(fst\ x * fst\ y,\ snd\ x * snd\ y)\})$

lemma *mult-fract* [*simp*]: $Fract\ (a::'a::idom)\ b * Fract\ c\ d = Fract\ (a * c)\ (b * d)$
⟨*proof*⟩

lemma *mult-fract-cancel*:
assumes $c \neq 0$
shows $Fract\ (c * a)\ (c * b) = Fract\ a\ b$
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *of-nat-fract*: $of\text{-}nat\ k = Fract\ (of\text{-}nat\ k)\ 1$
⟨*proof*⟩

lemma *Fract-of-nat-eq*: $Fract\ (of\text{-}nat\ k)\ 1 = of\text{-}nat\ k$
⟨*proof*⟩

lemma *fract-collapse* [*code-post*]:
$Fract\ 0\ k = 0$
$Fract\ 1\ 1 = 1$
$Fract\ k\ 0 = 0$
⟨*proof*⟩

lemma *fract-expand* [*code-unfold*]:
$0 = Fract\ 0\ 1$
$1 = Fract\ 1\ 1$
⟨*proof*⟩

lemma *Fract-cases-nonzero* [*case-names Fract 0*]:
assumes *Fract*: $\bigwedge a\ b.\ q = Fract\ a\ b \Longrightarrow b \neq 0 \Longrightarrow a \neq 0 \Longrightarrow C$
assumes *0*: $q = 0 \Longrightarrow C$
shows C
⟨*proof*⟩

40.1.3 The field of rational numbers

context *idom*
begin
subclass *ring-no-zero-divisors* ⟨*proof*⟩
thm *mult-eq-0-iff*
end

instantiation *fract* :: (*idom*) *field-inverse-zero*
begin

definition
inverse-fract-def [*code del*]:
inverse q = *Abs-fract* ($\bigcup$ *x* ∈ *Rep-fract q*.
fractrel `` {*if fst x* = *0 then* (*0*, *1*) *else* (*snd x*, *fst x*)})

lemma *inverse-fract* [*simp*]: *inverse* (*Fract a b*) = *Fract* (*b*::'*a*::*idom*) *a*
⟨*proof*⟩

definition
divide-fract-def [*code del*]: *q* / *r* = *q* ∗ *inverse* (*r*:: '*a fract*)

lemma *divide-fract* [*simp*]: *Fract a b* / *Fract c d* = *Fract* (*a* ∗ *d*) (*b* ∗ *c*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

40.1.4 The ordered field of fractions over an ordered idom

lemma *le-congruent2*:
(λ*x y*::'*a* × '*a*::*linordered-idom*.
{(*fst x* ∗ *snd y*)∗(*snd x* ∗ *snd y*) ≤ (*fst y* ∗ *snd x*)∗(*snd x* ∗ *snd y*)})
respects2 fractrel
⟨*proof*⟩

instantiation *fract* :: (*linordered-idom*) *linorder*
begin

definition
le-fract-def [*code del*]:
q ≤ *r* ⟷ *contents* ($\bigcup$ *x* ∈ *Rep-fract q*. $\bigcup$ *y* ∈ *Rep-fract r*.
{(*fst x* ∗ *snd y*)∗(*snd x* ∗ *snd y*) ≤ (*fst y* ∗ *snd x*)∗(*snd x* ∗ *snd y*)})

definition
less-fract-def [*code del*]: *z* < (*w*::'*a fract*) ⟷ *z* ≤ *w* ∧ ¬ *w* ≤ *z*

lemma *le-fract* [*simp*]:
assumes *b* ≠ *0* **and** *d* ≠ *0*
shows *Fract a b* ≤ *Fract c d* ⟷ (*a* ∗ *d*) ∗ (*b* ∗ *d*) ≤ (*c* ∗ *b*) ∗ (*b* ∗ *d*)
⟨*proof*⟩

lemma *less-fract* [*simp*]:
assumes *b* ≠ *0* **and** *d* ≠ *0*

shows $Fract\ a\ b < Fract\ c\ d \longleftrightarrow (a * d) * (b * d) < (c * b) * (b * d)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *fract* :: (*linordered-idom*) {*distrib-lattice*, *abs-if*, *sgn-if*}
begin

definition
abs-fract-def: $|q| = (if\ q < 0\ then\ -q\ else\ (q::'a\ fract))$

definition
sgn-fract-def:
$sgn\ (q::'a\ fract) = (if\ q=0\ then\ 0\ else\ if\ 0<q\ then\ 1\ else\ -\ 1)$

theorem *abs-fract* [*simp*]: $|Fract\ a\ b| = Fract\ |a|\ |b|$
⟨*proof*⟩

definition
inf-fract-def:
$(inf :: 'a\ fract \Rightarrow 'a\ fract \Rightarrow 'a\ fract) = min$

definition
sup-fract-def:
$(sup :: 'a\ fract \Rightarrow 'a\ fract \Rightarrow 'a\ fract) = max$

instance ⟨*proof*⟩

end

instance *fract* :: (*linordered-idom*) *linordered-field-inverse-zero*
⟨*proof*⟩

lemma *fract-induct-pos* [*case-names Fract*]:
fixes $P :: 'a::linordered\text{-}idom\ fract \Rightarrow bool$
assumes *step*: $\bigwedge a\ b.\ 0 < b \Longrightarrow P\ (Fract\ a\ b)$
shows $P\ q$
⟨*proof*⟩

lemma *zero-less-Fract-iff*:
$0 < b \Longrightarrow 0 < Fract\ a\ b \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *Fract-less-zero-iff*:
$0 < b \Longrightarrow Fract\ a\ b < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *zero-le-Fract-iff*:
$0 < b \Longrightarrow 0 \leq Fract\ a\ b \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *Fract-le-zero-iff*:
$0 < b \Longrightarrow Fract\ a\ b \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *one-less-Fract-iff*:
$0 < b \Longrightarrow 1 < Fract\ a\ b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *Fract-less-one-iff*:
$0 < b \Longrightarrow Fract\ a\ b < 1 \longleftrightarrow a < b$
⟨*proof*⟩

lemma *one-le-Fract-iff*:
$0 < b \Longrightarrow 1 \leq Fract\ a\ b \longleftrightarrow b \leq a$
⟨*proof*⟩

lemma *Fract-le-one-iff*:
$0 < b \Longrightarrow Fract\ a\ b \leq 1 \longleftrightarrow a \leq b$
⟨*proof*⟩

end

41 FuncSet: Pi and Function Sets

theory *FuncSet*
imports *Hilbert-Choice Main*
begin

definition
$Pi :: ['a\ set, 'a => 'b\ set] => ('a => 'b)\ set$ **where**
$Pi\ A\ B = \{f.\ \forall x.\ x \in A \dashrightarrow f\ x \in B\ x\}$

definition
$extensional :: 'a\ set => ('a => 'b)\ set$ **where**
$extensional\ A = \{f.\ \forall x.\ x^{\sim}{:}A \dashrightarrow f\ x = undefined\}$

definition
$restrict :: ['a => 'b, 'a\ set] => ('a => 'b)$ **where**
$restrict\ f\ A = (\%x.\ if\ x \in A\ then\ f\ x\ else\ undefined)$

abbreviation
$funcset :: ['a\ set, 'b\ set] => ('a => 'b)\ set$
(**infixr** $->$ *60*) **where**
$A -> B == Pi\ A\ (\%\text{-}.\ B)$

notation (*xsymbols*)
funcset (**infixr** $\rightarrow$ *60*)

syntax
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a* => *'b*) *set* ((*3PI -:-./ -*) *10*)
-lam :: [*pttrn*, *'a set*, *'a* => *'b*] => (*'a*=>*'b*) ((*3%-:-./ -*) [*0*,*0*,*3*] *3*)

syntax (*xsymbols*)
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a* => *'b*) *set* ((*3*Π *-*$\in$*-./ -*) *10*)
-lam :: [*pttrn*, *'a set*, *'a* => *'b*] => (*'a*=>*'b*) ((*3*λ*-*$\in$*-./ -*) [*0*,*0*,*3*] *3*)

syntax (*HTML* **output**)
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a* => *'b*) *set* ((*3*Π *-*$\in$*-./ -*) *10*)
-lam :: [*pttrn*, *'a set*, *'a* => *'b*] => (*'a*=>*'b*) ((*3*λ*-*$\in$*-./ -*) [*0*,*0*,*3*] *3*)

translations
PI x:*A*. *B* == *CONST Pi A* (%*x*. *B*)
%*x*:*A*. *f* == *CONST restrict* (%*x*. *f*) *A*

definition
compose :: [*'a set*, *'b* => *'c*, *'a* => *'b*] => (*'a* => *'c*) **where**
compose A g f = (λ*x*$\in$*A*. *g* (*f x*))

41.1 Basic Properties of *Pi*

lemma *Pi-I*[*intro*!]: (!!*x*. *x* $\in$ *A* ==> *f x* $\in$ *B x*) ==> *f* $\in$ *Pi A B*
⟨*proof*⟩

lemma *Pi-I'*[*simp*]: (!!*x*. *x* : *A* --> *f x* : *B x*) ==> *f* : *Pi A B*
⟨*proof*⟩

lemma *funcsetI*: (!!*x*. *x* $\in$ *A* ==> *f x* $\in$ *B*) ==> *f* $\in$ *A* -> *B*
⟨*proof*⟩

lemma *Pi-mem*: [|*f*: *Pi A B*; *x* $\in$ *A*|] ==> *f x* $\in$ *B x*
⟨*proof*⟩

lemma *PiE* [*elim*]:
f : *Pi A B* ==> (*f x* : *B x* ==> *Q*) ==> (*x* ~: *A* ==> *Q*) ==> *Q*
⟨*proof*⟩

lemma *funcset-id* [*simp*]: (λ*x*. *x*) $\in$ *A* $\rightarrow$ *A*
⟨*proof*⟩

lemma *funcset-mem*: [|*f* $\in$ *A* -> *B*; *x* $\in$ *A*|] ==> *f x* $\in$ *B*
⟨*proof*⟩

lemma *funcset-image*: *f* $\in$ *A*$\rightarrow$*B* ==> *f* ‘ *A* $\subseteq$ *B*

⟨*proof*⟩

lemma *Pi-eq-empty*[*simp*]: $((PI\ x{:}\ A.\ B\ x) = \{\}) = (\exists\, x{\in}A.\ B(x) = \{\})$
⟨*proof*⟩

lemma *Pi-empty* [*simp*]: $Pi\ \{\}\ B = UNIV$
⟨*proof*⟩

lemma *Pi-UNIV* [*simp*]: $A \mathrel{-}> UNIV = UNIV$
⟨*proof*⟩

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(!!x.\ x \in A \Longrightarrow B\ x <= C\ x) \Longrightarrow Pi\ A\ B <= Pi\ A\ C$
⟨*proof*⟩

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' <= A \Longrightarrow Pi\ A\ B <= Pi\ A'\ B$
⟨*proof*⟩

lemma *prod-final*:
assumes *1*: $fst \circ f \in Pi\ A\ B$ **and** *2*: $snd \circ f \in Pi\ A\ C$
shows $f \in (\Pi\ z \in A.\ B\ z \times C\ z)$
⟨*proof*⟩

41.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*:
$[|\ f \in A \mathrel{-}> B;\ g \in B \mathrel{-}> C\ |] \Longrightarrow compose\ A\ g\ f \in A \mathrel{-}> C$
⟨*proof*⟩

lemma *compose-assoc*:
$[|\ f \in A \mathrel{-}> B;\ g \in B \mathrel{-}> C;\ h \in C \mathrel{-}> D\ |]$
$\Longrightarrow compose\ A\ h\ (compose\ A\ g\ f) = compose\ A\ (compose\ B\ h\ g)\ f$
⟨*proof*⟩

lemma *compose-eq*: $x \in A \Longrightarrow compose\ A\ g\ f\ x = g(f(x))$
⟨*proof*⟩

lemma *surj-compose*: $[|\ f\ `\ A = B;\ g\ `\ B = C\ |] \Longrightarrow compose\ A\ g\ f\ `\ A = C$
⟨*proof*⟩

41.3 Bounded Abstraction: *restrict*

lemma *restrict-in-funcset*: $(!!x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in A \mathrel{-}> B$
⟨*proof*⟩

lemma *restrictI*[*intro!*]: $(!!x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in Pi\ A\ B$
⟨*proof*⟩

lemma *restrict-apply* [*simp*]:
(λ*y*∈*A*. *f y*) *x* = (*if x* ∈ *A then f x else undefined*)
⟨*proof*⟩

lemma *restrict-ext*:
(!!*x*. *x* ∈ *A* ==> *f x* = *g x*) ==> (λ*x*∈*A*. *f x*) = (λ*x*∈*A*. *g x*)
⟨*proof*⟩

lemma *inj-on-restrict-eq* [*simp*]: *inj-on* (*restrict f A*) *A* = *inj-on f A*
⟨*proof*⟩

lemma *Id-compose*:
[|*f* ∈ *A* -> *B*; *f* ∈ *extensional A*|] ==> *compose A* (λ*y*∈*B*. *y*) *f* = *f*
⟨*proof*⟩

lemma *compose-Id*:
[|*g* ∈ *A* -> *B*; *g* ∈ *extensional A*|] ==> *compose A g* (λ*x*∈*A*. *x*) = *g*
⟨*proof*⟩

lemma *image-restrict-eq* [*simp*]: (*restrict f A*) ‘ *A* = *f* ‘ *A*
⟨*proof*⟩

41.4 Bijections Between Sets

The definition of *bij-betw* is in *Fun.thy*, but most of the theorems belong here, or need at least *Hilbert-Choice*.

lemma *bij-betw-imp-funcset*: *bij-betw f A B* ⟹ *f* ∈ *A* → *B*
⟨*proof*⟩

lemma *inj-on-compose*:
[| *bij-betw f A B*; *inj-on g B* |] ==> *inj-on* (*compose A g f*) *A*
⟨*proof*⟩

lemma *bij-betw-compose*:
[| *bij-betw f A B*; *bij-betw g B C* |] ==> *bij-betw* (*compose A g f*) *A C*
⟨*proof*⟩

lemma *bij-betw-restrict-eq* [*simp*]:
bij-betw (*restrict f A*) *A B* = *bij-betw f A B*
⟨*proof*⟩

41.5 Extensionality

lemma *extensional-arb*: [|*f* ∈ *extensional A*; *x*∉ *A*|] ==> *f x* = *undefined*
⟨*proof*⟩

lemma *restrict-extensional* [*simp*]: *restrict f A* ∈ *extensional A*
⟨*proof*⟩

lemma *compose-extensional* [*simp*]: *compose A f g* ∈ *extensional A*
⟨*proof*⟩

lemma *extensionalityI*:
[| *f* ∈ *extensional A*; *g* ∈ *extensional A*;
!!*x*. *x*∈*A* ==> *f x* = *g x* |] ==> *f* = *g*
⟨*proof*⟩

lemma *inv-into-funcset*: *f* ' *A* = *B* ==> (λ*x*∈*B*. *inv-into A f x*) : *B* -> *A*
⟨*proof*⟩

lemma *compose-inv-into-id*:
bij-betw f A B ==> *compose A* (λ*y*∈*B*. *inv-into A f y*) *f* = (λ*x*∈*A*. *x*)
⟨*proof*⟩

lemma *compose-id-inv-into*:
f ' *A* = *B* ==> *compose B f* (λ*y*∈*B*. *inv-into A f y*) = (λ*x*∈*B*. *x*)
⟨*proof*⟩

41.6 Cardinality

lemma *card-inj*: [|*f* ∈ *A*→*B*; *inj-on f A*; *finite B*|] ==> *card*(*A*) ≤ *card*(*B*)
⟨*proof*⟩

lemma *card-bij*:
[|*f* ∈ *A*→*B*; *inj-on f A*;
g ∈ *B*→*A*; *inj-on g B*; *finite A*; *finite B*|] ==> *card*(*A*) = *card*(*B*)
⟨*proof*⟩

end

42 Polynomial: Univariate Polynomials

theory *Polynomial*
imports *Main*
begin

42.1 Definition of type *poly*

typedef (*Poly*) *'a poly* = {*f*::*nat* ⇒ *'a*::*zero*. ∃ *n*. ∀ *i*>*n*. *f i* = *0*}
morphisms *coeff Abs-poly*
⟨*proof*⟩

lemma *expand-poly-eq*: *p* = *q* ⟷ (∀ *n*. *coeff p n* = *coeff q n*)
⟨*proof*⟩

lemma *poly-ext*: (⋀*n*. *coeff p n* = *coeff q n*) ⟹ *p* = *q*
⟨*proof*⟩

42.2 Degree of a polynomial

definition
degree :: *'a*::*zero poly* ⇒ *nat* **where**
degree p = (*LEAST n*. ∀ *i*>*n*. *coeff p i* = *0*)

lemma *coeff-eq-0*: *degree p* < *n* ⟹ *coeff p n* = *0*
⟨*proof*⟩

lemma *le-degree*: *coeff p n* ≠ *0* ⟹ *n* ≤ *degree p*
⟨*proof*⟩

lemma *degree-le*: ∀ *i*>*n*. *coeff p i* = *0* ⟹ *degree p* ≤ *n*
⟨*proof*⟩

lemma *less-degree-imp*: *n* < *degree p* ⟹ ∃ *i*>*n*. *coeff p i* ≠ *0*
⟨*proof*⟩

42.3 The zero polynomial

instantiation *poly* :: (*zero*) *zero*
begin

definition
zero-poly-def: *0* = *Abs-poly* (λ*n*. *0*)

instance ⟨*proof*⟩
end

lemma *coeff-0* [*simp*]: *coeff 0 n* = *0*
⟨*proof*⟩

lemma *degree-0* [*simp*]: *degree 0* = *0*
⟨*proof*⟩

lemma *leading-coeff-neq-0*:
assumes *p* ≠ *0* **shows** *coeff p* (*degree p*) ≠ *0*
⟨*proof*⟩

lemma *leading-coeff-0-iff* [*simp*]: *coeff p* (*degree p*) = *0* ⟷ *p* = *0*
⟨*proof*⟩

42.4 List-style constructor for polynomials

definition
pCons :: *'a*::*zero* ⇒ *'a poly* ⇒ *'a poly*
where
[*code del*]: *pCons a p* = *Abs-poly* (*nat-case a* (*coeff p*))

syntax

-poly :: *args* ⇒ *'a poly* ([:(-):])

translations
[:*x*, *xs*:] == *CONST pCons x* [:*xs*:]
[:*x*:] == *CONST pCons x 0*
[:*x*:] <= *CONST pCons x* (*-constrain 0 t*)

lemma *Poly-nat-case*: *f* ∈ *Poly* ⟹ *nat-case a f* ∈ *Poly*
⟨*proof*⟩

lemma *coeff-pCons*:
coeff (*pCons a p*) = *nat-case a* (*coeff p*)
⟨*proof*⟩

lemma *coeff-pCons-0* [*simp*]: *coeff* (*pCons a p*) *0* = *a*
⟨*proof*⟩

lemma *coeff-pCons-Suc* [*simp*]: *coeff* (*pCons a p*) (*Suc n*) = *coeff p n*
⟨*proof*⟩

lemma *degree-pCons-le*: *degree* (*pCons a p*) ≤ *Suc* (*degree p*)
⟨*proof*⟩

lemma *degree-pCons-eq*:
p ≠ *0* ⟹ *degree* (*pCons a p*) = *Suc* (*degree p*)
⟨*proof*⟩

lemma *degree-pCons-0*: *degree* (*pCons a 0*) = *0*
⟨*proof*⟩

lemma *degree-pCons-eq-if* [*simp*]:
degree (*pCons a p*) = (*if p* = *0 then 0 else Suc* (*degree p*))
⟨*proof*⟩

lemma *pCons-0-0* [*simp*]: *pCons 0 0* = *0*
⟨*proof*⟩

lemma *pCons-eq-iff* [*simp*]:
pCons a p = *pCons b q* ⟷ *a* = *b* ∧ *p* = *q*
⟨*proof*⟩

lemma *pCons-eq-0-iff* [*simp*]: *pCons a p* = *0* ⟷ *a* = *0* ∧ *p* = *0*
⟨*proof*⟩

lemma *Poly-Suc*: *f* ∈ *Poly* ⟹ (λ*n*. *f* (*Suc n*)) ∈ *Poly*
⟨*proof*⟩

lemma *pCons-cases* [*cases type*: *poly*]:
obtains (*pCons*) *a q* **where** *p* = *pCons a q*

⟨*proof*⟩

lemma *pCons-induct* [*case-names 0 pCons*, *induct type*: *poly*]:
assumes *zero*: *P 0*
assumes *pCons*: ⋀*a p*. *P p* ⟹ *P* (*pCons a p*)
shows *P p*
⟨*proof*⟩

42.5 Recursion combinator for polynomials

function
poly-rec :: *'b* ⇒ (*'a*::*zero* ⇒ *'a poly* ⇒ *'b* ⇒ *'b*) ⇒ *'a poly* ⇒ *'b*
where
poly-rec-pCons-eq-if [*simp del*, *code del*]:
poly-rec z f (*pCons a p*) = *f a p* (*if p* = *0 then z else poly-rec z f p*)
⟨*proof*⟩

termination *poly-rec*
⟨*proof*⟩

lemma *poly-rec-0*:
f 0 0 z = *z* ⟹ *poly-rec z f 0* = *z*
⟨*proof*⟩

lemma *poly-rec-pCons*:
f 0 0 z = *z* ⟹ *poly-rec z f* (*pCons a p*) = *f a p* (*poly-rec z f p*)
⟨*proof*⟩

42.6 Monomials

definition
monom :: *'a* ⇒ *nat* ⇒ *'a*::*zero poly* **where**
monom a m = *Abs-poly* (λ*n*. *if m* = *n then a else 0*)

lemma *coeff-monom* [*simp*]: *coeff* (*monom a m*) *n* = (*if m*=*n then a else 0*)
⟨*proof*⟩

lemma *monom-0*: *monom a 0* = *pCons a 0*
⟨*proof*⟩

lemma *monom-Suc*: *monom a* (*Suc n*) = *pCons 0* (*monom a n*)
⟨*proof*⟩

lemma *monom-eq-0* [*simp*]: *monom 0 n* = *0*
⟨*proof*⟩

lemma *monom-eq-0-iff* [*simp*]: *monom a n* = *0* ⟷ *a* = *0*
⟨*proof*⟩

lemma *monom-eq-iff* [*simp*]: *monom a n* = *monom b n* ⟷ *a* = *b*

⟨*proof*⟩

lemma *degree-monom-le*: *degree* (*monom a n*) ≤ *n*
⟨*proof*⟩

lemma *degree-monom-eq*: *a* ≠ *0* ⟹ *degree* (*monom a n*) = *n*
⟨*proof*⟩

42.7 Addition and subtraction

instantiation *poly* :: (*comm-monoid-add*) *comm-monoid-add*
begin

definition
plus-poly-def [*code del*]:
p + *q* = *Abs-poly* (λ*n*. *coeff p n* + *coeff q n*)

lemma *Poly-add*:
fixes *f g* :: *nat* ⇒ *'a*
shows ⟦*f* ∈ *Poly*; *g* ∈ *Poly*⟧ ⟹ (λ*n*. *f n* + *g n*) ∈ *Poly*
⟨*proof*⟩

lemma *coeff-add* [*simp*]:
coeff (*p* + *q*) *n* = *coeff p n* + *coeff q n*
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *poly* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add*
⟨*proof*⟩

instantiation *poly* :: (*ab-group-add*) *ab-group-add*
begin

definition
uminus-poly-def [*code del*]:
− *p* = *Abs-poly* (λ*n*. − *coeff p n*)

definition
minus-poly-def [*code del*]:
p − *q* = *Abs-poly* (λ*n*. *coeff p n* − *coeff q n*)

lemma *Poly-minus*:
fixes *f* :: *nat* ⇒ *'a*
shows *f* ∈ *Poly* ⟹ (λ*n*. − *f n*) ∈ *Poly*
⟨*proof*⟩

lemma *Poly-diff*:
fixes $f\ g :: nat \Rightarrow 'a$
shows $[\![f \in Poly;\ g \in Poly]\!] \Longrightarrow (\lambda n.\ f\ n - g\ n) \in Poly$
⟨*proof*⟩

lemma *coeff-minus* [*simp*]: $coeff\ (-\ p)\ n = -\ coeff\ p\ n$
⟨*proof*⟩

lemma *coeff-diff* [*simp*]:
$coeff\ (p - q)\ n = coeff\ p\ n - coeff\ q\ n$
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *add-pCons* [*simp*]:
$pCons\ a\ p + pCons\ b\ q = pCons\ (a + b)\ (p + q)$
⟨*proof*⟩

lemma *minus-pCons* [*simp*]:
$-\ pCons\ a\ p = pCons\ (-\ a)\ (-\ p)$
⟨*proof*⟩

lemma *diff-pCons* [*simp*]:
$pCons\ a\ p - pCons\ b\ q = pCons\ (a - b)\ (p - q)$
⟨*proof*⟩

lemma *degree-add-le-max*: $degree\ (p + q) \leq max\ (degree\ p)\ (degree\ q)$
⟨*proof*⟩

lemma *degree-add-le*:
$[\![degree\ p \leq n;\ degree\ q \leq n]\!] \Longrightarrow degree\ (p + q) \leq n$
⟨*proof*⟩

lemma *degree-add-less*:
$[\![degree\ p < n;\ degree\ q < n]\!] \Longrightarrow degree\ (p + q) < n$
⟨*proof*⟩

lemma *degree-add-eq-right*:
$degree\ p < degree\ q \Longrightarrow degree\ (p + q) = degree\ q$
⟨*proof*⟩

lemma *degree-add-eq-left*:
$degree\ q < degree\ p \Longrightarrow degree\ (p + q) = degree\ p$
⟨*proof*⟩

lemma *degree-minus* [*simp*]: $degree\ (-\ p) = degree\ p$
⟨*proof*⟩

lemma *degree-diff-le-max*: *degree* $(p - q) \leq max\ (degree\ p)\ (degree\ q)$
⟨*proof*⟩

lemma *degree-diff-le*:
$[\![degree\ p \leq n;\ degree\ q \leq n]\!] \Longrightarrow degree\ (p - q) \leq n$
⟨*proof*⟩

lemma *degree-diff-less*:
$[\![degree\ p < n;\ degree\ q < n]\!] \Longrightarrow degree\ (p - q) < n$
⟨*proof*⟩

lemma *add-monom*: $monom\ a\ n + monom\ b\ n = monom\ (a + b)\ n$
⟨*proof*⟩

lemma *diff-monom*: $monom\ a\ n - monom\ b\ n = monom\ (a - b)\ n$
⟨*proof*⟩

lemma *minus-monom*: $- monom\ a\ n = monom\ (-a)\ n$
⟨*proof*⟩

lemma *coeff-setsum*: $coeff\ (\sum x{\in}A.\ p\ x)\ i = (\sum x{\in}A.\ coeff\ (p\ x)\ i)$
⟨*proof*⟩

lemma *monom-setsum*: $monom\ (\sum x{\in}A.\ a\ x)\ n = (\sum x{\in}A.\ monom\ (a\ x)\ n)$
⟨*proof*⟩

42.8 Multiplication by a constant

definition
$smult :: 'a{::}comm\text{-}semiring\text{-}0 \Rightarrow 'a\ poly \Rightarrow 'a\ poly$ **where**
$smult\ a\ p = Abs\text{-}poly\ (\lambda n.\ a * coeff\ p\ n)$

lemma *Poly-smult*:
fixes $f :: nat \Rightarrow 'a{::}comm\text{-}semiring\text{-}0$
shows $f \in Poly \Longrightarrow (\lambda n.\ a * f\ n) \in Poly$
⟨*proof*⟩

lemma *coeff-smult* [*simp*]: $coeff\ (smult\ a\ p)\ n = a * coeff\ p\ n$
⟨*proof*⟩

lemma *degree-smult-le*: $degree\ (smult\ a\ p) \leq degree\ p$
⟨*proof*⟩

lemma *smult-smult* [*simp*]: $smult\ a\ (smult\ b\ p) = smult\ (a * b)\ p$
⟨*proof*⟩

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

lemma *smult-0-left* [*simp*]: *smult 0 p = 0*
⟨*proof*⟩

lemma *smult-1-left* [*simp*]: *smult (1::'a::comm-semiring-1) p = p*
⟨*proof*⟩

lemma *smult-add-right*:
smult a (p + q) = smult a p + smult a q
⟨*proof*⟩

lemma *smult-add-left*:
smult (a + b) p = smult a p + smult b p
⟨*proof*⟩

lemma *smult-minus-right* [*simp*]:
smult (a::'a::comm-ring) (− p) = − smult a p
⟨*proof*⟩

lemma *smult-minus-left* [*simp*]:
smult (− a::'a::comm-ring) p = − smult a p
⟨*proof*⟩

lemma *smult-diff-right*:
smult (a::'a::comm-ring) (p − q) = smult a p − smult a q
⟨*proof*⟩

lemma *smult-diff-left*:
smult (a − b::'a::comm-ring) p = smult a p − smult b p
⟨*proof*⟩

lemmas *smult-distribs* =
smult-add-left smult-add-right
smult-diff-left smult-diff-right

lemma *smult-pCons* [*simp*]:
*smult a (pCons b p) = pCons (a * b) (smult a p)*
⟨*proof*⟩

lemma *smult-monom*: *smult a (monom b n) = monom (a * b) n*
⟨*proof*⟩

lemma *degree-smult-eq* [*simp*]:
fixes *a* :: *'a::idom*
shows *degree (smult a p) = (if a = 0 then 0 else degree p)*
⟨*proof*⟩

lemma *smult-eq-0-iff* [*simp*]:
fixes *a* :: *'a::idom*

shows *smult a p = 0 ⟷ a = 0 ∨ p = 0*
⟨*proof*⟩

42.9 Multiplication of polynomials

TODO: move to SetInterval.thy

lemma *setsum-atMost-Suc-shift*:
fixes *f* :: *nat* ⇒ *'a*::*comm-monoid-add*
shows (∑ *i*≤*Suc n*. *f i*) = *f 0* + (∑ *i*≤*n*. *f* (*Suc i*))
⟨*proof*⟩

instantiation *poly* :: (*comm-semiring-0*) *comm-semiring-0*
begin

definition
times-poly-def [*code del*]:
p ∗ *q* = *poly-rec 0* (λ*a p pq*. *smult a q* + *pCons 0 pq*) *p*

lemma *mult-poly-0-left*: (*0*::*'a poly*) ∗ *q* = *0*
⟨*proof*⟩

lemma *mult-pCons-left* [*simp*]:
pCons a p ∗ *q* = *smult a q* + *pCons 0* (*p* ∗ *q*)
⟨*proof*⟩

lemma *mult-poly-0-right*: *p* ∗ (*0*::*'a poly*) = *0*
⟨*proof*⟩

lemma *mult-pCons-right* [*simp*]:
p ∗ *pCons a q* = *smult a p* + *pCons 0* (*p* ∗ *q*)
⟨*proof*⟩

lemmas *mult-poly-0* = *mult-poly-0-left mult-poly-0-right*

lemma *mult-smult-left* [*simp*]: *smult a p* ∗ *q* = *smult a* (*p* ∗ *q*)
⟨*proof*⟩

lemma *mult-smult-right* [*simp*]: *p* ∗ *smult a q* = *smult a* (*p* ∗ *q*)
⟨*proof*⟩

lemma *mult-poly-add-left*:
fixes *p q r* :: *'a poly*
shows (*p* + *q*) ∗ *r* = *p* ∗ *r* + *q* ∗ *r*
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *poly* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel* ⟨*proof*⟩

lemma *coeff-mult*:
coeff (*p* ∗ *q*) *n* = (∑ *i*≤*n*. *coeff p i* ∗ *coeff q* (*n*−*i*))
⟨*proof*⟩

lemma *degree-mult-le*: *degree* (*p* ∗ *q*) ≤ *degree p* + *degree q*
⟨*proof*⟩

lemma *mult-monom*: *monom a m* ∗ *monom b n* = *monom* (*a* ∗ *b*) (*m* + *n*)
⟨*proof*⟩

42.10 The unit polynomial and exponentiation

instantiation *poly* :: (*comm-semiring-1*) *comm-semiring-1*
begin

definition
one-poly-def:
1 = *pCons 1 0*

instance ⟨*proof*⟩

end

instance *poly* :: (*comm-semiring-1-cancel*) *comm-semiring-1-cancel* ⟨*proof*⟩

lemma *coeff-1* [*simp*]: *coeff 1 n* = (*if n* = *0 then 1 else 0*)
⟨*proof*⟩

lemma *degree-1* [*simp*]: *degree 1* = *0*
⟨*proof*⟩

Lemmas about divisibility

lemma *dvd-smult*: *p dvd q* ⟹ *p dvd smult a q*
⟨*proof*⟩

lemma *dvd-smult-cancel*:
fixes *a* :: *'a*::*field*
shows *p dvd smult a q* ⟹ *a* ≠ *0* ⟹ *p dvd q*
⟨*proof*⟩

lemma *dvd-smult-iff*:
fixes *a* :: *'a*::*field*
shows *a* ≠ *0* ⟹ *p dvd smult a q* ⟷ *p dvd q*
⟨*proof*⟩

lemma *smult-dvd-cancel*:
smult a p dvd q ⟹ *p dvd q*
⟨*proof*⟩

lemma *smult-dvd*:
fixes $a :: 'a$::*field*
shows p *dvd* $q \Longrightarrow a \neq 0 \Longrightarrow$ *smult* a p *dvd* q
⟨*proof*⟩

lemma *smult-dvd-iff*:
fixes $a :: 'a$::*field*
shows *smult* a p *dvd* $q \longleftrightarrow$ (*if* $a = 0$ *then* $q = 0$ *else* p *dvd* q)
⟨*proof*⟩

lemma *degree-power-le*: *degree* $(p \hat{} n) \leq$ *degree* $p * n$
⟨*proof*⟩

instance *poly* :: (*comm-ring*) *comm-ring* ⟨*proof*⟩

instance *poly* :: (*comm-ring-1*) *comm-ring-1* ⟨*proof*⟩

instantiation *poly* :: (*comm-ring-1*) *number-ring*
begin

definition
number-of k = (*of-int* $k :: 'a$ *poly*)

instance
⟨*proof*⟩

end

42.11 Polynomials form an integral domain

lemma *coeff-mult-degree-sum*:
coeff $(p * q)$ (*degree* p + *degree* q) =
coeff p (*degree* p) $*$ *coeff* q (*degree* q)
⟨*proof*⟩

instance *poly* :: (*idom*) *idom*
⟨*proof*⟩

lemma *degree-mult-eq*:
fixes p $q :: 'a$::*idom poly*
shows $[\![p \neq 0; q \neq 0]\!] \Longrightarrow$ *degree* $(p * q)$ = *degree* p + *degree* q
⟨*proof*⟩

lemma *dvd-imp-degree-le*:
fixes p $q :: 'a$::*idom poly*
shows $[\![p$ *dvd* $q; q \neq 0]\!] \Longrightarrow$ *degree* $p \leq$ *degree* q
⟨*proof*⟩

42.12 Polynomials form an ordered integral domain

definition
pos-poly :: *'a*::*linordered-idom poly* ⇒ *bool*
where
pos-poly p ⟷ *0* < *coeff p* (*degree p*)

lemma *pos-poly-pCons*:
pos-poly (*pCons a p*) ⟷ *pos-poly p* ∨ (*p* = *0* ∧ *0* < *a*)
⟨*proof*⟩

lemma *not-pos-poly-0* [*simp*]: ¬ *pos-poly 0*
⟨*proof*⟩

lemma *pos-poly-add*: ⟦*pos-poly p*; *pos-poly q*⟧ ⟹ *pos-poly* (*p* + *q*)
⟨*proof*⟩

lemma *pos-poly-mult*: ⟦*pos-poly p*; *pos-poly q*⟧ ⟹ *pos-poly* (*p* ∗ *q*)
⟨*proof*⟩

lemma *pos-poly-total*: *p* = *0* ∨ *pos-poly p* ∨ *pos-poly* (− *p*)
⟨*proof*⟩

instantiation *poly* :: (*linordered-idom*) *linordered-idom*
begin

definition
[*code del*]:
x < *y* ⟷ *pos-poly* (*y* − *x*)

definition
[*code del*]:
x ≤ *y* ⟷ *x* = *y* ∨ *pos-poly* (*y* − *x*)

definition
[*code del*]:
abs (*x*::*'a poly*) = (*if x* < *0 then* − *x else x*)

definition
[*code del*]:
sgn (*x*::*'a poly*) = (*if x* = *0 then 0 else if 0* < *x then 1 else* − *1*)

instance ⟨*proof*⟩

end

TODO: Simplification rules for comparisons

42.13 Long division of polynomials

definition
pdivmod-rel :: *'a*::*field poly* ⇒ *'a poly* ⇒ *'a poly* ⇒ *'a poly* ⇒ *bool*
where
[*code del*]:
pdivmod-rel x y q r ⟷
x = *q* ∗ *y* + *r* ∧ (*if y* = *0 then q* = *0 else r* = *0* ∨ *degree r* < *degree y*)

lemma *pdivmod-rel-0*:
pdivmod-rel 0 y 0 0
⟨*proof*⟩

lemma *pdivmod-rel-by-0*:
pdivmod-rel x 0 0 x
⟨*proof*⟩

lemma *eq-zero-or-degree-less*:
assumes *degree p* ≤ *n* **and** *coeff p n* = *0*
shows *p* = *0* ∨ *degree p* < *n*
⟨*proof*⟩

lemma *pdivmod-rel-pCons*:
assumes *rel*: *pdivmod-rel x y q r*
assumes *y*: *y* ≠ *0*
assumes *b*: *b* = *coeff* (*pCons a r*) (*degree y*) / *coeff y* (*degree y*)
shows *pdivmod-rel* (*pCons a x*) *y* (*pCons b q*) (*pCons a r* − *smult b y*)
(**is** *pdivmod-rel ?x y ?q ?r*)
⟨*proof*⟩

lemma *pdivmod-rel-exists*: ∃ *q r*. *pdivmod-rel x y q r*
⟨*proof*⟩

lemma *pdivmod-rel-unique*:
assumes *1*: *pdivmod-rel x y q1 r1*
assumes *2*: *pdivmod-rel x y q2 r2*
shows *q1* = *q2* ∧ *r1* = *r2*
⟨*proof*⟩

lemma *pdivmod-rel-0-iff*: *pdivmod-rel 0 y q r* ⟷ *q* = *0* ∧ *r* = *0*
⟨*proof*⟩

lemma *pdivmod-rel-by-0-iff*: *pdivmod-rel x 0 q r* ⟷ *q* = *0* ∧ *r* = *x*
⟨*proof*⟩

lemmas *pdivmod-rel-unique-div* =
pdivmod-rel-unique [*THEN conjunct1*, *standard*]

lemmas *pdivmod-rel-unique-mod* =
pdivmod-rel-unique [*THEN conjunct2*, *standard*]

instantiation *poly* :: (*field*) *ring-div*
begin

definition *div-poly* **where**
[*code del*]: *x div y* = (*THE q*. ∃ *r*. *pdivmod-rel x y q r*)

definition *mod-poly* **where**
[*code del*]: *x mod y* = (*THE r*. ∃ *q*. *pdivmod-rel x y q r*)

lemma *div-poly-eq*:
pdivmod-rel x y q r ⟹ *x div y* = *q*
⟨*proof*⟩

lemma *mod-poly-eq*:
pdivmod-rel x y q r ⟹ *x mod y* = *r*
⟨*proof*⟩

lemma *pdivmod-rel*:
pdivmod-rel x y (*x div y*) (*x mod y*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *degree-mod-less*:
y ≠ *0* ⟹ *x mod y* = *0* ∨ *degree* (*x mod y*) < *degree y*
⟨*proof*⟩

lemma *div-poly-less*: *degree x* < *degree y* ⟹ *x div y* = *0*
⟨*proof*⟩

lemma *mod-poly-less*: *degree x* < *degree y* ⟹ *x mod y* = *x*
⟨*proof*⟩

lemma *pdivmod-rel-smult-left*:
pdivmod-rel x y q r
⟹ *pdivmod-rel* (*smult a x*) *y* (*smult a q*) (*smult a r*)
⟨*proof*⟩

lemma *div-smult-left*: (*smult a x*) *div y* = *smult a* (*x div y*)
⟨*proof*⟩

lemma *mod-smult-left*: (*smult a x*) *mod y* = *smult a* (*x mod y*)
⟨*proof*⟩

lemma *poly-div-minus-left* [*simp*]:
fixes *x y* :: *'a*::*field poly*

shows $(-\ x)\ div\ y = -\ (x\ div\ y)$
⟨*proof*⟩

lemma *poly-mod-minus-left* [*simp*]:
fixes $x\ y :: 'a$::*field poly*
shows $(-\ x)\ mod\ y = -\ (x\ mod\ y)$
⟨*proof*⟩

lemma *pdivmod-rel-smult-right*:
$[\![a \neq 0;\ pdivmod\text{-}rel\ x\ y\ q\ r]\!]$
$\Longrightarrow pdivmod\text{-}rel\ x\ (smult\ a\ y)\ (smult\ (inverse\ a)\ q)\ r$
⟨*proof*⟩

lemma *div-smult-right*:
$a \neq 0 \Longrightarrow x\ div\ (smult\ a\ y) = smult\ (inverse\ a)\ (x\ div\ y)$
⟨*proof*⟩

lemma *mod-smult-right*: $a \neq 0 \Longrightarrow x\ mod\ (smult\ a\ y) = x\ mod\ y$
⟨*proof*⟩

lemma *poly-div-minus-right* [*simp*]:
fixes $x\ y :: 'a$::*field poly*
shows $x\ div\ (-\ y) = -\ (x\ div\ y)$
⟨*proof*⟩

lemma *poly-mod-minus-right* [*simp*]:
fixes $x\ y :: 'a$::*field poly*
shows $x\ mod\ (-\ y) = x\ mod\ y$
⟨*proof*⟩

lemma *pdivmod-rel-mult*:
$[\![pdivmod\text{-}rel\ x\ y\ q\ r;\ pdivmod\text{-}rel\ q\ z\ q'\ r']\!]$
$\Longrightarrow pdivmod\text{-}rel\ x\ (y * z)\ q'\ (y * r' + r)$
⟨*proof*⟩

lemma *poly-div-mult-right*:
fixes $x\ y\ z :: 'a$::*field poly*
shows $x\ div\ (y * z) = (x\ div\ y)\ div\ z$
⟨*proof*⟩

lemma *poly-mod-mult-right*:
fixes $x\ y\ z :: 'a$::*field poly*
shows $x\ mod\ (y * z) = y * (x\ div\ y\ mod\ z) + x\ mod\ y$
⟨*proof*⟩

lemma *mod-pCons*:
fixes a **and** x
assumes y: $y \neq 0$
defines b: $b \equiv coeff\ (pCons\ a\ (x\ mod\ y))\ (degree\ y)\ /\ coeff\ y\ (degree\ y)$

shows $(pCons\ a\ x)\ mod\ y = (pCons\ a\ (x\ mod\ y) - smult\ b\ y)$
⟨*proof*⟩

42.14 GCD of polynomials

function
$poly\text{-}gcd :: 'a::field\ poly \Rightarrow 'a\ poly \Rightarrow 'a\ poly$ **where**
$poly\text{-}gcd\ x\ 0 = smult\ (inverse\ (coeff\ x\ (degree\ x)))\ x$
$| \ y \neq 0 \Longrightarrow poly\text{-}gcd\ x\ y = poly\text{-}gcd\ y\ (x\ mod\ y)$
⟨*proof*⟩

termination *poly-gcd*
⟨*proof*⟩

declare *poly-gcd.simps* [*simp del*, *code del*]

lemma *poly-gcd-dvd1* [*iff*]: $poly\text{-}gcd\ x\ y\ dvd\ x$
and *poly-gcd-dvd2* [*iff*]: $poly\text{-}gcd\ x\ y\ dvd\ y$
⟨*proof*⟩

lemma *poly-gcd-greatest*: $k\ dvd\ x \Longrightarrow k\ dvd\ y \Longrightarrow k\ dvd\ poly\text{-}gcd\ x\ y$
⟨*proof*⟩

lemma *dvd-poly-gcd-iff* [*iff*]:
$k\ dvd\ poly\text{-}gcd\ x\ y \longleftrightarrow k\ dvd\ x \wedge k\ dvd\ y$
⟨*proof*⟩

lemma *poly-gcd-monic*:
$coeff\ (poly\text{-}gcd\ x\ y)\ (degree\ (poly\text{-}gcd\ x\ y)) =$
$(if\ x = 0 \wedge y = 0\ then\ 0\ else\ 1)$
⟨*proof*⟩

lemma *poly-gcd-zero-iff* [*simp*]:
$poly\text{-}gcd\ x\ y = 0 \longleftrightarrow x = 0 \wedge y = 0$
⟨*proof*⟩

lemma *poly-gcd-0-0* [*simp*]: $poly\text{-}gcd\ 0\ 0 = 0$
⟨*proof*⟩

lemma *poly-dvd-antisym*:
fixes $p\ q :: 'a::idom\ poly$
assumes *coeff*: $coeff\ p\ (degree\ p) = coeff\ q\ (degree\ q)$
assumes *dvd1*: $p\ dvd\ q$ **and** *dvd2*: $q\ dvd\ p$ **shows** $p = q$
⟨*proof*⟩

lemma *poly-gcd-unique*:
assumes *dvd1*: $d\ dvd\ x$ **and** *dvd2*: $d\ dvd\ y$
and *greatest*: $\bigwedge k.\ k\ dvd\ x \Longrightarrow k\ dvd\ y \Longrightarrow k\ dvd\ d$
and *monic*: $coeff\ d\ (degree\ d) = (if\ x = 0 \wedge y = 0\ then\ 0\ else\ 1)$

shows *poly-gcd x y = d*
⟨*proof*⟩

interpretation *poly-gcd*!: *abel-semigroup poly-gcd*
⟨*proof*⟩

lemmas *poly-gcd-assoc = poly-gcd.assoc*
lemmas *poly-gcd-commute = poly-gcd.commute*
lemmas *poly-gcd-left-commute = poly-gcd.left-commute*

lemmas *poly-gcd-ac = poly-gcd-assoc poly-gcd-commute poly-gcd-left-commute*

lemma *poly-gcd-1-left* [*simp*]: *poly-gcd 1 y = 1*
⟨*proof*⟩

lemma *poly-gcd-1-right* [*simp*]: *poly-gcd x 1 = 1*
⟨*proof*⟩

lemma *poly-gcd-minus-left* [*simp*]: *poly-gcd (− x) y = poly-gcd x y*
⟨*proof*⟩

lemma *poly-gcd-minus-right* [*simp*]: *poly-gcd x (− y) = poly-gcd x y*
⟨*proof*⟩

42.15 Evaluation of polynomials

definition
poly :: *'a*::*comm-semiring-0 poly* ⇒ *'a* ⇒ *'a* **where**
poly = poly-rec (λ*x. 0*) (λ*a p f x. a + x * f x*)

lemma *poly-0* [*simp*]: *poly 0 x = 0*
⟨*proof*⟩

lemma *poly-pCons* [*simp*]: *poly* (*pCons a p*) *x = a + x * poly p x*
⟨*proof*⟩

lemma *poly-1* [*simp*]: *poly 1 x = 1*
⟨*proof*⟩

lemma *poly-monom*:
fixes *a x* :: *'a*::{*comm-semiring-1*}
shows *poly* (*monom a n*) *x = a * x ^ n*
⟨*proof*⟩

lemma *poly-add* [*simp*]: *poly* (*p + q*) *x = poly p x + poly q x*
⟨*proof*⟩

lemma *poly-minus* [*simp*]:
fixes *x* :: *'a*::*comm-ring*

shows *poly* $(-\ p)\ x = -\ poly\ p\ x$
⟨*proof*⟩

lemma *poly-diff* [*simp*]:
fixes x :: *'a*::*comm-ring*
shows *poly* $(p - q)\ x = poly\ p\ x - poly\ q\ x$
⟨*proof*⟩

lemma *poly-setsum*: *poly* $(\sum k{\in}A.\ p\ k)\ x = (\sum k{\in}A.\ poly\ (p\ k)\ x)$
⟨*proof*⟩

lemma *poly-smult* [*simp*]: *poly* $(smult\ a\ p)\ x = a * poly\ p\ x$
⟨*proof*⟩

lemma *poly-mult* [*simp*]: *poly* $(p * q)\ x = poly\ p\ x * poly\ q\ x$
⟨*proof*⟩

lemma *poly-power* [*simp*]:
fixes p :: *'a*::{*comm-semiring-1*} *poly*
shows *poly* $(p\ \hat{}\ n)\ x = poly\ p\ x\ \hat{}\ n$
⟨*proof*⟩

42.16 Synthetic division

Synthetic division is simply division by the linear polynomial $x - c$.

definition
synthetic-divmod :: *'a*::*comm-semiring-0 poly* ⇒ *'a* ⇒ *'a poly* × *'a*
where [*code del*]:
synthetic-divmod p c =
poly-rec $(0,\ 0)$ $(\lambda a\ p\ (q,\ r).\ (pCons\ r\ q,\ a + c * r))\ p$

definition
synthetic-div :: *'a*::*comm-semiring-0 poly* ⇒ *'a* ⇒ *'a poly*
where
synthetic-div p c = *fst* (*synthetic-divmod p c*)

lemma *synthetic-divmod-0* [*simp*]:
synthetic-divmod 0 c = $(0,\ 0)$
⟨*proof*⟩

lemma *synthetic-divmod-pCons* [*simp*]:
synthetic-divmod $(pCons\ a\ p)\ c$ =
$(\lambda(q,\ r).\ (pCons\ r\ q,\ a + c * r))$ (*synthetic-divmod p c*)
⟨*proof*⟩

lemma *snd-synthetic-divmod*: *snd* (*synthetic-divmod p c*) = *poly p c*
⟨*proof*⟩

lemma *synthetic-div-0* [*simp*]: *synthetic-div 0 c* = *0*

⟨*proof*⟩

lemma *synthetic-div-pCons* [*simp*]:
synthetic-div (*pCons a p*) *c* = *pCons* (*poly p c*) (*synthetic-div p c*)
⟨*proof*⟩

lemma *synthetic-div-eq-0-iff*:
synthetic-div p c = *0* ⟷ *degree p* = *0*
⟨*proof*⟩

lemma *degree-synthetic-div*:
degree (*synthetic-div p c*) = *degree p* − *1*
⟨*proof*⟩

lemma *synthetic-div-correct*:
p + *smult c* (*synthetic-div p c*) = *pCons* (*poly p c*) (*synthetic-div p c*)
⟨*proof*⟩

lemma *synthetic-div-unique-lemma*: *smult c p* = *pCons a p* ⟹ *p* = *0*
⟨*proof*⟩

lemma *synthetic-div-unique*:
p + *smult c q* = *pCons r q* ⟹ *r* = *poly p c* ∧ *q* = *synthetic-div p c*
⟨*proof*⟩

lemma *synthetic-div-correct'*:
fixes *c* :: *'a*::*comm-ring-1*
shows [:−*c*, *1*:] ∗ *synthetic-div p c* + [:*poly p c*:] = *p*
⟨*proof*⟩

lemma *poly-eq-0-iff-dvd*:
fixes *c* :: *'a*::*idom*
shows *poly p c* = *0* ⟷ [:−*c*, *1*:] *dvd p*
⟨*proof*⟩

lemma *dvd-iff-poly-eq-0*:
fixes *c* :: *'a*::*idom*
shows [:*c*, *1*:] *dvd p* ⟷ *poly p* (−*c*) = *0*
⟨*proof*⟩

lemma *poly-roots-finite*:
fixes *p* :: *'a*::*idom poly*
shows *p* ≠ *0* ⟹ *finite* {*x*. *poly p x* = *0*}
⟨*proof*⟩

lemma *poly-zero*:
fixes *p* :: *'a*::{*idom*,*ring-char-0*} *poly*
shows *poly p* = *poly 0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *poly-eq-iff*:
fixes *p q* :: *'a*::{*idom*,*ring-char-0*} *poly*
shows *poly p* = *poly q* ⟷ *p* = *q*
⟨*proof*⟩

42.17 Composition of polynomials

definition
pcompose :: *'a*::*comm-semiring-0 poly* ⇒ *'a poly* ⇒ *'a poly*
where
pcompose p q = *poly-rec 0* (λ*a* - *c*. [:*a*:] + *q* ∗ *c*) *p*

lemma *pcompose-0* [*simp*]: *pcompose 0 q* = *0*
⟨*proof*⟩

lemma *pcompose-pCons*:
pcompose (*pCons a p*) *q* = [:*a*:] + *q* ∗ *pcompose p q*
⟨*proof*⟩

lemma *poly-pcompose*: *poly* (*pcompose p q*) *x* = *poly p* (*poly q x*)
⟨*proof*⟩

lemma *degree-pcompose-le*:
degree (*pcompose p q*) ≤ *degree p* ∗ *degree q*
⟨*proof*⟩

42.18 Order of polynomial roots

definition
order :: *'a*::*idom* ⇒ *'a poly* ⇒ *nat*
where
[*code del*]:
order a p = (*LEAST n*. ¬ [:−*a*, *1*:] ^ *Suc n dvd p*)

lemma *coeff-linear-power*:
fixes *a* :: *'a*::*comm-semiring-1*
shows *coeff* ([:*a*, *1*:] ^ *n*) *n* = *1*
⟨*proof*⟩

lemma *degree-linear-power*:
fixes *a* :: *'a*::*comm-semiring-1*
shows *degree* ([:*a*, *1*:] ^ *n*) = *n*
⟨*proof*⟩

lemma *order-1*: [:−*a*, *1*:] ^ *order a p dvd p*
⟨*proof*⟩

lemma *order-2*: *p* ≠ *0* ⟹ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
⟨*proof*⟩

lemma *order*:
$p \neq 0 \Longrightarrow [:-a,\ 1:]\ \hat{}\ order\ a\ p\ dvd\ p \wedge \neg\ [:-a,\ 1:]\ \hat{}\ Suc\ (order\ a\ p)\ dvd\ p$
⟨*proof*⟩

lemma *order-degree*:
assumes *p*: $p \neq 0$
shows $order\ a\ p \leq degree\ p$
⟨*proof*⟩

lemma *order-root*: $poly\ p\ a = 0 \longleftrightarrow p = 0 \vee order\ a\ p \neq 0$
⟨*proof*⟩

42.19 Configuration of the code generator

code-datatype $0::'a::zero\ poly\ pCons$

declare *pCons-0-0* [*code-post*]

instantiation *poly* :: ({*zero*,*eq*}) *eq*
begin

definition [*code del*]:
$eq\text{-}class.eq\ (p::'a\ poly)\ q \longleftrightarrow p = q$

instance
⟨*proof*⟩

end

lemma *eq-poly-code* [*code*]:
$eq\text{-}class.eq\ (0::\text{-}\ poly)\ (0::\text{-}\ poly) \longleftrightarrow True$
$eq\text{-}class.eq\ (0::\text{-}\ poly)\ (pCons\ b\ q) \longleftrightarrow eq\text{-}class.eq\ 0\ b \wedge eq\text{-}class.eq\ 0\ q$
$eq\text{-}class.eq\ (pCons\ a\ p)\ (0::\text{-}\ poly) \longleftrightarrow eq\text{-}class.eq\ a\ 0 \wedge eq\text{-}class.eq\ p\ 0$
$eq\text{-}class.eq\ (pCons\ a\ p)\ (pCons\ b\ q) \longleftrightarrow eq\text{-}class.eq\ a\ b \wedge eq\text{-}class.eq\ p\ q$
⟨*proof*⟩

lemmas *coeff-code* [*code*] =
coeff-0 coeff-pCons-0 coeff-pCons-Suc

lemmas *degree-code* [*code*] =
degree-0 degree-pCons-eq-if

lemmas *monom-poly-code* [*code*] =
monom-0 monom-Suc

lemma *add-poly-code* [*code*]:
$0 + q = (q\ ::\ \text{-}\ poly)$
$p + 0 = (p\ ::\ \text{-}\ poly)$

pCons a p + pCons b q = pCons (a + b) (p + q)
⟨*proof*⟩

lemma *minus-poly-code* [*code*]:
− 0 = (0 :: - poly)
− pCons a p = pCons (− a) (− p)
⟨*proof*⟩

lemma *diff-poly-code* [*code*]:
0 − q = (− q :: - poly)
p − 0 = (p :: - poly)
pCons a p − pCons b q = pCons (a − b) (p − q)
⟨*proof*⟩

lemmas *smult-poly-code* [*code*] =
smult-0-right smult-pCons

lemma *mult-poly-code* [*code*]:
0 ∗ q = (0 :: - poly)
pCons a p ∗ q = smult a q + pCons 0 (p ∗ q)
⟨*proof*⟩

lemmas *poly-code* [*code*] =
poly-0 poly-pCons

lemmas *synthetic-divmod-code* [*code*] =
synthetic-divmod-0 synthetic-divmod-pCons

code generator setup for div and mod

definition
pdivmod :: *'a*::*field poly ⇒ 'a poly ⇒ 'a poly × 'a poly*
where
[*code del*]: *pdivmod x y = (x div y, x mod y)*

lemma *div-poly-code* [*code*]: *x div y = fst (pdivmod x y)*
⟨*proof*⟩

lemma *mod-poly-code* [*code*]: *x mod y = snd (pdivmod x y)*
⟨*proof*⟩

lemma *pdivmod-0* [*code*]: *pdivmod 0 y = (0, 0)*
⟨*proof*⟩

lemma *pdivmod-pCons* [*code*]:
pdivmod (pCons a x) y =
(if y = 0 then (0, pCons a x) else
(let (q, r) = pdivmod x y;
b = coeff (pCons a r) (degree y) / coeff y (degree y)
in (pCons b q, pCons a r − smult b y)))

⟨*proof*⟩

lemma *poly-gcd-code* [*code*]:
poly-gcd x y =
(*if y* = *0 then smult* (*inverse* (*coeff x* (*degree x*))) *x*
else poly-gcd y (*x mod y*))
⟨*proof*⟩

end

43 Fundamental-Theorem-Algebra: Fundamental Theorem of Algebra

theory *Fundamental-Theorem-Algebra*
imports *Polynomial Complex*
begin

43.1 Square root of complex numbers

definition *csqrt* :: *complex* ⇒ *complex* **where**
csqrt z = (*if Im z* = *0 then*
if 0 ≤ *Re z then Complex* (*sqrt*(*Re z*)) *0*
else Complex 0 (*sqrt*(− *Re z*))
else Complex (*sqrt*((*cmod z* + *Re z*) */2*))
((*Im z* / *abs*(*Im z*)) ∗ *sqrt*((*cmod z* − *Re z*) */2*)))

lemma *csqrt*[*algebra*]: *csqrt z* ^ *2* = *z*
⟨*proof*⟩

43.2 More lemmas about module of complex numbers

lemma *complex-of-real-power*: *complex-of-real x* ^ *n* = *complex-of-real* (*x*^*n*)
⟨*proof*⟩

lemma *real-down2*: (*0*::*real*) < *d1* ⟹ *0* < *d2* ==> *EX e*. *0* < *e* & *e* < *d1* & *e* < *d2*
⟨*proof*⟩

The triangle inequality for cmod

lemma *complex-mod-triangle-sub*: *cmod w* ≤ *cmod* (*w* + *z*) + *norm z*
⟨*proof*⟩

43.3 Basic lemmas about complex polynomials

lemma *poly-bound-exists*:
shows ∃ *m*. *m* > *0* ∧ (∀ *z*. *cmod z* <= *r* ⟶ *cmod* (*poly p z*) ≤ *m*)
⟨*proof*⟩

Offsetting the variable in a polynomial gives another of same degree

definition
offset-poly p h = poly-rec 0 (λa p q. smult h q + pCons a q) p

lemma *offset-poly-0*: *offset-poly 0 h = 0*
⟨*proof*⟩

lemma *offset-poly-pCons*:
offset-poly (pCons a p) h =
smult h (offset-poly p h) + pCons a (offset-poly p h)
⟨*proof*⟩

lemma *offset-poly-single*: *offset-poly [:a:] h = [:a:]*
⟨*proof*⟩

lemma *poly-offset-poly*: *poly (offset-poly p h) x = poly p (h + x)*
⟨*proof*⟩

lemma *offset-poly-eq-0-lemma*: *smult c p + pCons a p = 0 ⟹ p = 0*
⟨*proof*⟩

lemma *offset-poly-eq-0-iff*: *offset-poly p h = 0 ⟷ p = 0*
⟨*proof*⟩

lemma *degree-offset-poly*: *degree (offset-poly p h) = degree p*
⟨*proof*⟩

definition
psize p = (if p = 0 then 0 else Suc (degree p))

lemma *psize-eq-0-iff* [*simp*]: *psize p = 0 ⟷ p = 0*
⟨*proof*⟩

lemma *poly-offset*: ∃ *q. psize q = psize p ∧ (∀ x. poly q (x::complex) = poly p (a + x))*
⟨*proof*⟩

An alternative useful formulation of completeness of the reals

lemma *real-sup-exists*: **assumes** *ex*: ∃ *x. P x* **and** *bz*: ∃ *z*. ∀ *x. P x ⟶ x < z*
shows ∃ (*s*::*real*). ∀ *y*. (∃ *x. P x ∧ y < x*) ⟷ *y < s*
⟨*proof*⟩

43.4 Fundamental theorem of algebra

lemma *unimodular-reduce-norm*:
assumes *md*: *cmod z = 1*
shows *cmod (z + 1) < 1 ∨ cmod (z − 1) < 1 ∨ cmod (z + ii) < 1 ∨ cmod (z − ii) < 1*
⟨*proof*⟩

Hence we can always reduce modulus of $1 + b\ z\hat{}n$ if nonzero

lemma *reduce-poly-simple*:
assumes *b*: $b \neq 0$ **and** *n*: $n \neq 0$
shows $\exists z.\ cmod\ (1 + b * z\hat{}n) < 1$
⟨*proof*⟩

Bolzano-Weierstrass type property for closed disc in complex plane.

lemma *metric-bound-lemma*: $cmod\ (x - y) <= |Re\ x - Re\ y| + |Im\ x - Im\ y|$
⟨*proof*⟩

lemma *bolzano-weierstrass-complex-disc*:
assumes *r*: $\forall n.\ cmod\ (s\ n) \leq r$
shows $\exists f\ z.\ subseq\ f \wedge (\forall e > 0.\ \exists N.\ \forall n \geq N.\ cmod\ (s\ (f\ n) - z) < e)$
⟨*proof*⟩

Polynomial is continuous.

lemma *poly-cont*:
assumes *ep*: $e > 0$
shows $\exists d > 0.\ \forall w.\ 0 < cmod\ (w - z) \wedge cmod\ (w - z) < d \longrightarrow cmod\ (poly\ p\ w - poly\ p\ z) < e$
⟨*proof*⟩

Hence a polynomial attains minimum on a closed disc in the complex plane.

lemma *poly-minimum-modulus-disc*:
$\exists z.\ \forall w.\ cmod\ w \leq r \longrightarrow cmod\ (poly\ p\ z) \leq cmod\ (poly\ p\ w)$
⟨*proof*⟩

lemma $(rcis\ (sqrt\ (abs\ r))\ (a/2))\ \hat{}\ 2 = rcis\ (abs\ r)\ a$
⟨*proof*⟩

lemma *cispi*: $cis\ pi = -1$
⟨*proof*⟩

lemma $(rcis\ (sqrt\ (abs\ r))\ ((pi + a)/2))\ \hat{}\ 2 = rcis\ (-\ abs\ r)\ a$
⟨*proof*⟩

Nonzero polynomial in z goes to infinity as z does.

lemma *poly-infinity*:
assumes *ex*: $p \neq 0$
shows $\exists r.\ \forall z.\ r \leq cmod\ z \longrightarrow d \leq cmod\ (poly\ (pCons\ a\ p)\ z)$
⟨*proof*⟩

Hence polynomial's modulus attains its minimum somewhere.

lemma *poly-minimum-modulus*:
$\exists z.\forall w.\ cmod\ (poly\ p\ z) \leq cmod\ (poly\ p\ w)$
⟨*proof*⟩

Constant function (non-syntactic characterization).

definition *constant f* = (∀ *x y*. *f x* = *f y*)

lemma *nonconstant-length*: ¬ (*constant* (*poly p*)) ⟹ *psize p* ≥ *2*
⟨*proof*⟩

lemma *poly-replicate-append*:
poly (*monom 1 n* ∗ *p*) (*x*::′*a*::{*comm-ring-1*}) = *x* ^ *n* ∗ *poly p x*
⟨*proof*⟩

Decomposition of polynomial, skipping zero coefficients after the first.

lemma *poly-decompose-lemma*:
assumes *nz*: ¬(∀ *z*. *z*≠*0* ⟶ *poly p z* = (*0*::′*a*::{*idom*}))
shows ∃ *k a q*. *a*≠*0* ∧ *Suc* (*psize q* + *k*) = *psize p* ∧
(∀ *z*. *poly p z* = *z* ^ *k* ∗ *poly* (*pCons a q*) *z*)
⟨*proof*⟩

lemma *poly-decompose*:
assumes *nc*: ~*constant*(*poly p*)
shows ∃ *k a q*. *a*≠(*0*::′*a*::{*idom*}) ∧ *k*≠*0* ∧
psize q + *k* + *1* = *psize p* ∧
(∀ *z*. *poly p z* = *poly p 0* + *z* ^ *k* ∗ *poly* (*pCons a q*) *z*)
⟨*proof*⟩

Fundamental theorem of algebra

lemma *fundamental-theorem-of-algebra*:
assumes *nc*: ~*constant*(*poly p*)
shows ∃ *z*::*complex*. *poly p z* = *0*
⟨*proof*⟩

Alternative version with a syntactic notion of constant polynomial.

lemma *fundamental-theorem-of-algebra-alt*:
assumes *nc*: ~(∃ *a l*. *a*≠ *0* ∧ *l* = *0* ∧ *p* = *pCons a l*)
shows ∃ *z*. *poly p z* = (*0*::*complex*)
⟨*proof*⟩

43.5 Nullstellensatz, degrees and divisibility of polynomials

lemma *nullstellensatz-lemma*:
fixes *p* :: *complex poly*
assumes ∀ *x*. *poly p x* = *0* ⟶ *poly q x* = *0*
and *degree p* = *n* **and** *n* ≠ *0*
shows *p dvd* (*q* ^ *n*)
⟨*proof*⟩

lemma *nullstellensatz-univariate*:
(∀ *x*. *poly p x* = (*0*::*complex*) ⟶ *poly q x* = *0*) ⟷
p dvd (*q* ^ (*degree p*)) ∨ (*p* = *0* ∧ *q* = *0*)
⟨*proof*⟩

Useful lemma

lemma *constant-degree*:
fixes *p* :: *'a*::{*idom*,*ring-char-0*} *poly*
shows *constant* (*poly p*) ⟷ *degree p* = *0* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *divides-degree*: **assumes** *pq*: *p dvd* (*q*:: *complex poly*)
shows *degree p* ≤ *degree q* ∨ *q* = *0*
⟨*proof*⟩

lemma *mpoly-base-conv*:
(*0*::*complex*) ≡ *poly 0 x c* ≡ *poly* [:*c*:] *x x* ≡ *poly* [:*0*,*1*:] *x* ⟨*proof*⟩

lemma *mpoly-norm-conv*:
poly [:*0*:] (*x*::*complex*) ≡ *poly 0 x poly* [:*poly 0 y*:] *x* ≡ *poly 0 x* ⟨*proof*⟩

lemma *mpoly-sub-conv*:
poly p (*x*::*complex*) − *poly q x* ≡ *poly p x* + −*1* ∗ *poly q x*
⟨*proof*⟩

lemma *poly-pad-rule*: *poly p x* = *0* ==> *poly* (*pCons 0 p*) *x* = (*0*::*complex*)
⟨*proof*⟩

lemma *poly-cancel-eq-conv*: *p* = (*0*::*complex*) ⟹ *a* ≠ *0* ⟹ (*q* = *0*) ≡ (*a* ∗ *q* − *b* ∗ *p* = *0*) ⟨*proof*⟩

lemma *resolve-eq-raw*: *poly 0 x* ≡ *0 poly* [:*c*:] *x* ≡ (*c*::*complex*) ⟨*proof*⟩
lemma *resolve-eq-then*: (*P* ⟹ (*Q* ≡ *Q1*)) ⟹ (¬*P* ⟹ (*Q* ≡ *Q2*))
⟹ *Q* ≡ *P* ∧ *Q1* ∨ ¬*P*∧ *Q2* ⟨*proof*⟩

lemma *poly-divides-pad-rule*:
fixes *p q* :: *complex poly*
assumes *pq*: *p dvd q*
shows *p dvd* (*pCons* (*0*::*complex*) *q*)
⟨*proof*⟩

lemma *poly-divides-pad-const-rule*:
fixes *p q* :: *complex poly*
assumes *pq*: *p dvd q*
shows *p dvd* (*smult a q*)
⟨*proof*⟩

lemma *poly-divides-conv0*:
fixes *p* :: *complex poly*
assumes *lgpq*: *degree q* < *degree p* **and** *lq*:*p* ≠ *0*
shows *p dvd q* ≡ *q* = *0* (**is** *?lhs* ≡ *?rhs*)
⟨*proof*⟩

lemma *poly-divides-conv1*:
assumes *a0*: $a \neq (0::complex)$ **and** *pp'*: $(p::complex\ poly)\ dvd\ p'$
and *qrp'*: $smult\ a\ q - p' \equiv r$
shows $p\ dvd\ q \equiv p\ dvd\ (r::complex\ poly)$ (**is** *?lhs* $\equiv$ *?rhs*)
⟨*proof*⟩

lemma *basic-cqe-conv1*:
$(\exists x.\ poly\ p\ x = 0 \wedge poly\ 0\ x \neq 0) \equiv False$
$(\exists x.\ poly\ 0\ x \neq 0) \equiv False$
$(\exists x.\ poly\ [:c:]\ x \neq 0) \equiv c \neq 0$
$(\exists x.\ poly\ 0\ x = 0) \equiv True$
$(\exists x.\ poly\ [:c:]\ x = 0) \equiv c = 0$ ⟨*proof*⟩

lemma *basic-cqe-conv2*:
assumes *l*:$p \neq 0$
shows $(\exists x.\ poly\ (pCons\ a\ (pCons\ b\ p))\ x = (0::complex)) \equiv True$
⟨*proof*⟩

lemma *basic-cqe-conv-2b*: $(\exists x.\ poly\ p\ x \neq (0::complex)) \equiv (p \neq 0)$
⟨*proof*⟩

lemma *basic-cqe-conv3*:
fixes $p\ q :: complex\ poly$
assumes *l*: $p \neq 0$
shows $(\exists x.\ poly\ (pCons\ a\ p)\ x = 0 \wedge poly\ q\ x \neq 0) \equiv \neg\ ((pCons\ a\ p)\ dvd\ (q\ \hat{}\ (psize\ p)))$
⟨*proof*⟩

lemma *basic-cqe-conv4*:
fixes $p\ q :: complex\ poly$
assumes *h*: $\bigwedge x.\ poly\ (q\ \hat{}\ n)\ x \equiv poly\ r\ x$
shows $p\ dvd\ (q\ \hat{}\ n) \equiv p\ dvd\ r$
⟨*proof*⟩

lemma *pmult-Cons-Cons*: $(pCons\ (a::complex)\ (pCons\ b\ p) * q = (smult\ a\ q) + (pCons\ 0\ (pCons\ b\ p * q)))$
⟨*proof*⟩

lemma *elim-neg-conv*: $-z \equiv (-1) * (z::complex)$ ⟨*proof*⟩
lemma *eqT-intr*: $PROP\ P \Longrightarrow (True \Longrightarrow PROP\ P) \ PROP\ P \Longrightarrow True$ ⟨*proof*⟩
lemma *negate-negate-rule*: $Trueprop\ P \equiv \neg P \equiv False$ ⟨*proof*⟩

lemma *complex-entire*: $(z::complex) \neq 0 \wedge w \neq 0 \equiv z*w \neq 0$ ⟨*proof*⟩
lemma *resolve-eq-ne*: $(P \equiv True) \equiv (\neg P \equiv False)\ (P \equiv False) \equiv (\neg P \equiv True)$
⟨*proof*⟩
lemma *cqe-conv1*: $poly\ 0\ x = 0 \longleftrightarrow True$ ⟨*proof*⟩
lemma *cqe-conv2*: $(p \Longrightarrow (q \equiv r)) \equiv ((p \wedge q) \equiv (p \wedge r))$ (**is** *?l* $\equiv$ *?r*)
⟨*proof*⟩

lemma *poly-const-conv*: *poly* [:*c*:] (*x*::*complex*) = *y* ⟷ *c* = *y* ⟨*proof*⟩

end

44 Infinite-Set: Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

44.1 Infinite Sets

Some elementary facts about infinite sets, mostly by Stefan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

abbreviation
infinite :: *'a set* ⇒ *bool* **where**
infinite S == ¬ *finite S*

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

lemma *infinite-imp-nonempty*: *infinite S* ==> *S* ≠ {}
⟨*proof*⟩

lemma *infinite-remove*:
infinite S ⟹ *infinite* (*S* − {*a*})
⟨*proof*⟩

lemma *Diff-infinite-finite*:
assumes *T*: *finite T* **and** *S*: *infinite S*
shows *infinite* (*S* − *T*)
⟨*proof*⟩

lemma *Un-infinite*: *infinite S* ⟹ *infinite* (*S* ∪ *T*)
⟨*proof*⟩

lemma *infinite-Un*: *infinite* (*S* ∪ *T*) ⟷ *infinite S* ∨ *infinite T*
⟨*proof*⟩

lemma *infinite-super*:
assumes *T*: *S* ⊆ *T* **and** *S*: *infinite S*
shows *infinite T*
⟨*proof*⟩

As a concrete example, we prove that the set of natural numbers is infinite.

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

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

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

lemma *infinite-nat-iff-unbounded*:
infinite (S::*nat set*) = ($\forall\, m.\ \exists\, n.\ m<n \wedge n \in S$)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *infinite-nat-iff-unbounded-le*:
infinite (S::*nat set*) = ($\forall\, m.\ \exists\, n.\ m \leq n \wedge n \in S$)
(**is** *?lhs* = *?rhs*)
⟨*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*:
assumes k: $\forall\, m.\ k<m \longrightarrow (\exists\, n.\ m<n \wedge n \in S)$
shows *infinite* (S::*nat set*)
⟨*proof*⟩

lemma *nat-infinite*: *infinite* (*UNIV* :: *nat set*)
⟨*proof*⟩

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

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S.

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

lemma *int-infinite* [*simp*]:
shows *infinite* (*UNIV*::*int set*)
⟨*proof*⟩

The "only if" direction is harder because it requires the construction of

a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

lemma *linorder-injI*:
assumes *hyp*: $!!x\ y.\ x < (y::'a::linorder) ==> f\ x \neq f\ y$
shows *inj f*
⟨*proof*⟩

lemma *infinite-countable-subset*:
assumes *inf*: *infinite* $(S::'a\ set)$
shows $\exists f.\ inj\ (f::nat \Rightarrow 'a) \wedge range\ f \subseteq S$
⟨*proof*⟩

lemma *infinite-iff-countable-subset*:
$infinite\ S = (\exists f.\ inj\ (f::nat \Rightarrow 'a) \wedge range\ f \subseteq S)$
⟨*proof*⟩

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

lemma *inf-img-fin-dom*:
assumes *img*: *finite* $(f`A)$ **and** *dom*: *infinite A*
shows $\exists y \in f`A.\ infinite\ (f -` \{y\})$
⟨*proof*⟩

lemma *inf-img-fin-domE*:
assumes *finite* $(f`A)$ **and** *infinite A*
obtains *y* **where** $y \in f`A$ **and** *infinite* $(f -` \{y\})$
⟨*proof*⟩

44.2 Infinitely Many and Almost All

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

definition
Inf-many :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** *INFM* 10) **where**
Inf-many P = *infinite* $\{x.\ P\ x\}$

definition
Alm-all :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** *MOST* 10) **where**
Alm-all P = $(\neg\ (INFM\ x.\ \neg\ P\ x))$

notation (*xsymbols*)
Inf-many (**binder** $\exists_\infty$ 10) **and**
Alm-all (**binder** $\forall_\infty$ 10)

notation (*HTML* **output**)
Inf-many (**binder** $\exists_\infty$ *10*) **and**
Alm-all (**binder** $\forall_\infty$ *10*)

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*⟩

lemmas *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

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-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-E*: **assumes** $INFM\ x.\ P\ x$ **obtains** x **where** $P\ x$
⟨*proof*⟩

lemma *MOST-I*: **assumes** $\bigwedge x.\ P\ x$ **shows** $MOST\ x.\ P\ x$
⟨*proof*⟩

lemma *INFM-mono*:
assumes *inf*: $\exists_\infty x.\ P\ x$ **and** *q*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\exists_\infty x.\ Q\ x$
⟨*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 *INFM-imp-distrib*:
$(INFM\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow ((MOST\ x.\ P\ x) \longrightarrow (INFM\ x.\ Q\ x))$
⟨*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-conjI*:
$INFM\ x.\ P\ x \Longrightarrow MOST\ x.\ Q\ x \Longrightarrow INFM\ x.\ P\ x \wedge Q\ x$
⟨*proof*⟩

lemma *MOST-rev-mp*:
assumes $\forall_\infty x.\ P\ x$ **and** $\forall_\infty x.\ P\ x \longrightarrow Q\ x$
shows $\forall_\infty x.\ Q\ x$
⟨*proof*⟩

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

lemma *INFM-MOST-simps* [*simp*]:
$\bigwedge P\ Q.\ (INFM\ x.\ P\ x \wedge Q) \longleftrightarrow (INFM\ x.\ P\ x) \wedge Q$
$\bigwedge P\ Q.\ (INFM\ x.\ P \wedge Q\ x) \longleftrightarrow P \wedge (INFM\ x.\ Q\ x)$
$\bigwedge P\ Q.\ (MOST\ x.\ P\ x \vee Q) \longleftrightarrow (MOST\ x.\ P\ x) \vee Q$
$\bigwedge P\ Q.\ (MOST\ x.\ P \vee Q\ x) \longleftrightarrow P \vee (MOST\ x.\ Q\ x)$
$\bigwedge P\ Q.\ (MOST\ x.\ P\ x \longrightarrow Q) \longleftrightarrow ((INFM\ x.\ P\ x) \longrightarrow Q)$
$\bigwedge P\ Q.\ (MOST\ x.\ P \longrightarrow Q\ x) \longleftrightarrow (P \longrightarrow (MOST\ x.\ Q\ x))$
⟨*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 ≠ a
MOST x. a ≠ x
⟨*proof*⟩

lemma *INFM-neq* [*simp*]:
(*INFM x::'a. x ≠ a*) ⟷ *infinite* (*UNIV::'a set*)
(*INFM x::'a. a ≠ x*) ⟷ *infinite* (*UNIV::'a set*)
⟨*proof*⟩

lemma *MOST-eq* [*simp*]:
(*MOST x::'a. x = a*) ⟷ *finite* (*UNIV::'a set*)
(*MOST x::'a. a = x*) ⟷ *finite* (*UNIV::'a set*)
⟨*proof*⟩

lemma *MOST-eq-imp*:
MOST x. x = a ⟶ *P x*
MOST x. a = x ⟶ *P x*
⟨*proof*⟩

Properties of quantifiers over the naturals.

lemma *INFM-nat*: (∃ ∞*n. P* (*n::nat*)) = (∀ *m.* ∃ *n. m<n* ∧ *P n*)
⟨*proof*⟩

lemma *INFM-nat-le*: (∃ ∞*n. P* (*n::nat*)) = (∀ *m.* ∃ *n. m≤n* ∧ *P n*)
⟨*proof*⟩

lemma *MOST-nat*: (∀ ∞*n. P* (*n::nat*)) = (∃ *m.* ∀ *n. m<n* ⟶ *P n*)
⟨*proof*⟩

lemma *MOST-nat-le*: (∀ ∞*n. P* (*n::nat*)) = (∃ *m.* ∀ *n. m≤n* ⟶ *P n*)
⟨*proof*⟩

44.3 Enumeration of an Infinite Set

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

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*⟩

44.4 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

definition
atmost-one :: *'a set* ⇒ *bool* **where**
atmost-one S = (∀ *x y*. *x*∈*S* ∧ *y*∈*S* ⟶ *x*=*y*)

lemma *atmost-one-empty*: *S* = {} ⟹ *atmost-one S*
⟨*proof*⟩

lemma *atmost-one-singleton*: *S* = {*x*} ⟹ *atmost-one S*
⟨*proof*⟩

lemma *atmost-one-unique* [*elim*]: *atmost-one S* ⟹ *x* ∈ *S* ⟹ *y* ∈ *S* ⟹ *y* = *x*
⟨*proof*⟩

end

45 Lattice-Syntax: Pretty syntax for lattice operations

theory *Lattice-Syntax*
imports *Complete-Lattice*
begin

notation
top (⊤) **and**
bot (⊥) **and**
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅ - [*900*] *900*) **and**
Sup (⨆ - [*900*] *900*)

end

46 ListVector: Lists as vectors

theory *ListVector*
imports *List Main*
begin

A vector-space like structure of lists and arithmetic operations on them. Is only a vector space if restricted to lists of the same length.

Multiplication with a scalar:

abbreviation *scale* :: ('*a*::*times*) ⇒ '*a list* ⇒ '*a list* (**infix** $*_s$ *70*)
where $x *_s xs \equiv map\ (op * x)\ xs$

lemma *scale1*[*simp*]: (*1*::'*a*::*monoid-mult*) $*_s$ *xs* = *xs*
⟨*proof*⟩

46.1 + and −

fun *zipwith0* :: ('*a*::*zero* ⇒ '*b*::*zero* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list*
where
zipwith0 f [] [] = [] |
zipwith0 f (*x*#*xs*) (*y*#*ys*) = *f x y* # *zipwith0 f xs ys* |
zipwith0 f (*x*#*xs*) [] = *f x 0* # *zipwith0 f xs* [] |
zipwith0 f [] (*y*#*ys*) = *f 0 y* # *zipwith0 f* [] *ys*

instantiation *list* :: ({*zero*, *plus*}) *plus*
begin

definition
list-add-def: *op* + = *zipwith0* (*op* +)

instance ⟨*proof*⟩

end

instantiation *list* :: ({*zero*, *uminus*}) *uminus*
begin

definition
list-uminus-def: *uminus* = *map uminus*

instance ⟨*proof*⟩

end

instantiation *list* :: ({*zero*,*minus*}) *minus*
begin

definition
list-diff-def: *op* − = *zipwith0* (*op* −)

instance ⟨*proof*⟩

end

lemma *zipwith0-Nil*[*simp*]: *zipwith0 f* [] *ys* = *map* (*f 0*) *ys*
⟨*proof*⟩

lemma *list-add-Nil*[*simp*]: [] + *xs* = (*xs*::'*a*::*monoid-add list*)
⟨*proof*⟩

lemma *list-add-Nil2*[*simp*]: *xs* + [] = (*xs*::'*a*::*monoid-add list*)
⟨*proof*⟩

lemma *list-add-Cons*[*simp*]: (*x*#*xs*) + (*y*#*ys*) = (*x*+*y*)#(*xs*+*ys*)
⟨*proof*⟩

lemma *list-diff-Nil*[*simp*]: [] − *xs* = −(*xs*::'*a*::*group-add list*)
⟨*proof*⟩

lemma *list-diff-Nil2*[*simp*]: *xs* − [] = (*xs*::'*a*::*group-add list*)
⟨*proof*⟩

lemma *list-diff-Cons-Cons*[*simp*]: (*x*#*xs*) − (*y*#*ys*) = (*x*−*y*)#(*xs*−*ys*)
⟨*proof*⟩

lemma *list-uminus-Cons*[*simp*]: −(*x*#*xs*) = (−*x*)#(−*xs*)
⟨*proof*⟩

lemma *self-list-diff*:
xs − *xs* = *replicate* (*length*(*xs*::'*a*::*group-add list*)) *0*
⟨*proof*⟩

lemma *list-add-assoc*: **fixes** *xs* :: '*a*::*monoid-add list*
shows (*xs*+*ys*)+*zs* = *xs*+(*ys*+*zs*)
⟨*proof*⟩

46.2 Inner product

definition *iprod* :: '*a*::*ring list* ⇒ '*a list* ⇒ '*a* (⟨-,-⟩) **where**
⟨*xs*,*ys*⟩ = (∑ (*x*,*y*) ← *zip xs ys*. *x*∗*y*)

lemma *iprod-Nil*[*simp*]: ⟨[],*ys*⟩ = *0*
⟨*proof*⟩

lemma *iprod-Nil2*[*simp*]: ⟨*xs*,[]⟩ = *0*
⟨*proof*⟩

lemma *iprod-Cons*[*simp*]: ⟨*x*#*xs*,*y*#*ys*⟩ = *x*∗*y* + ⟨*xs*,*ys*⟩

⟨*proof*⟩

lemma *iprod0-if-coeffs0*: $\forall\, c{\in}set\ cs.\ c = 0 \Longrightarrow \langle cs,xs\rangle = 0$
⟨*proof*⟩

lemma *iprod-uminus*[*simp*]: $\langle -xs,ys\rangle = -\langle xs,ys\rangle$
⟨*proof*⟩

lemma *iprod-left-add-distrib*: $\langle xs + ys,zs\rangle = \langle xs,zs\rangle + \langle ys,zs\rangle$
⟨*proof*⟩

lemma *iprod-left-diff-distrib*: $\langle xs - ys,\ zs\rangle = \langle xs,zs\rangle - \langle ys,zs\rangle$
⟨*proof*⟩

lemma *iprod-assoc*: $\langle x *_s xs,\ ys\rangle = x * \langle xs,ys\rangle$
⟨*proof*⟩

end

47 Kleene-Algebra: Kleene Algebras

theory *Kleene-Algebra*
imports *Main*
begin

WARNING: This is work in progress. Expect changes in the future.

Various lemmas correspond to entries in a database of theorems about Kleene algebras and related structures maintained by Peter Höfner: see http://www.informatik.uni-augsburg.de/~hoefnepe/kleene_db/lemmas/index.html.

47.1 Preliminaries

A class where addition is idempotent.

class *idem-add* = *plus* +
  assumes *add-idem* [*simp*]: $x + x = x$

A class of idempotent abelian semigroups (written additively).

class *idem-ab-semigroup-add* = *ab-semigroup-add* + *idem-add*
begin

lemma *add-idem2* [*simp*]: $x + (x + y) = x + y$
⟨*proof*⟩

lemma *add-idem3* [*simp*]: $x + (y + x) = x + y$
⟨*proof*⟩

end

A class where order is defined in terms of addition.

class *order-by-add* = *plus* + *ord* +
  assumes *order-def*: $x \leq y \longleftrightarrow x + y = y$
  assumes *strict-order-def*: $x < y \longleftrightarrow x \leq y \wedge \neg\, y \leq x$
begin

lemma *ord-simp* [*simp*]: $x \leq y \Longrightarrow x + y = y$
  ⟨*proof*⟩

lemma *ord-intro*: $x + y = y \Longrightarrow x \leq y$
  ⟨*proof*⟩

end

A class of idempotent abelian semigroups (written additively) where order is defined in terms of addition.

class *ordered-idem-ab-semigroup-add* = *idem-ab-semigroup-add* + *order-by-add*
begin

lemma *ord-simp2* [*simp*]: $x \leq y \Longrightarrow y + x = y$
  ⟨*proof*⟩

subclass *order* ⟨*proof*⟩

subclass *ordered-ab-semigroup-add* ⟨*proof*⟩

lemma *plus-leI* [*simp*]:
  $x \leq z \Longrightarrow y \leq z \Longrightarrow x + y \leq z$
  ⟨*proof*⟩

lemma *less-add* [*simp*]: $x \leq x + y\ y \leq x + y$
⟨*proof*⟩

lemma *add-est1* [*elim*]: $x + y \leq z \Longrightarrow x \leq z$
⟨*proof*⟩

lemma *add-est2* [*elim*]: $x + y \leq z \Longrightarrow y \leq z$
⟨*proof*⟩

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

end

A class of commutative monoids (written additively) where order is defined in terms of addition.

class *ordered-comm-monoid-add* = *comm-monoid-add* + *order-by-add*
begin

lemma *zero-minimum* [*simp*]: $0 \leq x$
⟨*proof*⟩

end

A class of idempotent commutative monoids (written additively) where order is defined in terms of addition.

class *ordered-idem-comm-monoid-add* = *ordered-comm-monoid-add* + *idem-add*
begin

subclass *ordered-idem-ab-semigroup-add* ⟨*proof*⟩

lemma *sum-is-zero*: $(x + y = 0) = (x = 0 \wedge y = 0)$
⟨*proof*⟩

end

47.2 A class of Kleene algebras

Class *pre-kleene* provides all operations of Kleene algebras except for the Kleene star.

class *pre-kleene* = *semiring-1* + *idem-add* + *order-by-add*
begin

subclass *ordered-idem-comm-monoid-add* ⟨*proof*⟩

subclass *ordered-semiring* ⟨*proof*⟩

end

A class that provides a star operator.

class *star* =
fixes *star* :: $'a \Rightarrow 'a$

Finally, a class of Kleene algebras.

class *kleene* = *pre-kleene* + *star* +
assumes *star1*: $1 + a * star\ a \leq star\ a$
and *star2*: $1 + star\ a * a \leq star\ a$
and *star3*: $a * x \leq x \Longrightarrow star\ a * x \leq x$
and *star4*: $x * a \leq x \Longrightarrow x * star\ a \leq x$
begin

lemma *star3'* [*simp*]:
assumes *a*: $b + a * x \leq x$
shows $star\ a * b \leq x$
⟨*proof*⟩

lemma *star4'* [*simp*]:
assumes *a*: $b + x * a \leq x$

shows $b * star\ a \leq x$
⟨*proof*⟩

lemma *star-unfold-left*: $1 + a * star\ a = star\ a$
⟨*proof*⟩

lemma *star-unfold-right*: $1 + star\ a * a = star\ a$
⟨*proof*⟩

lemma *star-zero* [*simp*]: $star\ 0 = 1$
⟨*proof*⟩

lemma *star-one* [*simp*]: $star\ 1 = 1$
⟨*proof*⟩

lemma *one-less-star* [*simp*]: $1 \leq star\ x$
⟨*proof*⟩

lemma *ka1* [*simp*]: $x * star\ x \leq star\ x$
⟨*proof*⟩

lemma *star-mult-idem* [*simp*]: $star\ x * star\ x = star\ x$
⟨*proof*⟩

lemma *less-star* [*simp*]: $x \leq star\ x$
⟨*proof*⟩

lemma *star-simulation-leq-1*:
assumes *a*: $a * x \leq x * b$
shows $star\ a * x \leq x * star\ b$
⟨*proof*⟩

lemma *star-simulation-leq-2*:
assumes *a*: $x * a \leq b * x$
shows $x * star\ a \leq star\ b * x$
⟨*proof*⟩

lemma *star-simulation* [*simp*]:
assumes *a*: $a * x = x * b$
shows $star\ a * x = x * star\ b$
⟨*proof*⟩

lemma *star-slide2* [*simp*]: $star\ x * x = x * star\ x$
⟨*proof*⟩

lemma *star-idemp* [*simp*]: $star\ (star\ x) = star\ x$
⟨*proof*⟩

lemma *star-slide* [*simp*]: $star\ (x * y) * x = x * star\ (y * x)$

⟨*proof*⟩

lemma *star-one'*:
assumes $p * p' = 1$ $p' * p = 1$
shows $p' * star\ a * p = star\ (p' * a * p)$
⟨*proof*⟩

lemma *x-less-star* [*simp*]: $x \leq x * star\ a$
⟨*proof*⟩

lemma *star-mono* [*simp*]: $x \leq y \Longrightarrow star\ x \leq star\ y$
⟨*proof*⟩

lemma *star-sub*: $x \leq 1 \Longrightarrow star\ x = 1$
⟨*proof*⟩

lemma *star-unfold2*: $star\ x * y = y + x * star\ x * y$
⟨*proof*⟩

lemma *star-absorb-one* [*simp*]: $star\ (x + 1) = star\ x$
⟨*proof*⟩

lemma *star-absorb-one'* [*simp*]: $star\ (1 + x) = star\ x$
⟨*proof*⟩

lemma *ka16*: $(y * star\ x) * star\ (y * star\ x) \leq star\ x * star\ (y * star\ x)$
⟨*proof*⟩

lemma *ka16'*: $(star\ x * y) * star\ (star\ x * y) \leq star\ (star\ x * y) * star\ x$
⟨*proof*⟩

lemma *ka17*: $(x * star\ x) * star\ (y * star\ x) \leq star\ x * star\ (y * star\ x)$
⟨*proof*⟩

lemma *ka18*: $(x * star\ x) * star\ (y * star\ x) + (y * star\ x) * star\ (y * star\ x)$
$\leq star\ x * star\ (y * star\ x)$
⟨*proof*⟩

lemma *star-decomp*: $star\ (x + y) = star\ x * star\ (y * star\ x)$
⟨*proof*⟩

lemma *ka22*: $y * star\ x \leq star\ x * star\ y \Longrightarrow\ star\ y * star\ x \leq star\ x * star\ y$
⟨*proof*⟩

lemma *ka23*: $star\ y * star\ x \leq star\ x * star\ y \Longrightarrow y * star\ x \leq star\ x * star\ y$
⟨*proof*⟩

lemma *ka24*: $star\ (x + y) \leq star\ (star\ x * star\ y)$
⟨*proof*⟩

lemma *ka25*: $star\ y * star\ x \leq star\ x * star\ y \Longrightarrow star\ (star\ y * star\ x) \leq star\ x * star\ y$
⟨*proof*⟩

lemma *church-rosser*:
$star\ y * star\ x \leq star\ x * star\ y \Longrightarrow star\ (x + y) \leq star\ x * star\ y$
⟨*proof*⟩

lemma *kleene-bubblesort*: $y * x \leq x * y \Longrightarrow star\ (x + y) \leq star\ x * star\ y$
⟨*proof*⟩

lemma *ka27*: $star\ (x + star\ y) = star\ (x + y)$
⟨*proof*⟩

lemma *ka28*: $star\ (star\ x + star\ y) = star\ (x + y)$
⟨*proof*⟩

lemma *ka29*: $(y * (1 + x) \leq (1 + x) * star\ y) = (y * x \leq (1 + x) * star\ y)$
⟨*proof*⟩

lemma *ka30*: $star\ x * star\ y \leq star\ (x + y)$
⟨*proof*⟩

lemma *simple-simulation*: $x * y = 0 \Longrightarrow star\ x * y = y$
⟨*proof*⟩

lemma *ka32*: $star\ (x * y) = 1 + x * star\ (y * x) * y$
⟨*proof*⟩

lemma *ka33*: $x * y + 1 \leq y \Longrightarrow star\ x \leq y$
⟨*proof*⟩

end

47.3 Complete lattices are Kleene algebras

lemma (**in** *complete-lattice*) *le-SUPI'*:
assumes $l \leq M\ i$
shows $l \leq (SUP\ i.\ M\ i)$
⟨*proof*⟩

class *kleene-by-complete-lattice* = *pre-kleene*
+ *complete-lattice* + *power* + *star* +
assumes *star-cont*: $a * star\ b * c = SUPR\ UNIV\ (\lambda n.\ a * b\ \hat{}\ n * c)$
begin

subclass *kleene*
⟨*proof*⟩

end

47.4 Transitive closure

context *kleene*
begin

definition
tcl-def: $tcl\ x = star\ x * x$

lemma *tcl-zero*: $tcl\ 0 = 0$
⟨*proof*⟩

lemma *tcl-unfold-right*: $tcl\ a = a + tcl\ a * a$
⟨*proof*⟩

lemma *less-tcl*: $a \leq tcl\ a$
⟨*proof*⟩

end

47.5 A naive algorithm to generate the transitive closure

function (*default* $\lambda x.\ 0$, *tailrec*, *domintros*)
mk-tcl :: ('*a*::{*plus*,*times*,*ord*,*zero*}) ⇒ '*a* ⇒ '*a*
where
$mk\text{-}tcl\ A\ X = (if\ X * A \leq X\ then\ X\ else\ mk\text{-}tcl\ A\ (X + X * A))$
⟨*proof*⟩

declare *mk-tcl.simps*[*simp del*]

lemma *mk-tcl-code*[*code*]:
$mk\text{-}tcl\ A\ X =$
$(let\ XA = X * A$
$in\ if\ XA \leq X\ then\ X\ else\ mk\text{-}tcl\ A\ (X + XA))$
⟨*proof*⟩

context *kleene*
begin

lemma *mk-tcl-lemma1*: $(X + X * A) * star\ A = X * star\ A$
⟨*proof*⟩

lemma *mk-tcl-lemma2*: $X * A \leq X \Longrightarrow X * star\ A = X$
⟨*proof*⟩

end

lemma *mk-tcl-correctness*:

fixes $X :: 'a::kleene$
assumes *mk-tcl-dom* (A, X)
shows *mk-tcl* $A\ X = X * star\ A$
⟨*proof*⟩

lemma *graph-implies-dom*: *mk-tcl-graph* $x\ y \Longrightarrow$ *mk-tcl-dom* x
⟨*proof*⟩

lemma *mk-tcl-default*: $\neg$ *mk-tcl-dom* $(a,x) \Longrightarrow$ *mk-tcl* $a\ x = 0$
⟨*proof*⟩

We can replace the dom-Condition of the correctness theorem with something executable:

lemma *mk-tcl-correctness2*:
fixes $A\ X :: 'a :: \{kleene\}$
assumes *mk-tcl* $A\ A \neq 0$
shows *mk-tcl* $A\ A = tcl\ A$
⟨*proof*⟩

end

48 Multiset: (Finite) multisets

theory *Multiset*
imports *Main*
begin

48.1 The type of multisets

typedef $'a$ *multiset* $= \{f :: 'a => nat.\ finite\ \{x.\ f\ x > 0\}\}$
morphisms *count Abs-multiset*
⟨*proof*⟩

lemmas *multiset-typedef* = *Abs-multiset-inverse count-inverse count*

abbreviation *Melem* $:: 'a => 'a\ multiset => bool$ $((-/ :\# -)\ [50, 51]\ 50)$ **where**
$a :\# M == 0 < count\ M\ a$

notation (*xsymbols*)
Melem (**infix** $\in\#$ *50*)

lemma *multiset-ext-iff*:
$M = N \longleftrightarrow (\forall a.\ count\ M\ a = count\ N\ a)$
⟨*proof*⟩

lemma *multiset-ext*:
$(\bigwedge x.\ count\ A\ x = count\ B\ x) \Longrightarrow A = B$
⟨*proof*⟩

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*:
$(\lambda a.\ 0) \in multiset$
⟨*proof*⟩

lemma *only1-in-multiset*:
$(\lambda b.\ if\ b = a\ then\ n\ else\ 0) \in multiset$
⟨*proof*⟩

lemma *union-preserves-multiset*:
$M \in multiset \Longrightarrow N \in multiset \Longrightarrow (\lambda a.\ M\ a + N\ a) \in multiset$
⟨*proof*⟩

lemma *diff-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda a.\ M\ a - N\ a) \in multiset$
⟨*proof*⟩

lemma *MCollect-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 MCollect-preserves-multiset

48.2 Representing multisets

Multiset comprehension

definition *MCollect* :: *'a multiset => ('a => bool) => 'a multiset* **where**
MCollect M P = Abs-multiset $(\lambda x.\ if\ P\ x\ then\ count\ M\ x\ else\ 0)$

syntax
-MCollect :: *pttrn => 'a multiset => bool => 'a multiset* ((1{# - :# -./ -#}))
translations
{#*x* :# *M*. *P*#} == *CONST MCollect M* $(\lambda x.\ P)$

Multiset enumeration

instantiation *multiset* :: (*type*) {*zero*, *plus*}
begin

definition *Mempty-def*:
0 = Abs-multiset $(\lambda a.\ 0)$

abbreviation *Mempty* :: *'a multiset* ({#}) **where**
Mempty ≡ *0*

definition *union-def*:
M + N = Abs-multiset $(\lambda a.\ count\ M\ a + count\ N\ a)$

instance ⟨*proof*⟩

end

definition *single* :: *'a => 'a multiset* **where**
single a = Abs-multiset (λb. if b = a then 1 else 0)

syntax
-multiset :: *args => 'a multiset* ({#(-)#})
translations
{#*x*, *xs*#} == {#*x*#} + {#*xs*#}
{#*x*#} == *CONST single x*

lemma *count-empty* [*simp*]: *count* {#} *a = 0*
⟨*proof*⟩

lemma *count-single* [*simp*]: *count* {#*b*#} *a* = (*if b = a then 1 else 0*)
⟨*proof*⟩

48.3 Basic operations

48.3.1 Union

lemma *count-union* [*simp*]: *count* (*M* + *N*) *a* = *count M a* + *count N a*
⟨*proof*⟩

instance *multiset* :: (*type*) *cancel-comm-monoid-add* ⟨*proof*⟩

48.3.2 Difference

instantiation *multiset* :: (*type*) *minus*
begin

definition *diff-def*:
M − *N* = *Abs-multiset* (λ*a*. *count M a* − *count N a*)

instance ⟨*proof*⟩

end

lemma *count-diff* [*simp*]: *count* (*M* − *N*) *a* = *count M a* − *count N a*
⟨*proof*⟩

lemma *diff-empty* [*simp*]: *M* − {#} = *M* ∧ {#} − *M* = {#}
⟨*proof*⟩

lemma *diff-cancel*[*simp*]: *A* − *A* = {#}
⟨*proof*⟩

lemma *diff-union-cancelR* [*simp*]: $M + N - N = (M::'a\ multiset)$
⟨*proof*⟩

lemma *diff-union-cancelL* [*simp*]: $N + M - N = (M::'a\ multiset)$
⟨*proof*⟩

lemma *insert-DiffM*:
$x \in\# M \Longrightarrow \{\#x\#\} + (M - \{\#x\#\}) = M$
⟨*proof*⟩

lemma *insert-DiffM2* [*simp*]:
$x \in\# M \Longrightarrow M - \{\#x\#\} + \{\#x\#\} = M$
⟨*proof*⟩

lemma *diff-right-commute*:
$(M::'a\ multiset) - N - Q = M - Q - N$
⟨*proof*⟩

lemma *diff-add*:
$(M::'a\ multiset) - (N + Q) = M - N - Q$
⟨*proof*⟩

lemma *diff-union-swap*:
$a \neq b \Longrightarrow M - \{\#a\#\} + \{\#b\#\} = M + \{\#b\#\} - \{\#a\#\}$
⟨*proof*⟩

lemma *diff-union-single-conv*:
$a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

48.3.3 Equality of multisets

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
⟨*proof*⟩

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*]: $M = M + \{\#x\#\} \longleftrightarrow False$
⟨*proof*⟩

lemma *diff-single-trivial*:
$\neg\ x \in\# M \Longrightarrow M - \{\#x\#\} = M$

⟨*proof*⟩

lemma *diff-single-eq-union*:
$x \in\# M \Longrightarrow M - \{\#x\#\} = N \longleftrightarrow M = N + \{\#x\#\}$
⟨*proof*⟩

lemma *union-single-eq-diff*:
$M + \{\#x\#\} = N \Longrightarrow M = N - \{\#x\#\}$
⟨*proof*⟩

lemma *union-single-eq-member*:
$M + \{\#x\#\} = N \Longrightarrow x \in\# N$
⟨*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*:
$M + \{\#a\#\} = N + \{\#b\#\} \longleftrightarrow M = N \wedge a = b \vee M = N - \{\#a\#\} + \{\#b\#\} \wedge N = M - \{\#b\#\} + \{\#a\#\}$ (**is** *?lhs = ?rhs*)

⟨*proof*⟩

lemma *insert-noteq-member*:
assumes *BC*: $B + \{\#b\#\} = C + \{\#c\#\}$
and *bnotc*: $b \neq c$
shows $c \in\# B$
⟨*proof*⟩

lemma *add-eq-conv-ex*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee (\exists K.\ M = K + \{\#b\#\} \wedge N = K + \{\#a\#\}))$
⟨*proof*⟩

48.3.4 Pointwise ordering induced by count

instantiation *multiset* :: (*type*) *ordered-ab-semigroup-add-imp-le*
begin

definition *less-eq-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
mset-le-def: $A \leq B \longleftrightarrow (\forall a.\ count\ A\ a \leq count\ B\ a)$

definition *less-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
mset-less-def: $(A{::}'a\ multiset) < B \longleftrightarrow A \leq B \wedge A \neq B$

instance ⟨*proof*⟩

end

lemma *mset-less-eqI*:
(⋀*x*. *count A x* ≤ *count B x*) ⟹ *A* ≤ *B*
⟨*proof*⟩

lemma *mset-le-exists-conv*:
(*A*::'*a multiset*) ≤ *B* ⟷ (∃ *C*. *B* = *A* + *C*)
⟨*proof*⟩

lemma *mset-le-mono-add-right-cancel* [*simp*]:
(*A*::'*a multiset*) + *C* ≤ *B* + *C* ⟷ *A* ≤ *B*
⟨*proof*⟩

lemma *mset-le-mono-add-left-cancel* [*simp*]:
C + (*A*::'*a multiset*) ≤ *C* + *B* ⟷ *A* ≤ *B*
⟨*proof*⟩

lemma *mset-le-mono-add*:
(*A*::'*a multiset*) ≤ *B* ⟹ *C* ≤ *D* ⟹ *A* + *C* ≤ *B* + *D*
⟨*proof*⟩

lemma *mset-le-add-left* [*simp*]:
(*A*::'*a multiset*) ≤ *A* + *B*
⟨*proof*⟩

lemma *mset-le-add-right* [*simp*]:
B ≤ (*A*::'*a multiset*) + *B*
⟨*proof*⟩

lemma *mset-le-single*:
a :# *B* ⟹ {#*a*#} ≤ *B*
⟨*proof*⟩

lemma *multiset-diff-union-assoc*:
C ≤ *B* ⟹ (*A*::'*a multiset*) + *B* − *C* = *A* + (*B* − *C*)
⟨*proof*⟩

lemma *mset-le-multiset-union-diff-commute*:
B ≤ *A* ⟹ (*A*::'*a multiset*) − *B* + *C* = *A* + *C* − *B*
⟨*proof*⟩

lemma *mset-lessD*: *A* < *B* ⟹ *x* ∈# *A* ⟹ *x* ∈# *B*
⟨*proof*⟩

lemma *mset-leD*: *A* ≤ *B* ⟹ *x* ∈# *A* ⟹ *x* ∈# *B*

⟨*proof*⟩

lemma *mset-less-insertD*: $(A + \{\#x\#\} < B) \Longrightarrow (x \in\# B \wedge A < B)$
⟨*proof*⟩

lemma *mset-le-insertD*: $(A + \{\#x\#\} \leq B) \Longrightarrow (x \in\# B \wedge A \leq B)$
⟨*proof*⟩

lemma *mset-less-of-empty*[*simp*]: $A < \{\#\} \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *multi-psub-of-add-self*[*simp*]: $A < A + \{\#x\#\}$
⟨*proof*⟩

lemma *multi-psub-self*[*simp*]: $(A::'a\ multiset) < A = \mathit{False}$
⟨*proof*⟩

lemma *mset-less-add-bothsides*:
$T + \{\#x\#\} < S + \{\#x\#\} \Longrightarrow T < S$
⟨*proof*⟩

lemma *mset-less-empty-nonempty*:
$\{\#\} < S \longleftrightarrow S \neq \{\#\}$
⟨*proof*⟩

lemma *mset-less-diff-self*:
$c \in\# B \Longrightarrow B - \{\#c\#\} < B$
⟨*proof*⟩

48.3.5 Intersection

instantiation *multiset* :: (*type*) *semilattice-inf*
begin

definition *inf-multiset* :: $'a\ multiset \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ **where**
multiset-inter-def: $inf\text{-}multiset\ A\ B = A - (A - B)$

instance ⟨*proof*⟩

end

abbreviation *multiset-inter* :: $'a\ multiset \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ (**infixl** #∩ *70*) **where**
$multiset\text{-}inter \equiv inf$

lemma *multiset-inter-count*:
$count\ (A\ \#\cap\ B)\ x = min\ (count\ A\ x)\ (count\ B\ x)$
⟨*proof*⟩

lemma *multiset-inter-single*: $a \neq b \Longrightarrow \{\#a\#\} \mathbin{\#\cap} \{\#b\#\} = \{\#\}$
⟨*proof*⟩

lemma *multiset-union-diff-commute*:
assumes $B \mathbin{\#\cap} C = \{\#\}$
shows $A + B - C = A - C + B$
⟨*proof*⟩

48.3.6 Comprehension (filter)

lemma *count-MCollect* [*simp*]:
count $\{\#\ x{:}\#M.\ P\ x\ \#\}\ a = (\textit{if}\ P\ a\ \textit{then}\ \textit{count}\ M\ a\ \textit{else}\ 0)$
⟨*proof*⟩

lemma *MCollect-empty* [*simp*]: *MCollect* $\{\#\}\ P = \{\#\}$
⟨*proof*⟩

lemma *MCollect-single* [*simp*]:
MCollect $\{\#x\#\}\ P = (\textit{if}\ P\ x\ \textit{then}\ \{\#x\#\}\ \textit{else}\ \{\#\})$
⟨*proof*⟩

lemma *MCollect-union* [*simp*]:
MCollect $(M + N)\ f = \textit{MCollect}\ M\ f + \textit{MCollect}\ N\ f$
⟨*proof*⟩

48.3.7 Set of elements

definition *set-of* :: *'a multiset => 'a set* **where**
set-of $M = \{x.\ x :\# M\}$

lemma *set-of-empty* [*simp*]: *set-of* $\{\#\} = \{\}$
⟨*proof*⟩

lemma *set-of-single* [*simp*]: *set-of* $\{\#b\#\} = \{b\}$
⟨*proof*⟩

lemma *set-of-union* [*simp*]: *set-of* $(M + N) = \textit{set-of}\ M \cup \textit{set-of}\ N$
⟨*proof*⟩

lemma *set-of-eq-empty-iff* [*simp*]: $(\textit{set-of}\ M = \{\}) = (M = \{\#\})$
⟨*proof*⟩

lemma *mem-set-of-iff* [*simp*]: $(x \in \textit{set-of}\ M) = (x :\# M)$
⟨*proof*⟩

lemma *set-of-MCollect* [*simp*]: *set-of* $\{\#\ x{:}\#M.\ P\ x\ \#\} = \textit{set-of}\ M \cap \{x.\ P\ x\}$
⟨*proof*⟩

lemma *finite-set-of* [*iff*]: *finite* $(\textit{set-of}\ M)$
⟨*proof*⟩

48.3.8 Size

instantiation *multiset* :: (*type*) *size*
begin

definition *size-def*:
size M = *setsum* (*count M*) (*set-of M*)

instance ⟨*proof*⟩

end

lemma *size-empty* [*simp*]: *size* {#} = *0*
⟨*proof*⟩

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
⟨*proof*⟩

lemma *setsum-count-Int*:
finite A ==> *setsum* (*count N*) (*A* ∩ *set-of N*) = *setsum* (*count N*) *A*
⟨*proof*⟩

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
⟨*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* = *N* + {#*a*#}
⟨*proof*⟩

48.4 Induction and case splits

lemma *setsum-decr*:
finite F ==> (*0*::*nat*) < *f a* ==>
setsum (*f* (*a* := *f a* − *1*)) *F* = (*if a*∈*F then setsum f F* − *1 else setsum f F*)
⟨*proof*⟩

lemma *rep-multiset-induct-aux*:
assumes *1*: *P* (λ*a*. (*0*::*nat*))
and *2*: !!*f b*. *f* ∈ *multiset* ==> *P f* ==> *P* (*f* (*b* := *f b* + *1*))
shows ∀ *f*. *f* ∈ *multiset* −−> *setsum f* {*x*. *f x* ≠ *0*} = *n* −−> *P f*

⟨*proof*⟩

theorem *rep-multiset-induct*:
  $f \in multiset \Longrightarrow P\ (\lambda a.\ 0) \Longrightarrow$
    $(!!f\ b.\ f \in multiset \Longrightarrow P\ f \Longrightarrow P\ (f\ (b := f\ b + 1))) \Longrightarrow P\ f$
⟨*proof*⟩

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
assumes *empty*: $P\ \{\#\}$
  and *add*: $!!M\ x.\ P\ M \Longrightarrow P\ (M + \{\#x\#\})$
shows $P\ M$
⟨*proof*⟩

lemma *multi-nonempty-split*: $M \neq \{\#\} \Longrightarrow \exists A\ a.\ M = A + \{\#a\#\}$
⟨*proof*⟩

lemma *multiset-cases* [*cases type*, *case-names empty add*]:
assumes *em*: $M = \{\#\} \Longrightarrow P$
assumes *add*: $\bigwedge N\ x.\ M = N + \{\#x\#\} \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *multi-member-split*: $x \in\# M \Longrightarrow \exists A.\ M = A + \{\#x\#\}$
⟨*proof*⟩

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
⟨*proof*⟩

lemma *multiset-partition*: $M = \{\#\ x{:}\#M.\ P\ x\ \#\} + \{\#\ x{:}\#M.\ \neg\ P\ x\ \#\}$
⟨*proof*⟩

lemma *mset-less-size*: $(A::'a\ multiset) < B \Longrightarrow size\ A < size\ B$
⟨*proof*⟩

48.4.1 Strong induction and subset induction for multisets

Well-foundedness of proper subset operator:

proper multiset subset

definition
  mset-less-rel :: $('a\ multiset * 'a\ multiset)\ set$ **where**
  $mset\text{-}less\text{-}rel = \{(A,B).\ A < B\}$

lemma *multiset-add-sub-el-shuffle*:
  assumes $c \in\# B$ **and** $b \neq c$
  shows $B - \{\#c\#\} + \{\#b\#\} = B + \{\#b\#\} - \{\#c\#\}$
⟨*proof*⟩

lemma *wf-mset-less-rel*: *wf mset-less-rel*

⟨*proof*⟩

The induction rules:

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall (A::'a\ multiset).\ A < B \longrightarrow P\ A \Longrightarrow P\ B$
shows $P\ B$
⟨*proof*⟩

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \leq A$
and *empty*: $P\ \{\#\}$
and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (F + \{\#a\#\})$
shows $P\ F$
⟨*proof*⟩

48.5 Alternative representations

48.5.1 Lists

primrec *multiset-of* :: $'a\ list \Rightarrow 'a\ multiset$ **where**
multiset-of [] = {#} |
multiset-of (*a* # *x*) = *multiset-of x* + {# *a* #}

lemma *in-multiset-in-set*:
$x \in\#$ *multiset-of xs* $\longleftrightarrow x \in$ *set xs*
⟨*proof*⟩

lemma *count-multiset-of*:
count (*multiset-of xs*) *x* = *length* (*filter* (λ*y*. *x* = *y*) *xs*)
⟨*proof*⟩

lemma *multiset-of-zero-iff*[*simp*]: (*multiset-of x* = {#}) = (*x* = [])
⟨*proof*⟩

lemma *multiset-of-zero-iff-right*[*simp*]: ({#} = *multiset-of x*) = (*x* = [])
⟨*proof*⟩

lemma *set-of-multiset-of*[*simp*]: *set-of*(*multiset-of x*) = *set x*
⟨*proof*⟩

lemma *mem-set-multiset-eq*: *x* ∈ *set xs* = (*x* :# *multiset-of xs*)
⟨*proof*⟩

lemma *multiset-of-append* [*simp*]:
multiset-of (*xs* @ *ys*) = *multiset-of xs* + *multiset-of ys*
⟨*proof*⟩

lemma *surj-multiset-of*: *surj multiset-of*
⟨*proof*⟩

lemma *set-count-greater-0*: *set x = {a. count (multiset-of x) a > 0}*
⟨*proof*⟩

lemma *distinct-count-atmost-1*:
distinct x = (! a. count (multiset-of x) a = (if a ∈ set x then 1 else 0))
⟨*proof*⟩

lemma *multiset-of-eq-setD*:
multiset-of xs = multiset-of ys ⟹ set xs = set ys
⟨*proof*⟩

lemma *set-eq-iff-multiset-of-eq-distinct*:
distinct x ⟹ distinct y ⟹
(set x = set y) = (multiset-of x = multiset-of y)
⟨*proof*⟩

lemma *set-eq-iff-multiset-of-remdups-eq*:
(set x = set y) = (multiset-of (remdups x) = multiset-of (remdups y))
⟨*proof*⟩

lemma *multiset-of-compl-union* [*simp*]:
multiset-of [x←xs. P x] + multiset-of [x←xs. ¬P x] = multiset-of xs
⟨*proof*⟩

lemma *count-filter*:
count (multiset-of xs) x = length [y ← xs. y = x]
⟨*proof*⟩

lemma *nth-mem-multiset-of*: *i < length ls ⟹ (ls ! i) :# multiset-of ls*
⟨*proof*⟩

lemma *multiset-of-remove1* [*simp*]:
multiset-of (remove1 a xs) = multiset-of xs − {#a#}
⟨*proof*⟩

lemma *multiset-of-eq-length*:
assumes *multiset-of xs = multiset-of ys*
shows *length xs = length ys*
⟨*proof*⟩

lemma (**in** *linorder*) *multiset-of-insort* [*simp*]:
multiset-of (insort x xs) = {#x#} + multiset-of xs
⟨*proof*⟩

lemma (**in** *linorder*) *multiset-of-sort* [*simp*]:
multiset-of (sort xs) = multiset-of xs
⟨*proof*⟩

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

lemma (**in** *linorder*) *properties-for-sort*:
multiset-of ys = multiset-of xs ⟹ sorted ys ⟹ sort xs = ys
⟨*proof*⟩

lemma *multiset-of-remdups-le*: *multiset-of* (*remdups xs*) ≤ *multiset-of xs*
⟨*proof*⟩

lemma *multiset-of-update*:
i < length ls ⟹ multiset-of (*ls*[*i* := *v*]) = *multiset-of ls* − {#*ls* ! *i*#} + {#*v*#}
⟨*proof*⟩

lemma *multiset-of-swap*:
i < length ls ⟹ j < length ls ⟹
multiset-of (*ls*[*j* := *ls* ! *i*, *i* := *ls* ! *j*]) = *multiset-of ls*
⟨*proof*⟩

48.5.2 Association lists – including rudimentary code generation

definition *count-of* :: ('*a* × *nat*) *list* ⇒ '*a* ⇒ *nat* **where**
count-of xs x = (*case map-of xs x of None* ⇒ *0* | *Some n* ⇒ *n*)

lemma *count-of-multiset*:
count-of xs ∈ *multiset*
⟨*proof*⟩

lemma *count-simps* [*simp*]:
count-of [] = (λ-. *0*)
count-of ((*x*, *n*) # *xs*) = (λ*y*. *if x* = *y then n else count-of xs y*)
⟨*proof*⟩

lemma *count-of-empty*:
x ∉ *fst* ‘ *set xs ⟹ count-of xs x* = *0*
⟨*proof*⟩

lemma *count-of-filter*:
count-of (*filter* (*P* ∘ *fst*) *xs*) *x* = (*if P x then count-of xs x else 0*)
⟨*proof*⟩

definition *Bag* :: ('*a* × *nat*) *list* ⇒ '*a multiset* **where**
Bag xs = *Abs-multiset* (*count-of xs*)

code-datatype *Bag*

lemma *count-Bag* [*simp*, *code*]:
count (*Bag xs*) = *count-of xs*
⟨*proof*⟩

lemma *Mempty-Bag* [*code*]:
{#} = *Bag* []

⟨*proof*⟩

lemma *single-Bag* [*code*]:
{#*x*#} = *Bag* [(*x*, *1*)]
⟨*proof*⟩

lemma *MCollect-Bag* [*code*]:
MCollect (*Bag xs*) *P* = *Bag* (*filter* (*P* ∘ *fst*) *xs*)
⟨*proof*⟩

lemma *mset-less-eq-Bag* [*code*]:
Bag xs ≤ *A* ⟷ (∀ (*x*, *n*) ∈ *set xs*. *count-of xs x* ≤ *count A x*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

instantiation *multiset* :: (*eq*) *eq*
begin

definition
HOL.eq A B ⟷ (*A*::'*a multiset*) ≤ *B* ∧ *B* ≤ *A*

instance ⟨*proof*⟩

end

definition (**in** *term-syntax*)
bagify :: ('*a*::*typerep* × *nat*) *list* × (*unit* ⇒ *Code-Evaluation.term*)
⇒ '*a multiset* × (*unit* ⇒ *Code-Evaluation.term*) **where**
[*code-unfold*]: *bagify xs* = *Code-Evaluation.valtermify Bag* {·} *xs*

notation *fcomp* (**infixl** *o*> *60*)
notation *scomp* (**infixl** *o*→ *60*)

instantiation *multiset* :: (*random*) *random*
begin

definition
Quickcheck.random i = *Quickcheck.random i o*→ (λ*xs*. *Pair* (*bagify xs*))

instance ⟨*proof*⟩

end

no-notation *fcomp* (**infixl** *o*> *60*)
no-notation *scomp* (**infixl** *o*→ *60*)

hide-const (**open**) *bagify*

48.6 The multiset order

48.6.1 Well-foundedness

definition *mult1* :: $('a \times 'a)\ set \Rightarrow ('a\ multiset \times 'a\ multiset)\ set$ **where**
[*code del*]: $mult1\ r = \{(N, M).\ \exists a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge (\forall b.\ b :\# K \longrightarrow (b, a) \in r)\}$

definition *mult* :: $('a \times 'a)\ set \Rightarrow ('a\ multiset \times 'a\ multiset)\ set$ **where**
[*code del*]: $mult\ r = (mult1\ r)^+$

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin mult1\ r$
⟨*proof*⟩

lemma *less-add*: $(N, M0 + \{\#a\#\}) \in mult1\ r \Longrightarrow$
$(\exists M.\ (M, M0) \in mult1\ r \wedge N = M + \{\#a\#\}) \vee$
$(\exists K.\ (\forall b.\ b :\# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$
(**is** - $\Longrightarrow$ *?case1* $(mult1\ r) \vee$ *?case2*)
⟨*proof*⟩

lemma *all-accessible*: $wf\ r \Longrightarrow \forall M.\ M \in acc\ (mult1\ r)$
⟨*proof*⟩

theorem *wf-mult1*: $wf\ r \Longrightarrow wf\ (mult1\ r)$
⟨*proof*⟩

theorem *wf-mult*: $wf\ r \Longrightarrow wf\ (mult\ r)$
⟨*proof*⟩

48.6.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:
$trans\ r \Longrightarrow (M, N) \in mult\ r \Longrightarrow$
$\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge$
$(\forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r)$
⟨*proof*⟩

lemma *one-step-implies-mult-aux*:
$trans\ r \Longrightarrow$
$\forall I\ J\ K.\ (size\ J = n \wedge J \neq \{\#\} \wedge (\forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r))$
$\longrightarrow (I + K, I + J) \in mult\ r$
⟨*proof*⟩

lemma *one-step-implies-mult*:
$trans\ r \Longrightarrow J \neq \{\#\} \Longrightarrow \forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r$
$\Longrightarrow (I + K, I + J) \in mult\ r$
⟨*proof*⟩

48.6.3 Partial-order properties

definition *less-multiset* :: *'a::order multiset ⇒ 'a multiset ⇒ bool* (**infix** <# *50*)
where
M' <# M ⟷ (M', M) ∈ mult {(x', x). x' < x}

definition *le-multiset* :: *'a::order multiset ⇒ 'a multiset ⇒ bool* (**infix** <=# *50*)
where
M' <=# M ⟷ M' <# M ∨ M' = M

notation (*xsymbols*) *less-multiset* (**infix** ⊂# *50*)
notation (*xsymbols*) *le-multiset* (**infix** ⊆# *50*)

interpretation *multiset-order*: *order le-multiset less-multiset*
⟨*proof*⟩

lemma *mult-less-irrefl* [*elim*!]:
M ⊂# (M::'a::order multiset) ==> R
⟨*proof*⟩

48.6.4 Monotonicity of multiset union

lemma *mult1-union*:
(B, D) ∈ mult1 r ==> trans r ==> (C + B, C + D) ∈ mult1 r
⟨*proof*⟩

lemma *union-less-mono2*: *B ⊂# D ==> C + B ⊂# C + (D::'a::order multiset)*
⟨*proof*⟩

lemma *union-less-mono1*: *B ⊂# D ==> B + C ⊂# D + (C::'a::order multiset)*
⟨*proof*⟩

lemma *union-less-mono*:
A ⊂# C ==> B ⊂# D ==> A + B ⊂# C + (D::'a::order multiset)
⟨*proof*⟩

interpretation *multiset-order*: *ordered-ab-semigroup-add plus le-multiset less-multiset*
⟨*proof*⟩

48.7 The fold combinator

The intended behaviour is *fold-mset f z* $\{\#x_1, ..., x_n\#\} = f\ x_1\ (\ldots\ (f\ x_n\ z)\ldots)$ if f is associative-commutative.

The graph of *fold-mset*, z: the start element, f: folding function, A: the multiset, y: the result.

inductive
fold-msetG :: *('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a multiset ⇒ 'b ⇒ bool*
for *f* :: *'a ⇒ 'b ⇒ 'b*
and *z* :: *'b*

where
  emptyI [*intro*]: *fold-msetG f z* {#} *z*
| *insertI* [*intro*]: *fold-msetG f z A y* ⟹ *fold-msetG f z* (*A* + {#*x*#}) (*f x y*)

inductive-cases *empty-fold-msetGE* [*elim*!]: *fold-msetG f z* {#} *x*
inductive-cases *insert-fold-msetGE*: *fold-msetG f z* (*A* + {#}) *y*

definition
  fold-mset :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ '*a multiset* ⇒ '*b* **where**
  fold-mset f z A = (*THE x*. *fold-msetG f z A x*)

lemma *Diff1-fold-msetG*:
  fold-msetG f z (*A* − {#*x*#}) *y* ⟹ *x* ∈# *A* ⟹ *fold-msetG f z A* (*f x y*)
⟨*proof*⟩

lemma *fold-msetG-nonempty*: ∃ *x*. *fold-msetG f z A x*
⟨*proof*⟩

lemma *fold-mset-empty*[*simp*]: *fold-mset f z* {#} = *z*
⟨*proof*⟩

context *fun-left-comm*
begin

lemma *fold-msetG-determ*:
  fold-msetG f z A x ⟹ *fold-msetG f z A y* ⟹ *y* = *x*
⟨*proof*⟩

lemma *fold-mset-insert-aux*:
  (*fold-msetG f z* (*A* + {#*x*#}) *v*) =
    (∃ *y*. *fold-msetG f z A y* ∧ *v* = *f x y*)
⟨*proof*⟩

lemma *fold-mset-equality*: *fold-msetG f z A y* ⟹ *fold-mset f z A* = *y*
⟨*proof*⟩

lemma *fold-mset-insert*:
  fold-mset f z (*A* + {#*x*#}) = *f x* (*fold-mset f z A*)
⟨*proof*⟩

lemma *fold-mset-insert-idem*:
  fold-mset f z (*A* + {#*a*#}) = *f a* (*fold-mset f z A*)
⟨*proof*⟩

lemma *fold-mset-commute*: *f x* (*fold-mset f z A*) = *fold-mset f* (*f x z*) *A*
⟨*proof*⟩

lemma *fold-mset-single* [*simp*]: *fold-mset f z* {#*x*#} = *f x z*
⟨*proof*⟩

lemma *fold-mset-union* [*simp*]:
fold-mset f z (*A*+*B*) = *fold-mset f* (*fold-mset f z A*) *B*
⟨*proof*⟩

lemma *fold-mset-fusion*:
assumes *fun-left-comm g*
shows (⋀*x y*. *h* (*g x y*) = *f x* (*h y*)) ⟹ *h* (*fold-mset g w A*) = *fold-mset f* (*h w*) *A* (**is** *PROP ?P*)
⟨*proof*⟩

lemma *fold-mset-rec*:
assumes *a* ∈# *A*
shows *fold-mset f z A* = *f a* (*fold-mset f z* (*A* − {#*a*#}))
⟨*proof*⟩

end

A note on code generation: When defining some function containing a subterm *fold-mset F*, code generation is not automatic. When interpreting locale *left-commutative* with *F*, the would be code thms for *fold-mset* become thms like *fold-mset F z* {#} = *z* where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

48.8 Image

definition *image-mset* :: ('*a* ⇒ '*b*) ⇒ '*a multiset* ⇒ '*b multiset* **where**
image-mset f = *fold-mset* (*op* + ∘ *single* ∘ *f*) {#}

interpretation *image-left-comm*: *fun-left-comm op* + ∘ *single* ∘ *f*
⟨*proof*⟩

lemma *image-mset-empty* [*simp*]: *image-mset f* {#} = {#}
⟨*proof*⟩

lemma *image-mset-single* [*simp*]: *image-mset f* {#*x*#} = {#*f x*#}
⟨*proof*⟩

lemma *image-mset-insert*:
image-mset f (*M* + {#*a*#}) = *image-mset f M* + {#*f a*#}
⟨*proof*⟩

lemma *image-mset-union* [*simp*]:
image-mset f (*M*+*N*) = *image-mset f M* + *image-mset f N*
⟨*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*⟩

syntax
-comprehension1-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset*
(({#-/. - :# -#}))
translations
{#*e*. *x*:#*M*#} == *CONST image-mset* (%*x*. *e*) *M*

syntax
-comprehension2-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset*
(({#-/ | - :# -./ -#}))
translations
{#*e* | *x*:#*M*. *P*#} => {#*e*. *x* :# {# *x*:#*M*. *P*#}#}

This allows to write not just filters like {# *x* :# *M*. *x* < *c*#} but also images like {#*x* + *x*. *x* :# *M*#} and {#*x*+*x*|*x*:#*M*. *x*<*c*#}, where the latter is currently displayed as {#*x* + *x*. *x* :# {# *x* :# *M*. *x* < *c*#}#}.

48.9 Termination proofs with multiset orders

lemma *multi-member-skip*: *x* ∈# *XS* ⟹ *x* ∈# {# *y* #} + *XS*
and *multi-member-this*: *x* ∈# {# *x* #} + *XS*
and *multi-member-last*: *x* ∈# {# *x* #}
⟨*proof*⟩

definition *ms-strict* = *mult pair-less*
definition [*code del*]: *ms-weak* = *ms-strict* ∪ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
⟨*proof*⟩

lemma *smsI*:
(*set-of A*, *set-of B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-strict*
⟨*proof*⟩

lemma *wmsI*:
(*set-of A*, *set-of 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*:

$(x, y) \in \mathit{pair\text{-}leq} \Longrightarrow \mathit{pw\text{-}leq}\ \{\#x\#\}\ \{\#y\#\}$
⟨*proof*⟩

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows $\exists A\ B\ Z.\ X = A + Z \wedge Y = B + Z \wedge ((\mathit{set\text{-}of}\ A, \mathit{set\text{-}of}\ B) \in \mathit{max\text{-}strict} \vee (B = \{\#\} \wedge A = \{\#\}))$
⟨*proof*⟩

lemma
assumes *pwleq*: *pw-leq Z Z'*
shows *ms-strictI*: $(\mathit{set\text{-}of}\ A, \mathit{set\text{-}of}\ B) \in \mathit{max\text{-}strict} \Longrightarrow (Z + A, Z' + B) \in \mathit{ms\text{-}strict}$
and *ms-weakI1*: $(\mathit{set\text{-}of}\ A, \mathit{set\text{-}of}\ B) \in \mathit{max\text{-}strict} \Longrightarrow (Z + A, Z' + B) \in \mathit{ms\text{-}weak}$
and *ms-weakI2*: $(Z + \{\#\}, Z' + \{\#\}) \in \mathit{ms\text{-}weak}$
⟨*proof*⟩

lemma *empty-idemp*: $\{\#\} + x = x\ x + \{\#\} = x$
and *nonempty-plus*: $\{\#\ x\ \#\} + rs \neq \{\#\}$
and *nonempty-single*: $\{\#\ x\ \#\} \neq \{\#\}$
⟨*proof*⟩

⟨*ML*⟩

48.10 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-ext-iff* [*symmetric*]

lemma *union-commute*: $M + N = N + (M::'a\ \mathit{multiset})$
⟨*proof*⟩

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ \mathit{multiset}))$
⟨*proof*⟩

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ \mathit{multiset}))$
⟨*proof*⟩

lemmas *union-ac* = *union-assoc union-commute union-lcomm*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ \mathit{multiset})$
⟨*proof*⟩

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ \mathit{multiset})$
⟨*proof*⟩

lemma *multi-union-self-other-eq*: $(A::'a\ \mathit{multiset}) + X = A + Y \Longrightarrow X = Y$
⟨*proof*⟩

lemma *mset-less-trans*: $(M::'a\ multiset) < K \Longrightarrow K < N \Longrightarrow M < N$
⟨*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 *mult-less-not-refl*:
$\neg\ M \subset\# (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *mult-less-trans*:
$K \subset\# M ==> M \subset\# N ==> K \subset\# (N::'a::order\ multiset)$
⟨*proof*⟩

lemma *mult-less-not-sym*:
$M \subset\# N ==> \neg\ N \subset\# (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *mult-less-asym*:
$M \subset\# N ==> (\neg\ P ==> N \subset\# (M::'a::order\ multiset)) ==> P$
⟨*proof*⟩

⟨*ML*⟩

end

49 Nat-Infinity: Natural numbers with infinity

theory *Nat-Infinity*
imports *Main*
begin

49.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

datatype *inat* = *Fin nat* | *Infty*

notation (*xsymbols*)
Infty (∞)

notation (*HTML* **output**)
Infty (∞)

lemma *not-Infty-eq*[*iff*]: (*x* ~= *Infty*) = (*EX i*. *x* = *Fin i*)
⟨*proof*⟩

lemma *not-Fin-eq* [*iff*]: (*ALL y*. *x* ~= *Fin y*) = (*x* = *Infty*)
⟨*proof*⟩

49.2 Constructors and numbers

instantiation *inat* :: {*zero*, *one*, *number*}
begin

definition
0 = *Fin 0*

definition
[*code-unfold*]: *1* = *Fin 1*

definition
[*code-unfold*, *code del*]: *number-of k* = *Fin* (*number-of k*)

instance ⟨*proof*⟩

end

definition *iSuc* :: *inat* ⇒ *inat* **where**
iSuc i = (*case i of Fin n* ⇒ *Fin* (*Suc n*) | ∞ ⇒ ∞)

lemma *Fin-0*: *Fin 0* = *0*
⟨*proof*⟩

lemma *Fin-1*: *Fin 1* = *1*
⟨*proof*⟩

lemma *Fin-number*: *Fin* (*number-of k*) = *number-of k*
⟨*proof*⟩

lemma *one-iSuc*: *1* = *iSuc 0*
⟨*proof*⟩

lemma *Infty-ne-i0* [*simp*]: ∞ ≠ *0*

⟨*proof*⟩

lemma *i0-ne-Infty* [*simp*]: $0 \neq \infty$
⟨*proof*⟩

lemma *zero-inat-eq* [*simp*]:
$number\text{-}of\ k = (0::inat) \longleftrightarrow number\text{-}of\ k = (0::nat)$
$(0::inat) = number\text{-}of\ k \longleftrightarrow number\text{-}of\ k = (0::nat)$
⟨*proof*⟩

lemma *one-inat-eq* [*simp*]:
$number\text{-}of\ k = (1::inat) \longleftrightarrow number\text{-}of\ k = (1::nat)$
$(1::inat) = number\text{-}of\ k \longleftrightarrow number\text{-}of\ k = (1::nat)$
⟨*proof*⟩

lemma *zero-one-inat-neq* [*simp*]:
$\neg\ 0 = (1::inat)$
$\neg\ 1 = (0::inat)$
⟨*proof*⟩

lemma *Infty-ne-i1* [*simp*]: $\infty \neq 1$
⟨*proof*⟩

lemma *i1-ne-Infty* [*simp*]: $1 \neq \infty$
⟨*proof*⟩

lemma *Infty-ne-number* [*simp*]: $\infty \neq number\text{-}of\ k$
⟨*proof*⟩

lemma *number-ne-Infty* [*simp*]: $number\text{-}of\ k \neq \infty$
⟨*proof*⟩

lemma *iSuc-Fin*: $iSuc\ (Fin\ n) = Fin\ (Suc\ n)$
⟨*proof*⟩

lemma *iSuc-number-of*: $iSuc\ (number\text{-}of\ k) = Fin\ (Suc\ (number\text{-}of\ k))$
⟨*proof*⟩

lemma *iSuc-Infty* [*simp*]: $iSuc\ \infty = \infty$
⟨*proof*⟩

lemma *iSuc-ne-0* [*simp*]: $iSuc\ n \neq 0$
⟨*proof*⟩

lemma *zero-ne-iSuc* [*simp*]: $0 \neq iSuc\ n$
⟨*proof*⟩

lemma *iSuc-inject* [*simp*]: $iSuc\ m = iSuc\ n \longleftrightarrow m = n$
⟨*proof*⟩

lemma *number-of-inat-inject* [*simp*]:
(*number-of k* :: *inat*) = *number-of l* ⟷ (*number-of k* :: *nat*) = *number-of l*
⟨*proof*⟩

49.3 Addition

instantiation *inat* :: *comm-monoid-add*
begin

definition
[*code del*]: *m* + *n* = (*case m of* ∞ ⇒ ∞ | *Fin m* ⇒ (*case n of* ∞ ⇒ ∞ | *Fin n* ⇒ *Fin* (*m* + *n*)))

lemma *plus-inat-simps* [*simp*, *code*]:
Fin m + *Fin n* = *Fin* (*m* + *n*)
∞ + *q* = ∞
q + ∞ = ∞
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *plus-inat-0* [*simp*]:
0 + (*q*::*inat*) = *q*
(*q*::*inat*) + *0* = *q*
⟨*proof*⟩

lemma *plus-inat-number* [*simp*]:
(*number-of k* :: *inat*) + *number-of l* = (*if k* < *Int.Pls then number-of l*
else if l < *Int.Pls then number-of k else number-of* (*k* + *l*))
⟨*proof*⟩

lemma *iSuc-number* [*simp*]:
iSuc (*number-of k*) = (*if neg* (*number-of k* :: *int*) *then 1 else number-of* (*Int.succ k*))
⟨*proof*⟩

lemma *iSuc-plus-1*:
iSuc n = *n* + *1*
⟨*proof*⟩

lemma *plus-1-iSuc*:
1 + *q* = *iSuc q*
q + *1* = *iSuc q*
⟨*proof*⟩

49.4 Multiplication

instantiation *inat* :: *comm-semiring-1*
begin

definition
times-inat-def [*code del*]:
$m * n = (case\ m\ of\ \infty \Rightarrow if\ n = 0\ then\ 0\ else\ \infty \mid Fin\ m \Rightarrow$
$(case\ n\ of\ \infty \Rightarrow if\ m = 0\ then\ 0\ else\ \infty \mid Fin\ n \Rightarrow Fin\ (m * n)))$

lemma *times-inat-simps* [*simp*, *code*]:
$Fin\ m * Fin\ n = Fin\ (m * n)$
$\infty * \infty = \infty$
$\infty * Fin\ n = (if\ n = 0\ then\ 0\ else\ \infty)$
$Fin\ m * \infty = (if\ m = 0\ then\ 0\ else\ \infty)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *mult-iSuc*: $iSuc\ m * n = n + m * n$
⟨*proof*⟩

lemma *mult-iSuc-right*: $m * iSuc\ n = m + m * n$
⟨*proof*⟩

lemma *of-nat-eq-Fin*: $of\text{-}nat\ n = Fin\ n$
⟨*proof*⟩

instance *inat* :: *semiring-char-0*
⟨*proof*⟩

49.5 Ordering

instantiation *inat* :: *linordered-ab-semigroup-add*
begin

definition
[*code del*]: $m \leq n = (case\ n\ of\ Fin\ n1 \Rightarrow (case\ m\ of\ Fin\ m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow False)$
$\mid \infty \Rightarrow True)$

definition
[*code del*]: $m < n = (case\ m\ of\ Fin\ m1 \Rightarrow (case\ n\ of\ Fin\ n1 \Rightarrow m1 < n1 \mid \infty \Rightarrow True)$
$\mid \infty \Rightarrow False)$

lemma *inat-ord-simps* [*simp*]:
$Fin\ m \leq Fin\ n \longleftrightarrow m \leq n$

$Fin\ m < Fin\ n \longleftrightarrow m < n$
$q \leq \infty$
$q < \infty \longleftrightarrow q \neq \infty$
$\infty \leq q \longleftrightarrow q = \infty$
$\infty < q \longleftrightarrow False$
⟨*proof*⟩

lemma *inat-ord-code* [*code*]:
$Fin\ m \leq Fin\ n \longleftrightarrow m \leq n$
$Fin\ m < Fin\ n \longleftrightarrow m < n$
$q \leq \infty \longleftrightarrow True$
$Fin\ m < \infty \longleftrightarrow True$
$\infty \leq Fin\ n \longleftrightarrow False$
$\infty < q \longleftrightarrow False$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *inat* :: *ordered-comm-semiring*
⟨*proof*⟩

lemma *inat-ord-number* [*simp*]:
$(number\text{-}of\ m :: inat) \leq number\text{-}of\ n \longleftrightarrow (number\text{-}of\ m :: nat) \leq number\text{-}of\ n$
$(number\text{-}of\ m :: inat) < number\text{-}of\ n \longleftrightarrow (number\text{-}of\ m :: nat) < number\text{-}of\ n$
⟨*proof*⟩

lemma *i0-lb* [*simp*]: $(0::inat) \leq n$
⟨*proof*⟩

lemma *i0-neq* [*simp*]: $n \leq (0::inat) \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *Infty-ileE* [*elim!*]: $\infty \leq Fin\ m \Longrightarrow R$
⟨*proof*⟩

lemma *Infty-ilessE* [*elim!*]: $\infty < Fin\ m \Longrightarrow R$
⟨*proof*⟩

lemma *not-ilessi0* [*simp*]: $\neg\ n < (0::inat)$
⟨*proof*⟩

lemma *i0-eq* [*simp*]: $(0::inat) < n \longleftrightarrow n \neq 0$
⟨*proof*⟩

lemma *iSuc-ile-mono* [*simp*]: $iSuc\ n \leq iSuc\ m \longleftrightarrow n \leq m$
⟨*proof*⟩

lemma *iSuc-mono* [*simp*]: $iSuc\ n < iSuc\ m \longleftrightarrow n < m$
⟨*proof*⟩

lemma *ile-iSuc* [*simp*]: $n \leq iSuc\ n$
⟨*proof*⟩

lemma *not-iSuc-ilei0* [*simp*]: $\neg\ iSuc\ n \leq 0$
⟨*proof*⟩

lemma *i0-iless-iSuc* [*simp*]: $0 < iSuc\ n$
⟨*proof*⟩

lemma *ileI1*: $m < n \Longrightarrow iSuc\ m \leq n$
⟨*proof*⟩

lemma *Suc-ile-eq*: $Fin\ (Suc\ m) \leq n \longleftrightarrow Fin\ m < n$
⟨*proof*⟩

lemma *iless-Suc-eq* [*simp*]: $Fin\ m < iSuc\ n \longleftrightarrow Fin\ m \leq n$
⟨*proof*⟩

lemma *min-inat-simps* [*simp*]:
$min\ (Fin\ m)\ (Fin\ n) = Fin\ (min\ m\ n)$
$min\ q\ 0 = 0$
$min\ 0\ q = 0$
$min\ q\ \infty = q$
$min\ \infty\ q = q$
⟨*proof*⟩

lemma *max-inat-simps* [*simp*]:
$max\ (Fin\ m)\ (Fin\ n) = Fin\ (max\ m\ n)$
$max\ q\ 0 = q$
$max\ 0\ q = q$
$max\ q\ \infty = \infty$
$max\ \infty\ q = \infty$
⟨*proof*⟩

lemma *Fin-ile*: $n \leq Fin\ m \Longrightarrow \exists\, k.\ n = Fin\ k$
⟨*proof*⟩

lemma *Fin-iless*: $n < Fin\ m \Longrightarrow \exists\, k.\ n = Fin\ k$
⟨*proof*⟩

lemma *chain-incr*: $\forall\, i.\ \exists\, j.\ Y\ i < Y\ j \Longrightarrow \exists\, j.\ Fin\ k < Y\ j$
⟨*proof*⟩

instantiation *inat* :: {*bot*, *top*}
begin

definition *bot-inat* :: *inat* **where**
bot-inat = *0*

definition *top-inat* :: *inat* **where**
top-inat = ∞

instance ⟨*proof*⟩

end

49.6 Well-ordering

lemma *less-FinE*:
[| *n* < *Fin m*; !!*k*. *n* = *Fin k* ==> *k* < *m* ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *less-InftyE*:
[| *n* < *Infty*; !!*k*. *n* = *Fin k* ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *inat-less-induct*:
assumes *prem*: !!*n*. ∀ *m*::*inat*. *m* < *n* −−> *P m* ==> *P n* **shows** *P n*
⟨*proof*⟩

instance *inat* :: *wellorder*
⟨*proof*⟩

49.7 Traditional theorem names

lemmas *inat-defs* = *zero-inat-def one-inat-def number-of-inat-def iSuc-def*
plus-inat-def less-eq-inat-def less-inat-def

lemmas *inat-splits* = *inat.splits*

end

50 Nested-Environment: Nested environments

theory *Nested-Environment*
imports *Main*
begin

Consider a partial function *e* :: *'a* => *'b option*; this may be understood as an *environment* mapping indexes *'a* to optional entry values *'b* (cf. the basic theory *Map* of Isabelle/HOL). This basic idea is easily generalized to that of a *nested environment*, where entries may be either basic values or again proper environments. Then each entry is accessed by a *path*, i.e. a list of indexes leading to its position within the structure.

datatype $('a, 'b, 'c)$ *env* =
  Val $'a$
| *Env* $'b$ $'c$ => $('a, 'b, 'c)$ *env option*

In the type $('a, 'b, 'c)$ *env* the parameter $'a$ refers to basic values (occurring in terminal positions), type $'b$ to values associated with proper (inner) environments, and type $'c$ with the index type for branching. Note that there is no restriction on any of these types. In particular, arbitrary branching may yield rather large (transfinite) tree structures.

50.1 The lookup operation

Lookup in nested environments works by following a given path of index elements, leading to an optional result (a terminal value or nested environment). A *defined position* within a nested environment is one where *lookup* at its path does not yield *None*.

primrec
  lookup :: $('a, 'b, 'c)$ *env* => $'c$ *list* => $('a, 'b, 'c)$ *env option*
  and *lookup-option* :: $('a, 'b, 'c)$ *env option* => $'c$ *list* => $('a, 'b, 'c)$ *env option*
where
  lookup (*Val a*) *xs* = (*if xs* = [] *then Some* (*Val a*) *else None*)
| *lookup* (*Env b es*) *xs* =
    (*case xs of*
      [] => *Some* (*Env b es*)
    | *y* # *ys* => *lookup-option* (*es y*) *ys*)
| *lookup-option None xs* = *None*
| *lookup-option* (*Some e*) *xs* = *lookup e xs*

hide-const *lookup-option*

The characteristic cases of *lookup* are expressed by the following equalities.

theorem *lookup-nil*: *lookup e* [] = *Some e*
  ⟨*proof*⟩

theorem *lookup-val-cons*: *lookup* (*Val a*) (*x* # *xs*) = *None*
  ⟨*proof*⟩

theorem *lookup-env-cons*:
  lookup (*Env b es*) (*x* # *xs*) =
    (*case es x of*
      None => *None*
    | *Some e* => *lookup e xs*)
  ⟨*proof*⟩

lemmas *lookup-lookup-option.simps* [*simp del*]
  and *lookup-simps* [*simp*] = *lookup-nil lookup-val-cons lookup-env-cons*

theorem *lookup-eq*:
lookup env xs =
(*case xs of*
[] => *Some env*
| *x* # *xs* =>
(*case env of*
Val a => *None*
| *Env b es* =>
(*case es x of*
None => *None*
| *Some e* => *lookup e xs*)))
⟨*proof*⟩

Displaced *lookup* operations, relative to a certain base path prefix, may be reduced as follows. There are two cases, depending whether the environment actually extends far enough to follow the base path.

theorem *lookup-append-none*:
assumes *lookup env xs* = *None*
shows *lookup env* (*xs* @ *ys*) = *None*
⟨*proof*⟩

theorem *lookup-append-some*:
assumes *lookup env xs* = *Some e*
shows *lookup env* (*xs* @ *ys*) = *lookup e ys*
⟨*proof*⟩

Successful *lookup* deeper down an environment structure means we are able to peek further up as well. Note that this is basically just the contrapositive statement of *lookup-append-none* above.

theorem *lookup-some-append*:
assumes *lookup env* (*xs* @ *ys*) = *Some e*
shows ∃ *e*. *lookup env xs* = *Some e*
⟨*proof*⟩

The subsequent statement describes in more detail how a successful *lookup* with a non-empty path results in a certain situation at any upper position.

theorem *lookup-some-upper*:
assumes *lookup env* (*xs* @ *y* # *ys*) = *Some e*
shows ∃ *b′ es′ env′*.
lookup env xs = *Some* (*Env b′ es′*) ∧
es′ y = *Some env′* ∧
lookup env′ ys = *Some e*
⟨*proof*⟩

50.2 The update operation

Update at a certain position in a nested environment may either delete an existing entry, or overwrite an existing one. Note that update at undefined positions is simple absorbed, i.e. the environment is left unchanged.

primrec
update :: *'c list* => (*'a*, *'b*, *'c*) *env option*
=> (*'a*, *'b*, *'c*) *env* => (*'a*, *'b*, *'c*) *env*
and *update-option* :: *'c list* => (*'a*, *'b*, *'c*) *env option*
=> (*'a*, *'b*, *'c*) *env option* => (*'a*, *'b*, *'c*) *env option* **where**
update xs opt (*Val a*) =
(*if xs* = [] *then* (*case opt of None* => *Val a* | *Some e* => *e*)
else Val a)
| *update xs opt* (*Env b es*) =
(*case xs of*
[] => (*case opt of None* => *Env b es* | *Some e* => *e*)
| *y* # *ys* => *Env b* (*es* (*y* := *update-option ys opt* (*es y*))))
| *update-option xs opt None* =
(*if xs* = [] *then opt else None*)
| *update-option xs opt* (*Some e*) =
(*if xs* = [] *then opt else Some* (*update xs opt e*))

hide-const *update-option*

The characteristic cases of *update* are expressed by the following equalities.

theorem *update-nil-none*: *update* [] *None env* = *env*
⟨*proof*⟩

theorem *update-nil-some*: *update* [] (*Some e*) *env* = *e*
⟨*proof*⟩

theorem *update-cons-val*: *update* (*x* # *xs*) *opt* (*Val a*) = *Val a*
⟨*proof*⟩

theorem *update-cons-nil-env*:
update [*x*] *opt* (*Env b es*) = *Env b* (*es* (*x* := *opt*))
⟨*proof*⟩

theorem *update-cons-cons-env*:
update (*x* # *y* # *ys*) *opt* (*Env b es*) =
Env b (*es* (*x* :=
(*case es x of*
None => *None*
| *Some e* => *Some* (*update* (*y* # *ys*) *opt e*))))
⟨*proof*⟩

lemmas *update-update-option.simps* [*simp del*]
and *update-simps* [*simp*] = *update-nil-none update-nil-some*

update-cons-val update-cons-nil-env update-cons-cons-env

lemma *update-eq*:
```
update xs opt env =
  (case xs of
    [] =>
      (case opt of
        None => env
      | Some e => e)
  | x # xs =>
      (case env of
        Val a => Val a
      | Env b es =>
          (case xs of
            [] => Env b (es (x := opt))
          | y # ys =>
              Env b (es (x :=
                (case es x of
                  None => None
                | Some e => Some (update (y # ys) opt e)))))))
```
⟨*proof*⟩

The most basic correspondence of *lookup* and *update* states that after *update* at a defined position, subsequent *lookup* operations would yield the new value.

theorem *lookup-update-some*:
assumes *lookup env xs = Some e*
shows *lookup (update xs (Some env') env) xs = Some env'*
⟨*proof*⟩

The properties of displaced *update* operations are analogous to those of *lookup* above. There are two cases: below an undefined position *update* is absorbed altogether, and below a defined positions *update* affects subsequent *lookup* operations in the obvious way.

theorem *update-append-none*:
assumes *lookup env xs = None*
shows *update (xs @ y # ys) opt env = env*
⟨*proof*⟩

theorem *update-append-some*:
assumes *lookup env xs = Some e*
shows *lookup (update (xs @ y # ys) opt env) xs = Some (update (y # ys) opt e)*
⟨*proof*⟩

Apparently, *update* does not affect the result of subsequent *lookup* operations at independent positions, i.e. in case that the paths for *update* and *lookup* fork at a certain point.

theorem *lookup-update-other*:
assumes *neq*: $y \neq (z::'c)$
shows *lookup* (*update* (*xs* @ *z* # *zs*) *opt env*) (*xs* @ *y* # *ys*) =
lookup env (*xs* @ *y* # *ys*)
⟨*proof*⟩

Environments and code generation

lemma [*code*, *code del*]:
fixes *e1 e2* :: ('*b*::*eq*, '*a*::*eq*, '*c*::*eq*) *env*
shows *eq-class.eq e1 e2* ⟷ *eq-class.eq e1 e2* ⟨*proof*⟩

lemma *eq-env-code* [*code*]:
fixes *x y* :: '*a*::*eq*
and *f g* :: '*c*::{*eq*, *finite*} ⇒ ('*b*::*eq*, '*a*, '*c*) *env option*
shows *eq-class.eq* (*Env x f*) (*Env y g*) ⟷
eq-class.eq x y ∧ (∀ *z*∈*UNIV*. *case f z*
of None ⇒ (*case g z*
of None ⇒ *True* | *Some* - ⇒ *False*)
| *Some a* ⇒ (*case g z*
of None ⇒ *False* | *Some b* ⇒ *eq-class.eq a b*)) (**is** *?env*)
and *eq-class.eq* (*Val a*) (*Val b*) ⟷ *eq-class.eq a b*
and *eq-class.eq* (*Val a*) (*Env y g*) ⟷ *False*
and *eq-class.eq* (*Env x f*) (*Val b*) ⟷ *False*
⟨*proof*⟩

lemma [*code*, *code del*]:
(*Code-Evaluation.term-of* :: ('*a*::{*term-of*, *type*}, '*b*::{*term-of*, *type*}, '*c*::{*term-of*, *type*}) *env* ⇒ *term*) = *Code-Evaluation.term-of* ⟨*proof*⟩

end

51 Numeral-Type: Numeral Syntax for Types

theory *Numeral-Type*
imports *Main*
begin

51.1 Preliminary lemmas

lemma (**in** *type-definition*) *univ*:
UNIV = *Abs* ' *A*
⟨*proof*⟩

lemma (**in** *type-definition*) *card*: *card* (*UNIV* :: '*b set*) = *card A*
⟨*proof*⟩

51.2 Cardinalities of types

syntax *-type-card* :: *type* => *nat* ((*1CARD*/(*1'*(*-'*))))

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

⟨*ML*⟩

lemma *card-unit* [*simp*]: *CARD(unit) = 1*
⟨*proof*⟩

lemma *card-bool* [*simp*]: *CARD(bool) = 2*
⟨*proof*⟩

lemma *card-prod* [*simp*]: *CARD('a × 'b) = CARD('a::finite) * CARD('b::finite)*
⟨*proof*⟩

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

lemma *card-option* [*simp*]: *CARD('a option) = Suc CARD('a::finite)*
⟨*proof*⟩

lemma *card-set* [*simp*]: *CARD('a set) = 2 ^ CARD('a::finite)*
⟨*proof*⟩

lemma *card-nat* [*simp*]: *CARD(nat) = 0*
⟨*proof*⟩

51.3 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 ≤ CARD('a::finite)*
⟨*proof*⟩

Class for cardinality "at least 2"

class *card2 = finite +*
assumes *two-le-card*: *2 ≤ CARD('a)*

lemma *one-less-card*: *Suc 0 < CARD('a::card2)*
⟨*proof*⟩

lemma *one-less-int-card*: *1 < int CARD('a::card2)*
⟨*proof*⟩

51.4 Numeral Types

typedef (**open**) *num0* = *UNIV* :: *nat set* ⟨*proof*⟩
typedef (**open**) *num1* = *UNIV* :: *unit set* ⟨*proof*⟩

typedef (**open**) *'a bit0* = {*0* ..< *2* ∗ *int CARD*('a::*finite*)}
⟨*proof*⟩

typedef (**open**) *'a bit1* = {*0* ..< *1* + *2* ∗ *int CARD*('a::*finite*)}
⟨*proof*⟩

lemma *card-num0* [*simp*]: *CARD* (*num0*) = *0*
⟨*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*⟩

51.5 Locale 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* +
constrains $n :: int$
and $Rep :: 'a::\{number\text{-}ring\} \Rightarrow int$
and $Abs :: int \Rightarrow 'a::\{number\text{-}ring\}$

begin

lemma *of-nat-eq*: *of-nat k = Abs (int k mod n)*
⟨*proof*⟩

lemma *of-int-eq*: *of-int z = Abs (z mod n)*
⟨*proof*⟩

lemma *Rep-number-of*:
Rep (number-of w) = number-of w mod n
⟨*proof*⟩

lemma *iszero-number-of*:
iszero (number-of w::'a) ⟷ number-of w mod n = 0
⟨*proof*⟩

lemma *cases*:
assumes *1*: $\bigwedge z.\ [\![(x::'a) = \textit{of-int}\ z;\ 0 \leq z;\ z < n]\!] \Longrightarrow P$
shows *P*
⟨*proof*⟩

lemma *induct*:
$(\bigwedge z.\ [\![0 \leq z;\ z < n]\!] \Longrightarrow P\ (\textit{of-int}\ z)) \Longrightarrow P\ (x::'a)$
⟨*proof*⟩

end

51.6 Number ring instances

Unfortunately a number ring instance is not possible for *num1*, since 0 and 1 are not distinct.

instantiation *num1* :: {*comm-ring*,*comm-monoid-mult*,*number*}
begin

lemma *num1-eq-iff*: *(x::num1) = (y::num1) ⟷ 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*⟩

instantiation *bit0* **and** *bit1* :: (*finite*) *number-ring*
begin

definition (*number-of* w :: - *bit0*) = *of-int* w

definition (*number-of* w :: - *bit1*) = *of-int* w

instance ⟨*proof*⟩

end

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-number-of* [*simp*] = *bit0.iszero-number-of*
lemmas *bit1-iszero-number-of* [*simp*] = *bit1.iszero-number-of*

51.7 Syntax

syntax
-NumeralType :: num-const => type (-)
-NumeralType0 :: type (*0*)
-NumeralType1 :: type (*1*)

translations
(*type*) *1* == (*type*) *num1*
(*type*) *0* == (*type*) *num0*

⟨*ML*⟩

51.8 Examples

lemma *CARD(0) = 0* ⟨*proof*⟩
lemma *CARD(17) = 17* ⟨*proof*⟩
lemma *8 * 11 ^ 3 - 6 = (2::5)* ⟨*proof*⟩

end

⟨*ML*⟩

52 Option-ord: Canonical order on option type

theory *Option-ord*
imports *Option Main*
begin

instantiation *option* :: (*preorder*) *preorder*
begin

definition *less-eq-option* **where**
[*code del*]: $x \leq y \longleftrightarrow$ (*case x of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ (*case y of None* $\Rightarrow$ *False* | *Some y* $\Rightarrow$ $x \leq y$))

definition *less-option* **where**
[*code del*]: $x < y \longleftrightarrow$ (*case y of None* $\Rightarrow$ *False* | *Some y* $\Rightarrow$ (*case x of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ $x < y$))

lemma *less-eq-option-None* [*simp*]: *None* $\leq x$
⟨*proof*⟩

lemma *less-eq-option-None-code* [*code*]: *None* $\leq x \longleftrightarrow$ *True*
⟨*proof*⟩

lemma *less-eq-option-None-is-None*: $x \leq$ *None* $\Longrightarrow x =$ *None*
⟨*proof*⟩

lemma *less-eq-option-Some-None* [*simp*, *code*]: *Some x* $\leq$ *None* $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *less-eq-option-Some* [*simp*, *code*]: *Some x* $\leq$ *Some y* $\longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *less-option-None* [*simp*, *code*]: $x <$ *None* $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *less-option-None-is-Some*: *None* $< x \Longrightarrow \exists z.\ x =$ *Some z*
⟨*proof*⟩

lemma *less-option-None-Some* [*simp*]: *None* $<$ *Some x*
⟨*proof*⟩

lemma *less-option-None-Some-code* [*code*]: *None* $<$ *Some x* $\longleftrightarrow$ *True*
⟨*proof*⟩

lemma *less-option-Some* [*simp*, *code*]: *Some x* $<$ *Some y* $\longleftrightarrow x < y$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *option* :: (*order*) *order* ⟨*proof*⟩

instance *option* :: (*linorder*) *linorder* ⟨*proof*⟩

instantiation *option* :: (*preorder*) *bot*
begin

definition *bot* = *None*

instance ⟨*proof*⟩

end

instantiation *option* :: (*top*) *top*
begin

definition *top* = *Some top*

instance ⟨*proof*⟩

end

instance *option* :: (*wellorder*) *wellorder* ⟨*proof*⟩

end

53 Permutation: Permutations

theory *Permutation*
imports *Main Multiset*
begin

inductive
  perm :: *'a list* => *'a list* => *bool* (- <~~> - [*50*, *50*] *50*)
  where
    Nil [*intro*!]: [] <~~> []
  | *swap* [*intro*!]: *y* # *x* # *l* <~~> *x* # *y* # *l*
  | *Cons* [*intro*!]: *xs* <~~> *ys* ==> *z* # *xs* <~~> *z* # *ys*
  | *trans* [*intro*]: *xs* <~~> *ys* ==> *ys* <~~> *zs* ==> *xs* <~~> *zs*

lemma *perm-refl* [*iff*]: *l* <~~> *l*
  ⟨*proof*⟩

53.1 Some examples of rule induction on permutations

lemma *xperm-empty-imp*: [] <~~> *ys* ==> *ys* = []

⟨*proof*⟩

This more general theorem is easier to understand!

lemma *perm-length*: *xs* <~~> *ys* ==> *length xs* = *length ys*
⟨*proof*⟩

lemma *perm-empty-imp*: [] <~~> *xs* ==> *xs* = []
⟨*proof*⟩

lemma *perm-sym*: *xs* <~~> *ys* ==> *ys* <~~> *xs*
⟨*proof*⟩

53.2 Ways of making new permutations

We can insert the head anywhere in the list.

lemma *perm-append-Cons*: *a* # *xs* @ *ys* <~~> *xs* @ *a* # *ys*
⟨*proof*⟩

lemma *perm-append-swap*: *xs* @ *ys* <~~> *ys* @ *xs*
⟨*proof*⟩

lemma *perm-append-single*: *a* # *xs* <~~> *xs* @ [*a*]
⟨*proof*⟩

lemma *perm-rev*: *rev xs* <~~> *xs*
⟨*proof*⟩

lemma *perm-append1*: *xs* <~~> *ys* ==> *l* @ *xs* <~~> *l* @ *ys*
⟨*proof*⟩

lemma *perm-append2*: *xs* <~~> *ys* ==> *xs* @ *l* <~~> *ys* @ *l*
⟨*proof*⟩

53.3 Further results

lemma *perm-empty* [*iff*]: ([] <~~> *xs*) = (*xs* = [])
⟨*proof*⟩

lemma *perm-empty2* [*iff*]: (*xs* <~~> []) = (*xs* = [])
⟨*proof*⟩

lemma *perm-sing-imp*: *ys* <~~> *xs* ==> *xs* = [*y*] ==> *ys* = [*y*]
⟨*proof*⟩

lemma *perm-sing-eq* [*iff*]: (*ys* <~~> [*y*]) = (*ys* = [*y*])
⟨*proof*⟩

lemma *perm-sing-eq2* [*iff*]: ([*y*] <~~> *ys*) = (*ys* = [*y*])
⟨*proof*⟩

53.4 Removing elements

lemma *perm-remove*: $x \in set\ ys \Longrightarrow ys <^{\sim\sim}> x \# remove1\ x\ ys$
⟨*proof*⟩

Congruence rule

lemma *perm-remove-perm*: $xs <^{\sim\sim}> ys \Longrightarrow remove1\ z\ xs <^{\sim\sim}> remove1\ z\ ys$
⟨*proof*⟩

lemma *remove-hd* [*simp*]: $remove1\ z\ (z \# xs) = xs$
⟨*proof*⟩

lemma *cons-perm-imp-perm*: $z \# xs <^{\sim\sim}> z \# ys \Longrightarrow xs <^{\sim\sim}> ys$
⟨*proof*⟩

lemma *cons-perm-eq* [*iff*]: $(z\#xs <^{\sim\sim}> z\#ys) = (xs <^{\sim\sim}> ys)$
⟨*proof*⟩

lemma *append-perm-imp-perm*: $zs @ xs <^{\sim\sim}> zs @ ys \Longrightarrow xs <^{\sim\sim}> ys$
⟨*proof*⟩

lemma *perm-append1-eq* [*iff*]: $(zs @ xs <^{\sim\sim}> zs @ ys) = (xs <^{\sim\sim}> ys)$
⟨*proof*⟩

lemma *perm-append2-eq* [*iff*]: $(xs @ zs <^{\sim\sim}> ys @ zs) = (xs <^{\sim\sim}> ys)$
⟨*proof*⟩

lemma *multiset-of-eq-perm*: $(multiset\text{-}of\ xs = multiset\text{-}of\ ys) = (xs <^{\sim\sim}> ys)$
⟨*proof*⟩

lemma *multiset-of-le-perm-append*:
$multiset\text{-}of\ xs \leq multiset\text{-}of\ ys \longleftrightarrow (\exists zs.\ xs @ zs <^{\sim\sim}> ys)$
⟨*proof*⟩

lemma *perm-set-eq*: $xs <^{\sim\sim}> ys \Longrightarrow set\ xs = set\ ys$
⟨*proof*⟩

lemma *perm-distinct-iff*: $xs <^{\sim\sim}> ys \Longrightarrow distinct\ xs = distinct\ ys$
⟨*proof*⟩

lemma *eq-set-perm-remdups*: $set\ xs = set\ ys \Longrightarrow remdups\ xs <^{\sim\sim}> remdups\ ys$
⟨*proof*⟩

lemma *perm-remdups-iff-eq-set*: $remdups\ x <^{\sim\sim}> remdups\ y = (set\ x = set\ y)$
⟨*proof*⟩

end

54 Poly-Deriv: Polynomials and Differentiation

theory *Poly-Deriv*
imports *Deriv Polynomial*
begin

54.1 Derivatives of univariate polynomials

definition
pderiv :: *'a::real-normed-field poly* ⇒ *'a poly* **where**
pderiv = *poly-rec 0* (λ*a p p'*. *p* + *pCons 0 p'*)

lemma *pderiv-0* [*simp*]: *pderiv 0* = *0*
⟨*proof*⟩

lemma *pderiv-pCons*: *pderiv* (*pCons a p*) = *p* + *pCons 0* (*pderiv p*)
⟨*proof*⟩

lemma *coeff-pderiv*: *coeff* (*pderiv p*) *n* = *of-nat* (*Suc n*) ∗ *coeff p* (*Suc n*)
⟨*proof*⟩

lemma *pderiv-eq-0-iff*: *pderiv p* = *0* ⟷ *degree p* = *0*
⟨*proof*⟩

lemma *degree-pderiv*: *degree* (*pderiv p*) = *degree p* − *1*
⟨*proof*⟩

lemma *pderiv-singleton* [*simp*]: *pderiv* [:*a*:] = *0*
⟨*proof*⟩

lemma *pderiv-add*: *pderiv* (*p* + *q*) = *pderiv p* + *pderiv q*
⟨*proof*⟩

lemma *pderiv-minus*: *pderiv* (− *p*) = − *pderiv p*
⟨*proof*⟩

lemma *pderiv-diff*: *pderiv* (*p* − *q*) = *pderiv p* − *pderiv q*
⟨*proof*⟩

lemma *pderiv-smult*: *pderiv* (*smult a p*) = *smult a* (*pderiv p*)
⟨*proof*⟩

lemma *pderiv-mult*: *pderiv* (*p* ∗ *q*) = *p* ∗ *pderiv q* + *q* ∗ *pderiv p*
⟨*proof*⟩

lemma *pderiv-power-Suc*:
pderiv (*p* ^ *Suc n*) = *smult* (*of-nat* (*Suc n*)) (*p* ^ *n*) ∗ *pderiv p*
⟨*proof*⟩

lemma *DERIV-cmult2*: *DERIV f x* :> *D* ==> *DERIV* (%*x*. (*f x*) ∗ *c* :: *real*) *x*

$:> D * c$
⟨*proof*⟩

lemma *DERIV-pow2*: *DERIV* (%*x*. *x* ^ *Suc n*) *x* :> *real* (*Suc n*) * (*x* ^ *n*)
⟨*proof*⟩
declare *DERIV-pow2* [*simp*] *DERIV-pow* [*simp*]

lemma *DERIV-add-const*: *DERIV f x* :> *D* ==> *DERIV* (%*x*. *a* + *f x* :: 'a::*real-normed-field*) *x* :> *D*
⟨*proof*⟩

lemma *poly-DERIV*[*simp*]: *DERIV* (%*x*. *poly p x*) *x* :> *poly* (*pderiv p*) *x*
⟨*proof*⟩

Consequences of the derivative theorem above

lemma *poly-differentiable*[*simp*]: (%*x*. *poly p x*) *differentiable* (*x*::*real*)
⟨*proof*⟩

lemma *poly-isCont*[*simp*]: *isCont* (%*x*. *poly p x*) (*x*::*real*)
⟨*proof*⟩

lemma *poly-IVT-pos*: [| *a* < *b*; *poly p* (*a*::*real*) < *0*; *0* < *poly p b* |]
==> ∃ *x*. *a* < *x* & *x* < *b* & (*poly p x* = *0*)
⟨*proof*⟩

lemma *poly-IVT-neg*: [| (*a*::*real*) < *b*; *0* < *poly p a*; *poly p b* < *0* |]
==> ∃ *x*. *a* < *x* & *x* < *b* & (*poly p x* = *0*)
⟨*proof*⟩

lemma *poly-MVT*: (*a*::*real*) < *b* ==>
∃ *x*. *a* < *x* & *x* < *b* & (*poly p b* − *poly p a* = (*b* − *a*) * *poly* (*pderiv p*) *x*)
⟨*proof*⟩

Lemmas for Derivatives

lemma *order-unique-lemma*:
fixes *p* :: 'a::*idom poly*
assumes [:−*a*, *1*:] ^ *n dvd p* ∧ ¬ [:−*a*, *1*:] ^ *Suc n dvd p*
shows *n* = *order a p*
⟨*proof*⟩

lemma *lemma-order-pderiv1*:
pderiv ([:− *a*, *1*:] ^ *Suc n* * *q*) = [:− *a*, *1*:] ^ *Suc n* * *pderiv q* +
smult (*of-nat* (*Suc n*)) (*q* * [:− *a*, *1*:] ^ *n*)
⟨*proof*⟩

lemma *dvd-add-cancel1*:
fixes *a b c* :: 'a::*comm-ring-1*
shows *a dvd b* + *c* ⟹ *a dvd b* ⟹ *a dvd c*
⟨*proof*⟩

lemma *lemma-order-pderiv* [*rule-format*]:
$\forall p\ q\ a.\ 0 < n\ \&$
pderiv $p \neq 0\ \&$
$p = [:-\ a,\ 1:]\ \hat{}\ n * q\ \&\ \sim [:-\ a,\ 1:]\ dvd\ q$
$--> n = Suc\ (order\ a\ (pderiv\ p))$
⟨*proof*⟩

lemma *order-decomp*:
$p \neq 0$
$==> \exists q.\ p = [:-a,\ 1:]\ \hat{}\ (order\ a\ p) * q\ \&$
$\sim([:-a,\ 1:]\ dvd\ q)$
⟨*proof*⟩

lemma *order-pderiv*: [| *pderiv* $p \neq 0$; *order* $a\ p \neq 0$ |]
$==> (order\ a\ p = Suc\ (order\ a\ (pderiv\ p)))$
⟨*proof*⟩

lemma *order-mult*: $p * q \neq 0 \Longrightarrow order\ a\ (p * q) = order\ a\ p + order\ a\ q$
⟨*proof*⟩

Now justify the standard squarefree decomposition, i.e. f / gcd(f,f').

lemma *order-divides*: $[:-a,\ 1:]\ \hat{}\ n\ dvd\ p \longleftrightarrow p = 0 \vee n \leq order\ a\ p$
⟨*proof*⟩

lemma *poly-squarefree-decomp-order*:
assumes *pderiv* $p \neq 0$
and p: $p = q * d$
and p': *pderiv* $p = e * d$
and d: $d = r * p + s * pderiv\ p$
shows *order* $a\ q = (if\ order\ a\ p = 0\ then\ 0\ else\ 1)$
⟨*proof*⟩

lemma *poly-squarefree-decomp-order2*: [| *pderiv* $p \neq 0$;
$p = q * d$;
pderiv $p = e * d$;
$d = r * p + s * pderiv\ p$
|] $==> \forall a.\ order\ a\ q = (if\ order\ a\ p = 0\ then\ 0\ else\ 1)$
⟨*proof*⟩

lemma *order-pderiv2*: [| *pderiv* $p \neq 0$; *order* $a\ p \neq 0$ |]
$==> (order\ a\ (pderiv\ p) = n) = (order\ a\ p = Suc\ n)$
⟨*proof*⟩

definition
rsquarefree :: *'a::idom poly* => *bool* **where**
rsquarefree $p = (p \neq 0\ \&\ (\forall a.\ (order\ a\ p = 0) \mid (order\ a\ p = 1)))$

lemma *pderiv-iszero*: *pderiv* $p = 0 \Longrightarrow \exists h.\ p = [:h:]$

⟨*proof*⟩

lemma *rsquarefree-roots*:
rsquarefree p = (∀ *a*. ~(*poly p a* = *0* & *poly* (*pderiv p*) *a* = *0*))
⟨*proof*⟩

lemma *poly-squarefree-decomp*:
assumes *pderiv p* ≠ *0*
and *p* = *q* ∗ *d*
and *pderiv p* = *e* ∗ *d*
and *d* = *r* ∗ *p* + *s* ∗ *pderiv p*
shows *rsquarefree q* & (∀ *a*. (*poly q a* = *0*) = (*poly p a* = *0*))
⟨*proof*⟩

end

55 Preorder: Preorders with explicit equivalence relation

theory *Preorder*
imports *Orderings*
begin

class *preorder-equiv* = *preorder*
begin

definition *equiv* :: *'a* ⇒ *'a* ⇒ *bool* **where**
equiv x y ⟷ *x* ≤ *y* ∧ *y* ≤ *x*

notation
equiv (*op* ~~) **and**
equiv ((-/ ~~ -) [*51*, *51*] *50*)

notation (*xsymbols*)
equiv (*op* ≈) **and**
equiv ((-/ ≈ -) [*51*, *51*] *50*)

notation (*HTML* **output**)
equiv (*op* ≈) **and**
equiv ((-/ ≈ -) [*51*, *51*] *50*)

lemma *refl* [*iff*]:
x ≈ *x*
⟨*proof*⟩

lemma *trans*:
x ≈ *y* ⟹ *y* ≈ *z* ⟹ *x* ≈ *z*

⟨*proof*⟩

lemma *antisym*:
$x \leq y \Longrightarrow y \leq x \Longrightarrow x \approx y$
⟨*proof*⟩

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

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

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

lemma *less-imp-not-eq*: $x < y \Longrightarrow x \approx y \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *less-imp-not-eq2*: $x < y \Longrightarrow y \approx x \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *neq-le-trans*: $\neg\, a \approx b \Longrightarrow a \leq b \Longrightarrow a < b$
⟨*proof*⟩

lemma *le-neq-trans*: $a \leq b \Longrightarrow \neg\, a \approx b \Longrightarrow a < b$
⟨*proof*⟩

lemma *antisym-conv*: $y \leq x \Longrightarrow x \leq y \longleftrightarrow x \approx y$
⟨*proof*⟩

end

end

56 Quicksort: Quicksort

theory *Quicksort*
imports *Main 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*]:
multiset-of (*quicksort xs*) = *multiset-of xs*
⟨*proof*⟩

lemma *set-quicksort* [*simp*]: *set* (*quicksort xs*) = *set xs*
⟨*proof*⟩

lemma *sorted-quicksort*: *sorted* (*quicksort xs*)
⟨*proof*⟩

theorem *quicksort-sort* [*code-unfold*]:
sort = *quicksort*
⟨*proof*⟩

end

end

57 Quotient-Syntax: Pretty syntax for Quotient operations

theory *Quotient-Syntax*
imports *Main*
begin

notation
rel-conj (**infixr** *OOO* 75) **and**
fun-map (**infixr** −−−> 55) **and**
fun-rel (**infixr** ===> 55)

end

58 Quotient-List: Quotient infrastructure for the list type

theory *Quotient-List*
imports *Main Quotient-Syntax*
begin

fun
list-rel

where
  list-rel R [] [] = True
| *list-rel R (x#xs) [] = False*
| *list-rel R [] (x#xs) = False*
| *list-rel R (x#xs) (y#ys) = (R x y ∧ list-rel R xs ys)*

declare [[*map list = (map, list-rel)*]]

lemma *split-list-all*:
  shows *(∀ x. P x) ⟷ P [] ∧ (∀ x xs. P (x#xs))*
  ⟨*proof*⟩

lemma *map-id*[*id-simps*]:
  shows *map id = id*
  ⟨*proof*⟩

lemma *list-rel-reflp*:
  shows *equivp R ⟹ list-rel R xs xs*
  ⟨*proof*⟩

lemma *list-rel-symp*:
  assumes *a*: *equivp R*
  shows *list-rel R xs ys ⟹ list-rel R ys xs*
  ⟨*proof*⟩

lemma *list-rel-transp*:
  assumes *a*: *equivp R*
  shows *list-rel R xs1 xs2 ⟹ list-rel R xs2 xs3 ⟹ list-rel R xs1 xs3*
  ⟨*proof*⟩

lemma *list-equivp*[*quot-equiv*]:
  assumes *a*: *equivp R*
  shows *equivp (list-rel R)*
  ⟨*proof*⟩

lemma *list-rel-rel*:
  assumes *q*: *Quotient R Abs Rep*
  shows *list-rel R r s = (list-rel R r r ∧ list-rel R s s ∧ (map Abs r = map Abs s))*
  ⟨*proof*⟩

lemma *list-quotient*[*quot-thm*]:
  assumes *q*: *Quotient R Abs Rep*
  shows *Quotient (list-rel R) (map Abs) (map Rep)*
  ⟨*proof*⟩

lemma *cons-prs-aux*:

assumes *q*: *Quotient R Abs Rep*
shows (*map Abs*) ((*Rep h*) # (*map Rep t*)) = *h* # *t*
⟨*proof*⟩

lemma *cons-prs*[*quot-preserve*]:
assumes *q*: *Quotient R Abs Rep*
shows (*Rep* −−−> (*map Rep*) −−−> (*map Abs*)) (*op #*) = (*op #*)
⟨*proof*⟩

lemma *cons-rsp*[*quot-respect*]:
assumes *q*: *Quotient R Abs Rep*
shows (*R* ===> *list-rel R* ===> *list-rel R*) (*op #*) (*op #*)
⟨*proof*⟩

lemma *nil-prs*[*quot-preserve*]:
assumes *q*: *Quotient R Abs Rep*
shows *map Abs* [] = []
⟨*proof*⟩

lemma *nil-rsp*[*quot-respect*]:
assumes *q*: *Quotient R Abs Rep*
shows *list-rel R* [] []
⟨*proof*⟩

lemma *map-prs-aux*:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows (*map abs2*) (*map* ((*abs1* −−−> *rep2*) *f*) (*map rep1 l*)) = *map f l*
⟨*proof*⟩

lemma *map-prs*[*quot-preserve*]:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows ((*abs1* −−−> *rep2*) −−−> (*map rep1*) −−−> (*map abs2*)) *map* = *map*
and ((*abs1* −−−> *id*) −−−> *map rep1* −−−> *id*) *map* = *map*
⟨*proof*⟩

lemma *map-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
shows ((*R1* ===> *R2*) ===> (*list-rel R1*) ===> *list-rel R2*) *map map*
and ((*R1* ===> *op* =) ===> (*list-rel R1*) ===> *op* =) *map map*
⟨*proof*⟩

lemma *foldr-prs-aux*:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows *abs2* (*foldr* ((*abs1* −−−> *abs2* −−−> *rep2*) *f*) (*map rep1 l*) (*rep2 e*))
= *foldr f l e*

⟨*proof*⟩

lemma *foldr-prs*[*quot-preserve*]:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows ((*abs1* −−−> *abs2* −−−> *rep2*) −−−> (*map rep1*) −−−> *rep2* −−−> *abs2*) *foldr* = *foldr*
⟨*proof*⟩

lemma *foldl-prs-aux*:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows *abs1* (*foldl* ((*abs1* −−−> *abs2* −−−> *rep1*) *f*) (*rep1 e*) (*map rep2 l*)) = *foldl f e l*
⟨*proof*⟩

lemma *foldl-prs*[*quot-preserve*]:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows ((*abs1* −−−> *abs2* −−−> *rep1*) −−−> *rep1* −−−> (*map rep2*) −−−> *abs1*) *foldl* = *foldl*
⟨*proof*⟩

lemma *list-rel-empty*:
shows *list-rel R* [] *b* ⟹ *length b* = *0*
⟨*proof*⟩

lemma *list-rel-len*:
shows *list-rel R a b* ⟹ *length a* = *length b*
⟨*proof*⟩

lemma *foldl-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
shows ((*R1* ===> *R2* ===> *R1*) ===> *R1* ===> *list-rel R2* ===> *R1*) *foldl foldl*
⟨*proof*⟩

lemma *foldr-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
shows ((*R1* ===> *R2* ===> *R2*) ===> *list-rel R1* ===> *R2* ===> *R2*) *foldr foldr*
⟨*proof*⟩

lemma *list-rel-rsp*:
assumes *r*: ∀ *x y*. *R x y* ⟶ (∀ *a b*. *R a b* ⟶ *S x a* = *T y b*)

and *l1*: *list-rel R x y*
and *l2*: *list-rel R a b*
shows *list-rel S x a = list-rel T y b*
⟨*proof*⟩

lemma[*quot-respect*]:
((*R* ===> *R* ===> *op* =) ===> *list-rel R* ===> *list-rel R* ===> *op* =) *list-rel list-rel*
⟨*proof*⟩

lemma[*quot-preserve*]:
assumes *a*: *Quotient R abs1 rep1*
shows ((*abs1* ---> *abs1* ---> *id*) ---> *map rep1* ---> *map rep1* ---> *id*) *list-rel* = *list-rel*
⟨*proof*⟩

lemma[*quot-preserve*]:
assumes *a*: *Quotient R abs1 rep1*
shows (*list-rel* ((*rep1* ---> *rep1* ---> *id*) *R*) *l m*) = (*l* = *m*)
⟨*proof*⟩

lemma *list-rel-eq*[*id-simps*]:
shows (*list-rel* (*op* =)) = (*op* =)
⟨*proof*⟩

lemma *list-rel-find-element*:
assumes *a*: *x* ∈ *set a*
and *b*: *list-rel R a b*
shows ∃ *y*. (*y* ∈ *set b* ∧ *R x y*)
⟨*proof*⟩

lemma *list-rel-refl*:
assumes *a*: ⋀*x y*. *R x y* = (*R x* = *R y*)
shows *list-rel R x x*
⟨*proof*⟩

end

59 Quotient-Option: Quotient infrastructure for the option type

theory *Quotient-Option*
imports *Main Quotient-Syntax*
begin

fun
option-rel

where
  option-rel R None None = True
| *option-rel R (Some x) None = False*
| *option-rel R None (Some x) = False*
| *option-rel R (Some x) (Some y) = R x y*

declare [[*map option* = (*Option.map*, *option-rel*)]]

should probably be in Option.thy

lemma *split-option-all*:
  shows (∀ *x*. *P x*) ⟷ *P None* ∧ (∀ *a*. *P* (*Some a*))
  ⟨*proof*⟩

lemma *option-quotient*[*quot-thm*]:
  assumes *q*: *Quotient R Abs Rep*
  shows *Quotient* (*option-rel R*) (*Option.map Abs*) (*Option.map Rep*)
  ⟨*proof*⟩

lemma *option-equivp*[*quot-equiv*]:
  assumes *a*: *equivp R*
  shows *equivp* (*option-rel R*)
  ⟨*proof*⟩

lemma *option-None-rsp*[*quot-respect*]:
  assumes *q*: *Quotient R Abs Rep*
  shows *option-rel R None None*
  ⟨*proof*⟩

lemma *option-Some-rsp*[*quot-respect*]:
  assumes *q*: *Quotient R Abs Rep*
  shows (*R* ===> *option-rel R*) *Some Some*
  ⟨*proof*⟩

lemma *option-None-prs*[*quot-preserve*]:
  assumes *q*: *Quotient R Abs Rep*
  shows *Option.map Abs None = None*
  ⟨*proof*⟩

lemma *option-Some-prs*[*quot-preserve*]:
  assumes *q*: *Quotient R Abs Rep*
  shows (*Rep* ---> *Option.map Abs*) *Some = Some*
  ⟨*proof*⟩

lemma *option-map-id*[*id-simps*]:
  shows *Option.map id = id*
  ⟨*proof*⟩

lemma *option-rel-eq*[*id-simps*]:
  shows *option-rel* (*op =*) = (*op =*)

⟨*proof*⟩

end

60 Quotient-Product: Quotient infrastructure for the product type

theory *Quotient-Product*
imports *Main Quotient-Syntax*
begin

fun
prod-rel
where
prod-rel R1 R2 = (λ(*a*, *b*) (*c*, *d*). *R1 a c* ∧ *R2 b d*)

declare [[*map* ∗ = (*prod-fun*, *prod-rel*)]]

lemma *prod-equivp*[*quot-equiv*]:
assumes *a*: *equivp R1*
assumes *b*: *equivp R2*
shows *equivp* (*prod-rel R1 R2*)
⟨*proof*⟩

lemma *prod-quotient*[*quot-thm*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows *Quotient* (*prod-rel R1 R2*) (*prod-fun Abs1 Abs2*) (*prod-fun Rep1 Rep2*)
⟨*proof*⟩

lemma *Pair-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*R1* ===> *R2* ===> *prod-rel R1 R2*) *Pair Pair*
⟨*proof*⟩

lemma *Pair-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*Rep1* −−−> *Rep2* −−−> (*prod-fun Abs1 Abs2*)) *Pair* = *Pair*
⟨*proof*⟩

lemma *fst-rsp*[*quot-respect*]:
assumes *Quotient R1 Abs1 Rep1*
assumes *Quotient R2 Abs2 Rep2*
shows (*prod-rel R1 R2* ===> *R1*) *fst fst*

⟨*proof*⟩

lemma *fst-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*prod-fun Rep1 Rep2 ---> Abs1*) *fst* = *fst*
⟨*proof*⟩

lemma *snd-rsp*[*quot-respect*]:
assumes *Quotient R1 Abs1 Rep1*
assumes *Quotient R2 Abs2 Rep2*
shows (*prod-rel R1 R2 ===> R2*) *snd snd*
⟨*proof*⟩

lemma *snd-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*prod-fun Rep1 Rep2 ---> Abs2*) *snd* = *snd*
⟨*proof*⟩

lemma *split-rsp*[*quot-respect*]:
shows ((*R1 ===> R2 ===> (op =)*) *===> (prod-rel R1 R2) ===> (op =)*) *split split*
⟨*proof*⟩

lemma *split-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
shows (((*Abs1 ---> Abs2 ---> id*) *---> prod-fun Rep1 Rep2 ---> id*) *split*) = *split*
⟨*proof*⟩

lemma [*quot-respect*]:
shows ((*R2 ===> R2 ===> op =*) *===> (R1 ===> R1 ===> op =) ===>*
prod-rel R2 R1 ===> prod-rel R2 R1 ===> op =) *prod-rel prod-rel*
⟨*proof*⟩

lemma [*quot-preserve*]:
assumes *q1*: *Quotient R1 abs1 rep1*
and *q2*: *Quotient R2 abs2 rep2*
shows ((*abs1 ---> abs1 ---> id*) *---> (abs2 ---> abs2 ---> id) ---> prod-fun rep1 rep2 ---> prod-fun rep1 rep2 ---> id*) *prod-rel* = *prod-rel*
⟨*proof*⟩

lemma [*quot-preserve*]:
shows(*prod-rel* ((*rep1 ---> rep1 ---> id*) *R1*) ((*rep2 ---> rep2 ---> id*) *R2*)

(*l1*, *l2*) (*r1*, *r2*)) = (*R1* (*rep1 l1*) (*rep1 r1*) ∧ *R2* (*rep2 l2*) (*rep2 r2*))
⟨*proof*⟩

declare *Pair-eq*[*quot-preserve*]

lemma *prod-fun-id*[*id-simps*]:
shows *prod-fun id id* = *id*
⟨*proof*⟩

lemma *prod-rel-eq*[*id-simps*]:
shows *prod-rel* (*op* =) (*op* =) = (*op* =)
⟨*proof*⟩

end

61 Quotient-Sum: Quotient infrastructure for the sum type

theory *Quotient-Sum*
imports *Main Quotient-Syntax*
begin

fun
sum-rel
where
sum-rel R1 R2 (*Inl a1*) (*Inl b1*) = *R1 a1 b1*
| *sum-rel R1 R2* (*Inl a1*) (*Inr b2*) = *False*
| *sum-rel R1 R2* (*Inr a2*) (*Inl b1*) = *False*
| *sum-rel R1 R2* (*Inr a2*) (*Inr b2*) = *R2 a2 b2*

fun
sum-map
where
sum-map f1 f2 (*Inl a*) = *Inl* (*f1 a*)
| *sum-map f1 f2* (*Inr a*) = *Inr* (*f2 a*)

declare [[*map* + = (*sum-map*, *sum-rel*)]]

should probably be in *Sum-Type*

lemma *split-sum-all*:
shows (∀ *x*. *P x*) ⟷ (∀ *x*. *P* (*Inl x*)) ∧ (∀ *x*. *P* (*Inr x*))
⟨*proof*⟩

lemma *sum-equivp*[*quot-equiv*]:
assumes *a*: *equivp R1*
assumes *b*: *equivp R2*
shows *equivp* (*sum-rel R1 R2*)

⟨*proof*⟩

lemma *sum-quotient*[*quot-thm*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows *Quotient* (*sum-rel R1 R2*) (*sum-map Abs1 Abs2*) (*sum-map Rep1 Rep2*)
⟨*proof*⟩

lemma *sum-Inl-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*R1* ===> *sum-rel R1 R2*) *Inl Inl*
⟨*proof*⟩

lemma *sum-Inr-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*R2* ===> *sum-rel R1 R2*) *Inr Inr*
⟨*proof*⟩

lemma *sum-Inl-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*Rep1* ---> *sum-map Abs1 Abs2*) *Inl* = *Inl*
⟨*proof*⟩

lemma *sum-Inr-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*Rep2* ---> *sum-map Abs1 Abs2*) *Inr* = *Inr*
⟨*proof*⟩

lemma *sum-map-id*[*id-simps*]:
shows *sum-map id id* = *id*
⟨*proof*⟩

lemma *sum-rel-eq*[*id-simps*]:
shows *sum-rel* (*op* =) (*op* =) = (*op* =)
⟨*proof*⟩

end

62 Quotient-Type: Quotient types

theory *Quotient-Type*
imports *Main*
begin

We introduce the notion of quotient types over equivalence relations via type classes.

62.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim :: 'a => 'a => bool$.

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

class *equiv* = *eqv* +
  assumes *equiv-refl* [*intro*]: $x \sim x$
  assumes *equiv-trans* [*trans*]: $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$
  assumes *equiv-sym* [*sym*]: $x \sim y \Longrightarrow y \sim x$

lemma *equiv-not-sym* [*sym*]: $\neg (x \sim y) ==> \neg (y \sim (x::'a::equiv))$
⟨*proof*⟩

lemma *not-equiv-trans1* [*trans*]: $\neg (x \sim y) ==> y \sim z ==> \neg (x \sim (z::'a::equiv))$
⟨*proof*⟩

lemma *not-equiv-trans2* [*trans*]: $x \sim y ==> \neg (y \sim z) ==> \neg (x \sim (z::'a::equiv))$
⟨*proof*⟩

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

typedef *'a quot* = $\{\{x.\ a \sim x\} \mid a::'a::eqv.\ True\}$
  ⟨*proof*⟩

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

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

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

definition
  class :: *'a::equiv => 'a quot* (⌊-⌋) **where**
  $\lfloor a \rfloor$ = *Abs-quot* $\{x.\ a \sim x\}$

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

lemma *quot-cases* [*cases type*: *quot*]: $(!!a.\ A = \lfloor a \rfloor ==> C) ==> C$
  ⟨*proof*⟩

62.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $(\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim b)$
⟨*proof*⟩

62.3 Picking representing elements

definition
pick :: *'a::equiv quot* => *'a* **where**
pick A = (*SOME a*. *A* = $\lfloor a \rfloor$)

theorem *pick-equiv* [*intro*]: *pick* $\lfloor a \rfloor \sim a$
⟨*proof*⟩

theorem *pick-inverse* [*intro*]: $\lfloor$*pick A*$\rfloor$ = *A*
⟨*proof*⟩

The following rules support canonical function definitions on quotient types (with up to two arguments). Note that the stripped-down version without additional conditions is sufficient most of the time.

theorem *quot-cond-function*:
assumes *eq*: !!*X Y*. *P X Y* ==> *f X Y* == *g* (*pick X*) (*pick Y*)
and *cong*: !!*x x' y y'*. $\lfloor x \rfloor = \lfloor x' \rfloor$ ==> $\lfloor y \rfloor = \lfloor y' \rfloor$
==> *P* $\lfloor x \rfloor$ $\lfloor y \rfloor$ ==> *P* $\lfloor x' \rfloor$ $\lfloor y' \rfloor$ ==> *g x y* = *g x' y'*
and *P*: *P* $\lfloor a \rfloor$ $\lfloor b \rfloor$
shows *f* $\lfloor a \rfloor$ $\lfloor b \rfloor$ = *g a b*
⟨*proof*⟩

theorem *quot-function*:
assumes !!*X Y*. *f X Y* == *g* (*pick X*) (*pick Y*)
and !!*x x' y y'*. $\lfloor x \rfloor = \lfloor x' \rfloor$ ==> $\lfloor y \rfloor = \lfloor y' \rfloor$ ==> *g x y* = *g x' y'*
shows *f* $\lfloor a \rfloor$ $\lfloor b \rfloor$ = *g a b*
⟨*proof*⟩

theorem *quot-function'*:
(!!*X Y*. *f X Y* == *g* (*pick X*) (*pick Y*)) ==>
(!!*x x' y y'*. *x* ∼ *x'* ==> *y* ∼ *y'* ==> *g x y* = *g x' y'*) ==>
f $\lfloor a \rfloor$ $\lfloor b \rfloor$ = *g a b*
⟨*proof*⟩

end

63 Ramsey: Ramsey's Theorem

theory *Ramsey*
imports *Main Infinite-Set*
begin

63.1 Preliminaries

63.1.1 "Axiom" of Dependent Choice

primrec *choice* :: $('a \Rightarrow bool) \Rightarrow ('a * 'a)\ set \Rightarrow nat \Rightarrow 'a$ **where**
— An integer-indexed chain of choices
choice-0: $choice\ P\ r\ 0 = (SOME\ x.\ P\ x)$
| *choice-Suc*: $choice\ P\ r\ (Suc\ n) = (SOME\ y.\ P\ y\ \&\ (choice\ P\ r\ n,\ y) \in r)$

lemma *choice-n*:
assumes *P0*: $P\ x0$
and *Pstep*: $!!x.\ P\ x \Longrightarrow \exists y.\ P\ y\ \&\ (x,y) \in r$
shows $P\ (choice\ P\ r\ n)$
⟨*proof*⟩

lemma *dependent-choice*:
assumes *trans*: *trans r*
and *P0*: *P x0*
and *Pstep*: $!!x.\ P\ x \Longrightarrow \exists y.\ P\ y\ \&\ (x,y) \in r$
obtains *f* :: *nat* => *'a* **where**
$!!n.\ P\ (f\ n)$ **and** $!!n\ m.\ n < m \Longrightarrow (f\ n, f\ m) \in r$
⟨*proof*⟩

63.1.2 Partitions of a Set

definition
part :: *nat* => *nat* => *'a set* => (*'a set* => *nat*) => *bool*
— the function *f* partitions the *r*-subsets of the typically infinite set *Y* into *s* distinct categories.
where
$part\ r\ s\ Y\ f = (\forall X.\ X \subseteq Y\ \&\ finite\ X\ \&\ card\ X = r \longrightarrow f\ X < s)$

For induction, we decrease the value of *r* in partitions.

lemma *part-Suc-imp-part*:
$[|\ infinite\ Y;\ part\ (Suc\ r)\ s\ Y\ f;\ y \in Y\ |]$
$\Longrightarrow part\ r\ s\ (Y - \{y\})\ (\%u.\ f\ (insert\ y\ u))$
⟨*proof*⟩

lemma *part-subset*: $part\ r\ s\ YY\ f \Longrightarrow Y \subseteq YY \Longrightarrow part\ r\ s\ Y\ f$
⟨*proof*⟩

63.2 Ramsey's Theorem: Infinitary Version

lemma *Ramsey-induction*:
fixes *s* **and** *r*::*nat*
shows
$!!(YY::'a\ set)\ (f::'a\ set \Rightarrow nat).$
$[|infinite\ YY;\ part\ r\ s\ YY\ f|]$
$\Longrightarrow \exists Y'\ t'.\ Y' \subseteq YY\ \&\ infinite\ Y'\ \&\ t' < s\ \&$
$(\forall X.\ X \subseteq Y'\ \&\ finite\ X\ \&\ card\ X = r \longrightarrow f\ X = t')$

⟨*proof*⟩

theorem *Ramsey*:
fixes $s\ r :: nat$ **and** $Z::'a\ set$ **and** $f::'a\ set => nat$
shows
$[|infinite\ Z;$
$\forall X.\ X \subseteq Z\ \&\ finite\ X\ \&\ card\ X = r\ --> f\ X < s|]$
$==> \exists Y\ t.\ Y \subseteq Z\ \&\ infinite\ Y\ \&\ t < s$
$\&\ (\forall X.\ X \subseteq Y\ \&\ finite\ X\ \&\ card\ X = r\ --> f\ X = t)$
⟨*proof*⟩

corollary *Ramsey2*:
fixes $s::nat$ **and** $Z::'a\ set$ **and** $f::'a\ set => nat$
assumes *infZ*: *infinite Z*
and *part*: $\forall x{\in}Z.\ \forall y{\in}Z.\ x{\neq}y\ --> f\{x,y\} < s$
shows
$\exists Y\ t.\ Y \subseteq Z\ \&\ infinite\ Y\ \&\ t < s\ \&\ (\forall x{\in}Y.\ \forall y{\in}Y.\ x{\neq}y\ --> f\{x,y\} = t)$
⟨*proof*⟩

63.3 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [3].

definition
disj-wf :: $('a * 'a)set => bool$
where
disj-wf r = $(\exists T.\ \exists n::nat.\ (\forall i{<}n.\ wf(T\ i))\ \&\ r = (\bigcup i{<}n.\ T\ i))$

definition
transition-idx :: $[nat => 'a,\ nat => ('a{*}'a)set,\ nat\ set] => nat$
where
transition-idx s T A =
$(LEAST\ k.\ \exists i\ j.\ A = \{i,j\}\ \&\ i{<}j\ \&\ (s\ j,\ s\ i) \in T\ k)$

lemma *transition-idx-less*:
$[|i{<}j;\ (s\ j,\ s\ i) \in T\ k;\ k{<}n|] ==>$ *transition-idx s T* $\{i,j\} < n$
⟨*proof*⟩

lemma *transition-idx-in*:
$[|i{<}j;\ (s\ j,\ s\ i) \in T\ k|] ==> (s\ j,\ s\ i) \in T$ (*transition-idx s T* $\{i,j\}$)
⟨*proof*⟩

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

lemma *disj-wf*:
disj-wf$(r) = (\exists T.\ \exists n::nat.\ (\forall i{<}n.\ wf(T\ i))\ \&\ r \subseteq (\bigcup i{<}n.\ T\ i))$
⟨*proof*⟩

theorem *trans-disj-wf-implies-wf*:
assumes *transr*: *trans r*
and *dwf*: *disj-wf*(*r*)
shows *wf r*
⟨*proof*⟩

end

64 Reflection: Generic reflection and reification

theory *Reflection*
imports *Main*
uses *reify-data.ML* (*reflection.ML*)
begin

⟨*ML*⟩

lemma *ext2*: $(\forall x.\ f\ x = g\ x) \Longrightarrow f = g$
⟨*proof*⟩

⟨*ML*⟩

end

65 RBT-Impl: Implementation of Red-Black Trees

theory *RBT-Impl*
imports *Main*
begin

For applications, you should use theory *RBT* which defines an abstract type of red-black tree obeying the invariant.

65.1 Datatype of RB trees

datatype *color* = *R* | *B*
datatype ('*a*, '*b*) *rbt* = *Empty* | *Branch color* ('*a*, '*b*) *rbt* '*a* '*b* ('*a*, '*b*) *rbt*

lemma *rbt-cases*:
obtains (*Empty*) *t* = *Empty*
| (*Red*) *l k v r* **where** *t* = *Branch R l k v r*
| (*Black*) *l k v r* **where** *t* = *Branch B l k v r*
⟨*proof*⟩

65.2 Tree properties

65.2.1 Content of a tree

primrec *entries* :: $('a, 'b)$ *rbt* $\Rightarrow$ $('a \times 'b)$ *list*
where
entries Empty = []
| *entries* (*Branch* - *l k v r*) = *entries l* @ (*k*,*v*) # *entries r*

abbreviation (*input*) *entry-in-tree* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *rbt* $\Rightarrow$ *bool*
where
entry-in-tree k v t $\equiv$ $(k, v) \in$ *set* (*entries t*)

definition *keys* :: $('a, 'b)$ *rbt* $\Rightarrow$ $'a$ *list* **where**
keys t = *map fst* (*entries t*)

lemma *keys-simps* [*simp*, *code*]:
keys Empty = []
keys (*Branch c l k v r*) = *keys l* @ *k* # *keys r*
⟨*proof*⟩

lemma *entry-in-tree-keys*:
assumes $(k, v) \in$ *set* (*entries t*)
shows $k \in$ *set* (*keys t*)
⟨*proof*⟩

lemma *keys-entries*:
$k \in$ *set* (*keys t*) $\longleftrightarrow$ $(\exists\, v.\ (k, v) \in$ *set* (*entries t*))
⟨*proof*⟩

65.2.2 Search tree properties

definition *tree-less* :: $'a$::*order* $\Rightarrow$ $('a, 'b)$ *rbt* $\Rightarrow$ *bool*
where
tree-less-prop: *tree-less k t* $\longleftrightarrow$ $(\forall\, x{\in}$*set* (*keys t*)$.\ x < k)$

abbreviation *tree-less-symbol* (**infix** $|\ll$ *50*)
where *t* $|\ll$ *x* $\equiv$ *tree-less x t*

definition *tree-greater* :: $'a$::*order* $\Rightarrow$ $('a, 'b)$ *rbt* $\Rightarrow$ *bool* (**infix** $\ll|$ *50*)
where
tree-greater-prop: *tree-greater k t* = $(\forall\, x{\in}$*set* (*keys t*)$.\ k < x)$

lemma *tree-less-simps* [*simp*]:
tree-less k Empty = *True*
tree-less k (*Branch c lt kt v rt*) $\longleftrightarrow$ *kt* $<$ *k* $\wedge$ *tree-less k lt* $\wedge$ *tree-less k rt*
⟨*proof*⟩

lemma *tree-greater-simps* [*simp*]:
tree-greater k Empty = *True*

tree-greater k (Branch c lt kt v rt) ⟷ k < kt ∧ tree-greater k lt ∧ tree-greater k rt
⟨*proof*⟩

lemmas *tree-ord-props = tree-less-prop tree-greater-prop*

lemmas *tree-greater-nit = tree-greater-prop entry-in-tree-keys*
lemmas *tree-less-nit = tree-less-prop entry-in-tree-keys*

lemma *tree-less-eq-trans*: *l |≪ u ⟹ u ≤ v ⟹ l |≪ v*
and *tree-less-trans*: *t |≪ x ⟹ x < y ⟹ t |≪ y*
and *tree-greater-eq-trans*: *u ≤ v ⟹ v ≪| r ⟹ u ≪| r*
and *tree-greater-trans*: *x < y ⟹ y ≪| t ⟹ x ≪| t*
⟨*proof*⟩

primrec *sorted* :: (*'a*::*linorder*, *'b*) *rbt* ⇒ *bool*
where
sorted Empty = True
| *sorted (Branch c l k v r) = (l |≪ k ∧ k ≪| r ∧ sorted l ∧ sorted r)*

lemma *sorted-entries*:
sorted t ⟹ List.sorted (List.map fst (entries t))
⟨*proof*⟩

lemma *distinct-entries*:
sorted t ⟹ distinct (List.map fst (entries t))
⟨*proof*⟩

65.2.3 Tree lookup

primrec *lookup* :: (*'a*::*linorder*, *'b*) *rbt* ⇒ *'a* ⇀ *'b*
where
lookup Empty k = None
| *lookup (Branch - l x y r) k = (if k < x then lookup l k else if x < k then lookup r k else Some y)*

lemma *lookup-keys*: *sorted t ⟹ dom (lookup t) = set (keys t)*
⟨*proof*⟩

lemma *dom-lookup-Branch*:
sorted (Branch c t1 k v t2) ⟹
dom (lookup (Branch c t1 k v t2))
= Set.insert k (dom (lookup t1) ∪ dom (lookup t2))
⟨*proof*⟩

lemma *finite-dom-lookup* [*simp*, *intro*!]: *finite (dom (lookup t))*
⟨*proof*⟩

lemma *lookup-tree-less*[*simp*]: *t |≪ k ⟹ lookup t k = None*

⟨*proof*⟩

lemma *lookup-tree-greater*[*simp*]: $k \ll\vert\ t \Longrightarrow lookup\ t\ k = None$
⟨*proof*⟩

lemma *lookup-Empty*: $lookup\ Empty = empty$
⟨*proof*⟩

lemma *map-of-entries*:
$sorted\ t \Longrightarrow map\text{-}of\ (entries\ t) = lookup\ t$
⟨*proof*⟩

lemma *lookup-in-tree*: $sorted\ t \Longrightarrow lookup\ t\ k = Some\ v \longleftrightarrow (k, v) \in set\ (entries\ t)$
⟨*proof*⟩

lemma *set-entries-inject*:
assumes *sorted*: *sorted t1 sorted t2*
shows $set\ (entries\ t1) = set\ (entries\ t2) \longleftrightarrow entries\ t1 = entries\ t2$
⟨*proof*⟩

lemma *entries-eqI*:
assumes *sorted*: *sorted t1 sorted t2*
assumes *lookup*: $lookup\ t1 = lookup\ t2$
shows $entries\ t1 = entries\ t2$
⟨*proof*⟩

lemma *entries-lookup*:
assumes *sorted t1 sorted t2*
shows $entries\ t1 = entries\ t2 \longleftrightarrow lookup\ t1 = lookup\ t2$
⟨*proof*⟩

lemma *lookup-from-in-tree*:
assumes *sorted t1 sorted t2*
and $\bigwedge v.\ (k::'a::linorder, v) \in set\ (entries\ t1) \longleftrightarrow (k, v) \in set\ (entries\ t2)$
shows $lookup\ t1\ k = lookup\ t2\ k$
⟨*proof*⟩

65.2.4 Red-black properties

primrec $color\text{-}of :: ('a, 'b)\ rbt \Rightarrow color$
where
$color\text{-}of\ Empty = B$
| $color\text{-}of\ (Branch\ c\ \text{-}\ \text{-}\ \text{-}\ \text{-}) = c$

primrec $bheight :: ('a,'b)\ rbt \Rightarrow nat$
where
$bheight\ Empty = 0$
| $bheight\ (Branch\ c\ lt\ k\ v\ rt) = (if\ c = B\ then\ Suc\ (bheight\ lt)\ else\ bheight\ lt)$

primrec *inv1* :: $('a, 'b)$ *rbt* $\Rightarrow$ *bool*
where
inv1 Empty = *True*
| *inv1* (*Branch c lt k v rt*) $\longleftrightarrow$ *inv1 lt* $\wedge$ *inv1 rt* $\wedge$ ($c = B \vee$ *color-of lt* $= B \wedge$ *color-of rt* $= B$)

primrec *inv1l* :: $('a, 'b)$ *rbt* $\Rightarrow$ *bool* — Weaker version
where
inv1l Empty = *True*
| *inv1l* (*Branch c l k v r*) = (*inv1 l* $\wedge$ *inv1 r*)
lemma [*simp*]: *inv1 t* $\Longrightarrow$ *inv1l t* ⟨*proof*⟩

primrec *inv2* :: $('a, 'b)$ *rbt* $\Rightarrow$ *bool*
where
inv2 Empty = *True*
| *inv2* (*Branch c lt k v rt*) = (*inv2 lt* $\wedge$ *inv2 rt* $\wedge$ *bheight lt* = *bheight rt*)

definition *is-rbt* :: $('a$::*linorder*, $'b)$ *rbt* $\Rightarrow$ *bool* **where**
is-rbt t $\longleftrightarrow$ *inv1 t* $\wedge$ *inv2 t* $\wedge$ *color-of t* $= B \wedge$ *sorted t*

lemma *is-rbt-sorted* [*simp*]:
is-rbt t $\Longrightarrow$ *sorted t* ⟨*proof*⟩

theorem *Empty-is-rbt* [*simp*]:
is-rbt Empty ⟨*proof*⟩

65.3 Insertion

fun
balance :: $('a,'b)$ *rbt* $\Rightarrow$ $'a \Rightarrow$ $'b \Rightarrow$ $('a,'b)$ *rbt* $\Rightarrow$ $('a,'b)$ *rbt*
where
balance (*Branch R a w x b*) *s t* (*Branch R c y z d*) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance (*Branch R* (*Branch R a w x b*) *s t c*) *y z d* = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance (*Branch R a w x* (*Branch R b s t c*)) *y z d* = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a w x (*Branch R b s t* (*Branch R c y z d*)) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a w x (*Branch R* (*Branch R b s t c*) *y z d*) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a s t b = *Branch B a s t b*

lemma *balance-inv1*: ⟦*inv1l l*; *inv1l r*⟧ $\Longrightarrow$ *inv1* (*balance l k v r*)
⟨*proof*⟩

lemma *balance-bheight*: *bheight l* = *bheight r* $\Longrightarrow$ *bheight* (*balance l k v r*) = *Suc* (*bheight l*)

⟨*proof*⟩

lemma *balance-inv2*:
assumes *inv2 l inv2 r bheight l = bheight r*
shows *inv2 (balance l k v r)*
⟨*proof*⟩

lemma *balance-tree-greater*[*simp*]: $(v \ll| \ balance\ a\ k\ x\ b) = (v \ll| \ a \wedge v \ll| \ b \wedge v < k)$
⟨*proof*⟩

lemma *balance-tree-less*[*simp*]: $(balance\ a\ k\ x\ b \ |\ll v) = (a \ |\ll v \wedge b \ |\ll v \wedge k < v)$
⟨*proof*⟩

lemma *balance-sorted*:
fixes $k :: 'a$::*linorder*
assumes *sorted l sorted r l* $|\ll$ *k k* $\ll|$ *r*
shows *sorted (balance l k v r)*
⟨*proof*⟩

lemma *entries-balance* [*simp*]:
entries (balance l k v r) = entries l @ (k, v) # entries r
⟨*proof*⟩

lemma *keys-balance* [*simp*]:
keys (balance l k v r) = keys l @ k # keys r
⟨*proof*⟩

lemma *balance-in-tree*:
entry-in-tree k x (balance l v y r) $\longleftrightarrow$ *entry-in-tree k x l* $\vee$ *k = v* $\wedge$ *x = y* $\vee$ *entry-in-tree k x r*
⟨*proof*⟩

lemma *lookup-balance*[*simp*]:
fixes $k :: 'a$::*linorder*
assumes *sorted l sorted r l* $|\ll$ *k k* $\ll|$ *r*
shows *lookup (balance l k v r) x = lookup (Branch B l k v r) x*
⟨*proof*⟩

primrec *paint* :: *color* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
paint c Empty = Empty
| *paint c (Branch - l k v r) = Branch c l k v r*

lemma *paint-inv1l*[*simp*]: *inv1l t* $\Longrightarrow$ *inv1l (paint c t)* ⟨*proof*⟩
lemma *paint-inv1*[*simp*]: *inv1l t* $\Longrightarrow$ *inv1 (paint B t)* ⟨*proof*⟩
lemma *paint-inv2*[*simp*]: *inv2 t* $\Longrightarrow$ *inv2 (paint c t)* ⟨*proof*⟩
lemma *paint-color-of*[*simp*]: *color-of (paint B t) = B* ⟨*proof*⟩

lemma *paint-sorted*[*simp*]: *sorted t* ⟹ *sorted* (*paint c t*) ⟨*proof*⟩
lemma *paint-in-tree*[*simp*]: *entry-in-tree k x* (*paint c t*) = *entry-in-tree k x t* ⟨*proof*⟩
lemma *paint-lookup*[*simp*]: *lookup* (*paint c t*) = *lookup t* ⟨*proof*⟩
lemma *paint-tree-greater*[*simp*]: (*v* ≪| *paint c t*) = (*v* ≪| *t*) ⟨*proof*⟩
lemma *paint-tree-less*[*simp*]: (*paint c t* |≪ *v*) = (*t* |≪ *v*) ⟨*proof*⟩

fun
ins :: ('*a*::*linorder* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
ins f k v Empty = *Branch R Empty k v Empty* |
ins f k v (*Branch B l x y r*) = (*if k* < *x then balance* (*ins f k v l*) *x y r*
else if k > *x then balance l x y* (*ins f k v r*)
else Branch B l x (*f k y v*) *r*) |
ins f k v (*Branch R l x y r*) = (*if k* < *x then Branch R* (*ins f k v l*) *x y r*
else if k > *x then Branch R l x y* (*ins f k v r*)
else Branch R l x (*f k y v*) *r*)

lemma *ins-inv1-inv2*:
assumes *inv1 t inv2 t*
shows *inv2* (*ins f k x t*) *bheight* (*ins f k x t*) = *bheight t*
color-of t = *B* ⟹ *inv1* (*ins f k x t*) *inv1l* (*ins f k x t*)
⟨*proof*⟩

lemma *ins-tree-greater*[*simp*]: (*v* ≪| *ins f k x t*) = (*v* ≪| *t* ∧ *k* > *v*)
⟨*proof*⟩
lemma *ins-tree-less*[*simp*]: (*ins f k x t* |≪ *v*) = (*t* |≪ *v* ∧ *k* < *v*)
⟨*proof*⟩
lemma *ins-sorted*[*simp*]: *sorted t* ⟹ *sorted* (*ins f k x t*)
⟨*proof*⟩

lemma *keys-ins*: *set* (*keys* (*ins f k v t*)) = { *k* } ∪ *set* (*keys t*)
⟨*proof*⟩

lemma *lookup-ins*:
fixes *k* :: '*a*::*linorder*
assumes *sorted t*
shows *lookup* (*ins f k v t*) *x* = ((*lookup t*)(*k* |–> *case lookup t k of None* ⇒ *v*
| *Some w* ⇒ *f k w v*)) *x*
⟨*proof*⟩

definition
insert-with-key :: ('*a*::*linorder* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒
('*a*,'*b*) *rbt*
where
insert-with-key f k v t = *paint B* (*ins f k v t*)

lemma *insertwk-sorted*: *sorted t* ⟹ *sorted* (*insert-with-key f k x t*)
⟨*proof*⟩

theorem *insertwk-is-rbt*:
assumes *inv*: *is-rbt t*
shows *is-rbt* (*insert-with-key f k x t*)
⟨*proof*⟩

lemma *lookup-insertwk*:
assumes *sorted t*
shows *lookup* (*insert-with-key f k v t*) *x* = ((*lookup t*)(*k* |−> *case lookup t k of None* ⇒ *v*
| *Some w* ⇒ *f k w v*)) *x*
⟨*proof*⟩

definition
insertw-def: *insert-with f* = *insert-with-key* (λ-. *f*)

lemma *insertw-sorted*: *sorted t* ⟹ *sorted* (*insert-with f k v t*) ⟨*proof*⟩
theorem *insertw-is-rbt*: *is-rbt t* ⟹ *is-rbt* (*insert-with f k v t*) ⟨*proof*⟩

lemma *lookup-insertw*:
assumes *is-rbt t*
shows *lookup* (*insert-with f k v t*) = (*lookup t*)(*k* ↦ (*if k*:*dom* (*lookup t*) *then f* (*the* (*lookup t k*)) *v else v*))
⟨*proof*⟩

definition *insert* :: *'a*::*linorder* ⇒ *'b* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where**
insert = *insert-with-key* (λ- - *nv*. *nv*)

lemma *insert-sorted*: *sorted t* ⟹ *sorted* (*insert k v t*) ⟨*proof*⟩
theorem *insert-is-rbt* [*simp*]: *is-rbt t* ⟹ *is-rbt* (*insert k v t*) ⟨*proof*⟩

lemma *lookup-insert*:
assumes *is-rbt t*
shows *lookup* (*insert k v t*) = (*lookup t*)(*k*↦*v*)
⟨*proof*⟩

65.4 Deletion

lemma *bheight-paintR'*[*simp*]: *color-of t* = *B* ⟹ *bheight* (*paint R t*) = *bheight t* − *1*
⟨*proof*⟩

fun
balance-left :: (*'a*,*'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
balance-left (*Branch R a k x b*) *s y c* = *Branch R* (*Branch B a k x b*) *s y c* |
balance-left bl k x (*Branch B a s y b*) = *balance bl k x* (*Branch R a s y b*) |
balance-left bl k x (*Branch R* (*Branch B a s y b*) *t z c*) = *Branch R* (*Branch B bl k x a*) *s y* (*balance b t z* (*paint R c*)) |
balance-left t k x s = *Empty*

lemma *balance-left-inv2-with-inv1*:
assumes *inv2 lt inv2 rt bheight lt + 1 = bheight rt inv1 rt*
shows *bheight (balance-left lt k v rt) = bheight lt + 1*
and *inv2 (balance-left lt k v rt)*
⟨*proof*⟩

lemma *balance-left-inv2-app*:
assumes *inv2 lt inv2 rt bheight lt + 1 = bheight rt color-of rt = B*
shows *inv2 (balance-left lt k v rt)*
bheight (balance-left lt k v rt) = bheight rt
⟨*proof*⟩

lemma *balance-left-inv1*: ⟦*inv1l a*; *inv1 b*; *color-of b = B*⟧ ⟹ *inv1 (balance-left a k x b)*
⟨*proof*⟩

lemma *balance-left-inv1l*: ⟦ *inv1l lt*; *inv1 rt* ⟧ ⟹ *inv1l (balance-left lt k x rt)*
⟨*proof*⟩

lemma *balance-left-sorted*: ⟦ *sorted l*; *sorted r*; *tree-less k l*; *tree-greater k r* ⟧ ⟹ *sorted (balance-left l k v r)*
⟨*proof*⟩

lemma *balance-left-tree-greater*:
fixes *k* :: *'a*::*order*
assumes *k* ≪| *a k* ≪| *b k < x*
shows *k* ≪| *balance-left a x t b*
⟨*proof*⟩

lemma *balance-left-tree-less*:
fixes *k* :: *'a*::*order*
assumes *a* |≪ *k b* |≪ *k x < k*
shows *balance-left a x t b* |≪ *k*
⟨*proof*⟩

lemma *balance-left-in-tree*:
assumes *inv1l l inv1 r bheight l + 1 = bheight r*
shows *entry-in-tree k v (balance-left l a b r) = (entry-in-tree k v l ∨ k = a ∧ v = b ∨ entry-in-tree k v r)*
⟨*proof*⟩

fun
balance-right :: (*'a*,*'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
balance-right a k x (Branch R b s y c) = Branch R a k x (Branch B b s y c) |
balance-right (Branch B a k x b) s y bl = balance (Branch R a k x b) s y bl |
balance-right (Branch R a k x (Branch B b s y c)) t z bl = Branch R (balance (paint R a) k x b) s y (Branch B c t z bl) |

balance-right t k x s = Empty

lemma *balance-right-inv2-with-inv1*:
assumes *inv2 lt inv2 rt bheight lt = bheight rt + 1 inv1 lt*
shows *inv2 (balance-right lt k v rt) ∧ bheight (balance-right lt k v rt) = bheight lt*
⟨*proof*⟩

lemma *balance-right-inv1*: ⟦*inv1 a*; *inv1l b*; *color-of a = B*⟧ ⟹ *inv1 (balance-right a k x b)*
⟨*proof*⟩

lemma *balance-right-inv1l*: ⟦ *inv1 lt*; *inv1l rt* ⟧ ⟹*inv1l (balance-right lt k x rt)*
⟨*proof*⟩

lemma *balance-right-sorted*: ⟦ *sorted l*; *sorted r*; *tree-less k l*; *tree-greater k r* ⟧ ⟹ *sorted (balance-right l k v r)*
⟨*proof*⟩

lemma *balance-right-tree-greater*:
fixes *k* :: *'a*::*order*
assumes *k* ≪| *a k* ≪| *b k < x*
shows *k* ≪| *balance-right a x t b*
⟨*proof*⟩

lemma *balance-right-tree-less*:
fixes *k* :: *'a*::*order*
assumes *a* |≪ *k b* |≪ *k x < k*
shows *balance-right a x t b* |≪ *k*
⟨*proof*⟩

lemma *balance-right-in-tree*:
assumes *inv1 l inv1l r bheight l = bheight r + 1 inv2 l inv2 r*
shows *entry-in-tree x y (balance-right l k v r) = (entry-in-tree x y l ∨ x = k ∧ y = v ∨ entry-in-tree x y r)*
⟨*proof*⟩

fun
combine :: (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
combine Empty x = x
| *combine x Empty = x*
| *combine (Branch R a k x b) (Branch R c s y d) = (case (combine b c) of*
Branch R b2 t z c2 ⇒ *(Branch R (Branch R a k x b2) t z (Branch R c2 s y d))* |
bc ⇒ *Branch R a k x (Branch R bc s y d))*
| *combine (Branch B a k x b) (Branch B c s y d) = (case (combine b c) of*
Branch R b2 t z c2 ⇒ *Branch R (Branch B a k x b2) t z (Branch B c2 s y d)* |

$bc \Rightarrow$ *balance-left a k x* (*Branch B bc s y d*))
| *combine a* (*Branch R b k x c*) = *Branch R* (*combine a b*) *k x c*
| *combine* (*Branch R a k x b*) *c* = *Branch R a k x* (*combine b c*)

lemma *combine-inv2*:
assumes *inv2 lt inv2 rt bheight lt* = *bheight rt*
shows *bheight* (*combine lt rt*) = *bheight lt inv2* (*combine lt rt*)
⟨*proof*⟩

lemma *combine-inv1*:
assumes *inv1 lt inv1 rt*
shows *color-of lt* = *B* ⟹ *color-of rt* = *B* ⟹ *inv1* (*combine lt rt*)
inv1l (*combine lt rt*)
⟨*proof*⟩

lemma *combine-tree-greater*[*simp*]:
fixes *k* :: *'a*::*linorder*
assumes *k* ≪| *l k* ≪| *r*
shows *k* ≪| *combine l r*
⟨*proof*⟩

lemma *combine-tree-less*[*simp*]:
fixes *k* :: *'a*::*linorder*
assumes *l* |≪ *k r* |≪ *k*
shows *combine l r* |≪ *k*
⟨*proof*⟩

lemma *combine-sorted*:
fixes *k* :: *'a*::*linorder*
assumes *sorted l sorted r l* |≪ *k k* ≪| *r*
shows *sorted* (*combine l r*)
⟨*proof*⟩

lemma *combine-in-tree*:
assumes *inv2 l inv2 r bheight l* = *bheight r inv1 l inv1 r*
shows *entry-in-tree k v* (*combine l r*) = (*entry-in-tree k v l* ∨ *entry-in-tree k v r*)
⟨*proof*⟩

fun
del-from-left :: (*'a*::*linorder*) ⇒ (*'a*,*'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
and
del-from-right :: (*'a*::*linorder*) ⇒ (*'a*,*'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt* **and**
del :: (*'a*::*linorder*) ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
del x Empty = *Empty* |
del x (*Branch c a y s b*) = (*if x* < *y then del-from-left x a y s b else* (*if x* > *y then del-from-right x a y s b else combine a b*)) |

del-from-left x (Branch B lt z v rt) y s b = balance-left (del x (Branch B lt z v rt)) y s b |
del-from-left x a y s b = Branch R (del x a) y s b |
del-from-right x a y s (Branch B lt z v rt) = balance-right a y s (del x (Branch B lt z v rt)) |
del-from-right x a y s b = Branch R a y s (del x b)

lemma
assumes *inv2 lt inv1 lt*
shows
$[\![$*inv2 rt*; *bheight lt = bheight rt*; *inv1 rt*$]\!] \Longrightarrow$
inv2 (del-from-left x lt k v rt) ∧ bheight (del-from-left x lt k v rt) = bheight lt ∧ (color-of lt = B ∧ color-of rt = B ∧ inv1 (del-from-left x lt k v rt) ∨ (color-of lt ≠ B ∨ color-of rt ≠ B) ∧ inv1l (del-from-left x lt k v rt))
and $[\![$*inv2 rt*; *bheight lt = bheight rt*; *inv1 rt*$]\!] \Longrightarrow$
inv2 (del-from-right x lt k v rt) ∧ bheight (del-from-right x lt k v rt) = bheight lt ∧ (color-of lt = B ∧ color-of rt = B ∧ inv1 (del-from-right x lt k v rt) ∨ (color-of lt ≠ B ∨ color-of rt ≠ B) ∧ inv1l (del-from-right x lt k v rt))
and *del-inv1-inv2*: *inv2 (del x lt) ∧ (color-of lt = R ∧ bheight (del x lt) = bheight lt ∧ inv1 (del x lt)*
∨ color-of lt = B ∧ bheight (del x lt) = bheight lt − 1 ∧ inv1l (del x lt))
⟨*proof*⟩

lemma
del-from-left-tree-less: $[\![$*tree-less v lt*; *tree-less v rt*; *k < v*$]\!] \Longrightarrow$ *tree-less v (del-from-left x lt k y rt)*
and *del-from-right-tree-less*: $[\![$*tree-less v lt*; *tree-less v rt*; *k < v*$]\!] \Longrightarrow$ *tree-less v (del-from-right x lt k y rt)*
and *del-tree-less*: *tree-less v lt* $\Longrightarrow$ *tree-less v (del x lt)*
⟨*proof*⟩

lemma *del-from-left-tree-greater*: $[\![$*tree-greater v lt*; *tree-greater v rt*; *k > v*$]\!] \Longrightarrow$ *tree-greater v (del-from-left x lt k y rt)*
and *del-from-right-tree-greater*: $[\![$*tree-greater v lt*; *tree-greater v rt*; *k > v*$]\!] \Longrightarrow$ *tree-greater v (del-from-right x lt k y rt)*
and *del-tree-greater*: *tree-greater v lt* $\Longrightarrow$ *tree-greater v (del x lt)*
⟨*proof*⟩

lemma $[\![$*sorted lt*; *sorted rt*; *tree-less k lt*; *tree-greater k rt*$]\!] \Longrightarrow$ *sorted (del-from-left x lt k y rt)*
and $[\![$*sorted lt*; *sorted rt*; *tree-less k lt*; *tree-greater k rt*$]\!] \Longrightarrow$ *sorted (del-from-right x lt k y rt)*
and *del-sorted*: *sorted lt* $\Longrightarrow$ *sorted (del x lt)*
⟨*proof*⟩

lemma $[\![$*sorted lt*; *sorted rt*; *tree-less kt lt*; *tree-greater kt rt*; *inv1 lt*; *inv1 rt*; *inv2 lt*; *inv2 rt*; *bheight lt = bheight rt*; *x < kt*$]\!] \Longrightarrow$ *entry-in-tree k v (del-from-left x lt kt y rt) = (False ∨ (x ≠ k ∧ entry-in-tree k v (Branch c lt kt y rt)))*
and $[\![$*sorted lt*; *sorted rt*; *tree-less kt lt*; *tree-greater kt rt*; *inv1 lt*; *inv1 rt*; *inv2 lt*;

inv2 rt; *bheight lt* = *bheight rt*; $x > kt$⟧ ⟹ *entry-in-tree k v* (*del-from-right x lt kt y rt*) = (*False* ∨ ($x \neq k$ ∧ *entry-in-tree k v* (*Branch c lt kt y rt*)))
and *del-in-tree*: ⟦*sorted t*; *inv1 t*; *inv2 t*⟧ ⟹ *entry-in-tree k v* (*del x t*) = (*False* ∨ ($x \neq k$ ∧ *entry-in-tree k v t*))
⟨*proof*⟩

definition *delete* **where**
delete-def: *delete k t* = *paint B* (*del k t*)

theorem *delete-is-rbt* [*simp*]: **assumes** *is-rbt t* **shows** *is-rbt* (*delete k t*)
⟨*proof*⟩

lemma *delete-in-tree*:
assumes *is-rbt t*
shows *entry-in-tree k v* (*delete x t*) = ($x \neq k$ ∧ *entry-in-tree k v t*)
⟨*proof*⟩

lemma *lookup-delete*:
assumes *is-rbt*: *is-rbt t*
shows *lookup* (*delete k t*) = (*lookup t*)|`(−{*k*})
⟨*proof*⟩

65.5 Union

primrec
union-with-key :: ('*a*::*linorder* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
union-with-key f t Empty = *t*
| *union-with-key f t* (*Branch c lt k v rt*) = *union-with-key f* (*union-with-key f* (*insert-with-key f k v t*) *lt*) *rt*

lemma *unionwk-sorted*: *sorted lt* ⟹ *sorted* (*union-with-key f lt rt*)
⟨*proof*⟩
theorem *unionwk-is-rbt*[*simp*]: *is-rbt lt* ⟹ *is-rbt* (*union-with-key f lt rt*)
⟨*proof*⟩

definition
union-with **where**
union-with f = *union-with-key* (λ-. *f*)

theorem *unionw-is-rbt*: *is-rbt lt* ⟹ *is-rbt* (*union-with f lt rt*) ⟨*proof*⟩

definition *union* **where**
union = *union-with-key* (%- - *rv*. *rv*)

theorem *union-is-rbt*: *is-rbt lt* ⟹ *is-rbt* (*union lt rt*) ⟨*proof*⟩

lemma *union-Branch*[*simp*]:
union t (*Branch c lt k v rt*) = *union* (*union* (*insert k v t*) *lt*) *rt*
⟨*proof*⟩

lemma *lookup-union*:
assumes *is-rbt s sorted t*
shows *lookup* (*union s t*) = *lookup s* ++ *lookup t*
⟨*proof*⟩

65.6 Modifying existing entries

primrec
map-entry :: *'a*::*linorder* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt*
where
map-entry k f Empty = *Empty*
| *map-entry k f* (*Branch c lt x v rt*) =
(*if k* < *x then Branch c* (*map-entry k f lt*) *x v rt*
else if k > *x then* (*Branch c lt x v* (*map-entry k f rt*))
else Branch c lt x (*f v*) *rt*)

lemma *map-entry-color-of*: *color-of* (*map-entry k f t*) = *color-of t* ⟨*proof*⟩
lemma *map-entry-inv1*: *inv1* (*map-entry k f t*) = *inv1 t* ⟨*proof*⟩
lemma *map-entry-inv2*: *inv2* (*map-entry k f t*) = *inv2 t bheight* (*map-entry k f t*)
= *bheight t* ⟨*proof*⟩
lemma *map-entry-tree-greater*: *tree-greater a* (*map-entry k f t*) = *tree-greater a t*
⟨*proof*⟩
lemma *map-entry-tree-less*: *tree-less a* (*map-entry k f t*) = *tree-less a t* ⟨*proof*⟩
lemma *map-entry-sorted*: *sorted* (*map-entry k f t*) = *sorted t*
⟨*proof*⟩

theorem *map-entry-is-rbt* [*simp*]: *is-rbt* (*map-entry k f t*) = *is-rbt t*
⟨*proof*⟩

theorem *lookup-map-entry*:
lookup (*map-entry k f t*) = (*lookup t*)(*k* := *Option.map f* (*lookup t k*))
⟨*proof*⟩

65.7 Mapping all entries

primrec
map :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'c*) *rbt*
where
map f Empty = *Empty*
| *map f* (*Branch c lt k v rt*) = *Branch c* (*map f lt*) *k* (*f k v*) (*map f rt*)

lemma *map-entries* [*simp*]: *entries* (*map f t*) = *List.map* (λ(*k*, *v*). (*k*, *f k v*))
(*entries t*)
⟨*proof*⟩
lemma *map-keys* [*simp*]: *keys* (*map f t*) = *keys t* ⟨*proof*⟩
lemma *map-tree-greater*: *tree-greater k* (*map f t*) = *tree-greater k t* ⟨*proof*⟩

lemma *map-tree-less*: *tree-less k* (*map f t*) = *tree-less k t* ⟨*proof*⟩
lemma *map-sorted*: *sorted* (*map f t*) = *sorted t* ⟨*proof*⟩
lemma *map-color-of*: *color-of* (*map f t*) = *color-of t* ⟨*proof*⟩
lemma *map-inv1*: *inv1* (*map f t*) = *inv1 t* ⟨*proof*⟩
lemma *map-inv2*: *inv2* (*map f t*) = *inv2 t bheight* (*map f t*) = *bheight t* ⟨*proof*⟩
theorem *map-is-rbt* [*simp*]: *is-rbt* (*map f t*) = *is-rbt t*
⟨*proof*⟩

theorem *lookup-map*: *lookup* (*map f t*) *x* = *Option.map* (*f x*) (*lookup t x*)
⟨*proof*⟩

65.8 Folding over entries

definition *fold* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a*, '*b*) *rbt* ⇒ '*c* ⇒ '*c* **where**
fold f t s = *foldl* (λ*s* (*k*, *v*). *f k v s*) *s* (*entries t*)

lemma *fold-simps* [*simp*, *code*]:
fold f Empty = *id*
fold f (*Branch c lt k v rt*) = *fold f rt* ∘ *f k v* ∘ *fold f lt*
⟨*proof*⟩

65.9 Bulkloading a tree

definition *bulkload* :: ('*a* × '*b*) *list* ⇒ ('*a*::*linorder*, '*b*) *rbt* **where**
bulkload xs = *foldr* (λ(*k*, *v*). *insert k v*) *xs Empty*

lemma *bulkload-is-rbt* [*simp*, *intro*]:
is-rbt (*bulkload xs*)
⟨*proof*⟩

lemma *lookup-bulkload*:
lookup (*bulkload xs*) = *map-of xs*
⟨*proof*⟩

hide-const (**open**) *Empty insert delete entries keys bulkload lookup map-entry map fold union sorted*

end

66 RBT: Abstract type of Red-Black Trees

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨

This theory defines abstract red-black trees as an efficient representation of finite maps, backed by the implementation in *RBT-Impl*.

66.1 Data type and invariant

The type $('k, 'v)$ *RBT-Impl.rbt* denotes red-black trees with keys of type $'k$ and values of type $'v$. To function properly, the key type musorted belong to the *linorder* class.

A value t of this type is a valid red-black tree if it satisfies the invariant *is-rbt t*. The abstract type $('k, 'v)$ *RBT.rbt* always obeys this invariant, and for this reason you should only use this in our application. Going back to $('k, 'v)$ *RBT-Impl.rbt* may be necessary in proofs if not yet proven properties about the operations must be established.

The interpretation function *RBT.lookup* returns the partial map represented by a red-black tree:

RBT.lookup$::('a, 'b)$ *RBT.rbt* $\Rightarrow 'a \rightharpoonup 'b$

This function should be used for reasoning about the semantics of the RBT operations. Furthermore, it implements the lookup functionality for the data structure: It is executable and the lookup is performed in $O(\log n)$.

66.2 Operations

Currently, the following operations are supported:

RBT.empty$::('a, 'b)$ *RBT.rbt*

Returns the empty tree. $O(1)$

RBT.insert$::'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *RBT.rbt* $\Rightarrow ('a, 'b)$ *RBT.rbt*

Updates the map at a given position. $O(\log n)$

RBT.delete$::'a \Rightarrow ('a, 'b)$ *RBT.rbt* $\Rightarrow ('a, 'b)$ *RBT.rbt*

Deletes a map entry at a given position. $O(\log n)$

RBT.entries$::('a, 'b)$ *RBT.rbt* $\Rightarrow ('a \times 'b)$ *list*

Return a corresponding key-value list for a tree.

RBT.bulkload$::('a \times 'b)$ *list* $\Rightarrow ('a, 'b)$ *RBT.rbt*

Builds a tree from a key-value list.

RBT.map-entry$::'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b)$ *RBT.rbt* $\Rightarrow ('a, 'b)$ *RBT.rbt*

Maps a single entry in a tree.

RBT.map$::('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b)$ *RBT.rbt* $\Rightarrow ('a, 'b)$ *RBT.rbt*

Maps all values in a tree. $O(n)$

RBT.fold$::('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a, 'b)$ *RBT.rbt* $\Rightarrow 'c \Rightarrow 'c$

Folds over all entries in a tree. $O(n)$

66.3 Invariant preservation

is-rbt rbt.Empty (*Empty-is-rbt*)
is-rbt ?t $\Longrightarrow$ *is-rbt* (*RBT-Impl.insert ?k ?v ?t*) (*insert-is-rbt*)
is-rbt ?t $\Longrightarrow$ *is-rbt* (*RBT-Impl.delete ?k ?t*) (*delete-is-rbt*)
is-rbt (*RBT-Impl.bulkload ?xs*) (*bulkload-is-rbt*)
is-rbt (*RBT-Impl.map-entry ?k ?f ?t*) = *is-rbt ?t* (*map-entry-is-rbt*)
is-rbt (*RBT-Impl.map ?f ?t*) = *is-rbt ?t* (*map-is-rbt*)
is-rbt ?lt $\Longrightarrow$ *is-rbt* (*RBT-Impl.union ?lt ?rt*) (*union-is-rbt*)

66.4 Map Semantics

lookup-empty

RBT.lookup RBT.empty = *Map.empty*

lookup-insert

RBT.lookup (*RBT.insert ?k ?v ?t*) = *RBT.lookup ?t*(*?k* $\mapsto$ *?v*)

lookup-delete

RBT.lookup (*RBT.delete ?k ?t*) = (*RBT.lookup ?t*)(*?k* := *None*)

lookup-bulkload

RBT.lookup (*RBT.bulkload ?xs*) = *map-of ?xs*

lookup-map

RBT.lookup (*RBT.map ?f ?t*) *?k* = *Option.map* (*?f ?k*) (*RBT.lookup ?t ?k*)

end

67 SML-Quickcheck: Install quickcheck of SML code generator

theory *SML-Quickcheck*
imports *Main*
begin

⟨*ML*⟩

end

68 State-Monad: Combinator syntax for generic, open state monads (single threaded monads)

theory *State-Monad*
imports *Main*
begin

68.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is transformed single-threaded).

To enter from the Haskell world, http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

68.2 State transformations and combinators

We classify functions operating on states into two categories:

transformations with type signature $\sigma \Rightarrow \sigma'$, transforming a state.

"yielding" transformations with type signature $\sigma \Rightarrow \alpha \times \sigma'$, "yielding" a side result while transforming a state.

queries with type signature $\sigma \Rightarrow \alpha$, computing a result dependent on a state.

By convention we write σ for types representing states and α, β, γ, ... for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type σ are used in a single-threaded way: after application of a transformation on a value of type σ, the former value should not be used again. To achieve this, we use a set of monad combinators:

notation *fcomp* (**infixl** *o>* *60*)
notation (*xsymbols*) *fcomp* (**infixl** *o>* *60*)
notation *scomp* (**infixl** *o->* *60*)
notation (*xsymbols*) *scomp* (**infixl** *o→* *60*)

abbreviation (*input*)
return ≡ *Pair*

Given two transformations f and g, they may be directly composed using the *op o>* combinator, forming a forward composition: $(f\ o{>}\ g)\ s = f\ (g\ s)$.

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the *op* $o\rightarrow$ combinator: $(f\ o\rightarrow\ (\lambda x.\ g))\ s = (let\ (x,\ s') = f\ s\ in\ g\ s')$.

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *Pair* for this purpose.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.
- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units ("open monad").
- The type of states may change due to a transformation.

68.3 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

lemmas *monad-simp = Pair-scomp scomp-Pair id-fcomp fcomp-id scomp-scomp scomp-fcomp fcomp-scomp fcomp-assoc*

Evaluation of monadic expressions by force:

lemmas *monad-collapse = monad-simp fcomp-apply scomp-apply split-beta*

68.4 Syntax

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

nonterminals *do-expr*

syntax

-do :: *do-expr* ⇒ *'a*
(*do - done* [*12*] *12*)
-scomp :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*- <− -;// -* [*1000*, *13*, *12*] *12*)
-fcomp :: *'a* ⇒ *do-expr* ⇒ *do-expr*
(*-;// -* [*13*, *12*] *12*)
-let :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*let - = -;// -* [*1000*, *13*, *12*] *12*)

-done :: *'a* ⇒ *do-expr*
(- [*12*] *12*)

syntax (*xsymbols*)
-scomp :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(- ← -;// - [*1000*, *13*, *12*] *12*)

translations
-do f => *f*
-scomp x f g => *f o→* (λ*x*. *g*)
-fcomp f g => *f o*> *g*
-let x t f => *CONST Let t* (λ*x*. *f*)
-done f => *f*

⟨*ML*⟩

For an example, see HOL/Extraction/Higman.thy.

end

69 Sum-Of-Squares: A decision method for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming

theory *Sum-Of-Squares*
imports *Complex-Main*
uses
positivstellensatz.ML
Sum-Of-Squares/sum-of-squares.ML
Sum-Of-Squares/positivstellensatz-tools.ML
Sum-Of-Squares/sos-wrapper.ML
begin

In order to use the method sos, call it with (*sos remote-csdp*) to use the remote solver. Or install CSDP (https://projects.coin-or.org/Csdp), configure the Isabelle setting *CSDP-EXE*, and call it with (*sos csdp*). By default, sos calls *remote-csdp*. This can take of the order of a minute for one sos call, because sos calls CSDP repeatedly. If you install CSDP locally, sos calls typically takes only a few seconds. sos generates a certificate which can be used to repeat the proof without calling an external prover.

⟨*ML*⟩
Tests

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) <= x & 0 <= y & 0 <= z & x + y + z <= 3 −−> x * y + x * z + y * z >= 3 * x * y * z*
⟨*proof*⟩

lemma *((x::real)^2 + y^2 + z^2 = 1) −−> (x + y + z)^2 <= 3*
⟨*proof*⟩

lemma *(w^2 + x^2 + y^2 + z^2 = 1) −−> (w + x + y + z)^2 <= (4::real)*
⟨*proof*⟩

lemma *(x::real) >= 1 & y >= 1 −−> x * y >= x + y − 1*
⟨*proof*⟩

lemma *(x::real) > 1 & y > 1 −−> x * y > x + y − 1*
⟨*proof*⟩

lemma *abs(x) <= 1 −−> abs(64 * x^7 − 112 * x^5 + 56 * x^3 − 7 * x) <= (1::real)*
⟨*proof*⟩

lemma *2 <= x & x <= 125841 / 50000 & 2 <= y & y <= 125841 / 50000 & 2 <= z & z <= 125841 / 50000 −−> 2 * (x * z + x * y + y * z) − (x * x + y * y + z * z) >= (0::real)*
⟨*proof*⟩

lemma *(2::real) <= x & x <= 4 & 2 <= y & y <= 4 & 2 <= z & z <= 4 −−> 0 <= 2 * (x * z + x * y + y * z) − (x * x + y * y + z * z)*
⟨*proof*⟩

lemma *2 <= (x::real) & x <= 4 & 2 <= y & y <= 4 & 2 <= z & z <= 4 --> 12 <= 2 * (x * z + x * y + y * z) - (x * x + y * y + z * z)*
⟨*proof*⟩

lemma *0 <= (x::real) & 0 <= y & (x * y = 1) --> x + y <= x^2 + y^2*
⟨*proof*⟩

lemma *0 <= (x::real) & 0 <= y & (x * y = 1) --> x * y * (x + y) <= x^2 + y^2*
⟨*proof*⟩

lemma *0 <= (x::real) & 0 <= y --> x * y * (x + y) ^2 <= (x^2 + y^2) ^2*
⟨*proof*⟩

lemma *(0::real) <= a & 0 <= b & 0 <= c & c * (2 * a + b) ^3/ 27 <= x ⟶ c * a^2 * b <= x*
⟨*proof*⟩

lemma *(0::real) < x --> 0 < 1 + x + x^2*
⟨*proof*⟩

lemma *(0::real) <= x --> 0 < 1 + x + x^2*
⟨*proof*⟩

lemma *(0::real) < 1 + x^2*
⟨*proof*⟩

lemma *(0::real) <= 1 + 2 * x + x^2*
⟨*proof*⟩

lemma *(0::real) < 1 + abs x*
⟨*proof*⟩

lemma *(0::real) < 1 + (1 + x) ^2 * (abs x)*
⟨*proof*⟩

lemma *abs ((1::real) + x^2) = (1::real) + x^2*
⟨*proof*⟩
lemma *(3::real) * x + 7 * a < 4 ∧ 3 < 2 * x ⟶ a < 0*
⟨*proof*⟩

lemma $(0::real) < x \longrightarrow 1 < y \longrightarrow y * x <= z \longrightarrow x < z$
⟨*proof*⟩
lemma $(1::real) < x \longrightarrow x\hat{}2 < y \longrightarrow 1 < y$
⟨*proof*⟩
lemma $(b::real)\hat{}2 < 4 * a * c \longrightarrow \sim(a * x\hat{}2 + b * x + c = 0)$
⟨*proof*⟩
lemma $(b::real)\hat{}2 < 4 * a * c \longrightarrow \sim(a * x\hat{}2 + b * x + c = 0)$
⟨*proof*⟩
lemma $((a::real) * x\hat{}2 + b * x + c = 0) \longrightarrow b\hat{}2 >= 4 * a * c$
⟨*proof*⟩
lemma $(0::real) <= b \;\&\; 0 <= c \;\&\; 0 <= x \;\&\; 0 <= y \;\&\; (x\hat{}2 = c) \;\&\; (y\hat{}2 = a\hat{}2 * c + b) \longrightarrow a * c <= y * x$
⟨*proof*⟩
lemma $abs(x - z) <= e \;\&\; abs(y - z) <= e \;\&\; 0 <= u \;\&\; 0 <= v \;\&\; (u + v = 1) \longrightarrow abs((u * x + v * y) - z) <= (e::real)$
⟨*proof*⟩

lemma $(0::real) <= x \longrightarrow (1 + x + x\hat{}2)/(1 + x\hat{}2) <= 1 + x$
⟨*proof*⟩

lemma $(0::real) <= x \longrightarrow 1 - x <= 1 \;/\; (1 + x + x\hat{}2)$
⟨*proof*⟩

lemma $(x::real) <= 1 \;/\; 2 \longrightarrow - x - 2 * x\hat{}2 <= - x \;/\; (1 - x)$
⟨*proof*⟩

lemma $4*r\hat{}2 = p\hat{}2 - 4*q \;\&\; r >= (0::real) \;\&\; x\hat{}2 + p*x + q = 0 \longrightarrow 2*(x::real) = - p + 2*r \mid 2*x = -p - 2*r$
⟨*proof*⟩

end

70 Transitive-Closure-Table: A tabled implementation of the reflexive transitive closure

theory *Transitive-Closure-Table*
imports *Main*
begin

inductive *rtrancl-path* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow bool$
  for $r :: 'a \Rightarrow 'a \Rightarrow bool$
where

base: *rtrancl-path r x [] x*
| *step*: *r x y ⟹ rtrancl-path r y ys z ⟹ rtrancl-path r x (y # ys) z*

lemma *rtranclp-eq-rtrancl-path*: *r** x y = (∃ xs. rtrancl-path r x xs y)*
⟨*proof*⟩

lemma *rtrancl-path-trans*:
assumes *xy*: *rtrancl-path r x xs y*
and *yz*: *rtrancl-path r y ys z*
shows *rtrancl-path r x (xs @ ys) z* ⟨*proof*⟩

lemma *rtrancl-path-appendE*:
assumes *xz*: *rtrancl-path r x (xs @ y # ys) z*
obtains *rtrancl-path r x (xs @ [y]) y* **and** *rtrancl-path r y ys z* ⟨*proof*⟩

lemma *rtrancl-path-distinct*:
assumes *xy*: *rtrancl-path r x xs y*
obtains *xs'* **where** *rtrancl-path r x xs' y* **and** *distinct (x # xs')* ⟨*proof*⟩

inductive *rtrancl-tab* :: *('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a ⇒ 'a ⇒ bool*
for *r* :: *'a ⇒ 'a ⇒ bool*
where
base: *rtrancl-tab r xs x x*
| *step*: *x ∉ set xs ⟹ r x y ⟹ rtrancl-tab r (x # xs) y z ⟹ rtrancl-tab r xs x z*

lemma *rtrancl-path-imp-rtrancl-tab*:
assumes *path*: *rtrancl-path r x xs y*
and *x*: *distinct (x # xs)*
and *ys*: *({x} ∪ set xs) ∩ set ys = {}*
shows *rtrancl-tab r ys x y* ⟨*proof*⟩

lemma *rtrancl-tab-imp-rtrancl-path*:
assumes *tab*: *rtrancl-tab r ys x y*
obtains *xs* **where** *rtrancl-path r x xs y* ⟨*proof*⟩

lemma *rtranclp-eq-rtrancl-tab-nil*: *r** x y = rtrancl-tab r [] x y*
⟨*proof*⟩

declare *rtranclp-eq-rtrancl-tab-nil* [*code-unfold*, *code-inline del*]

declare *rtranclp-eq-rtrancl-tab-nil*[*THEN iffD2*, *code-pred-intro*]

code-pred *rtranclp* ⟨*proof*⟩

70.1 A simple example

datatype *ty = A | B | C*

inductive *test* :: *ty ⇒ ty ⇒ bool*

where
  test A B
| *test B A*
| *test B C*

70.1.1 Invoking with the SML code generator

code-module *Test*
contains
test1 = *test*** *A C*
test2 = *test*** *C A*
test3 = *test*** *A* -
test4 = *test*** - *C*

⟨*ML*⟩

70.1.2 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*

hide-type *ty*
hide-const *test A B C*

end

71 Univ-Poly: Univariate Polynomials

theory *Univ-Poly*
imports *Main*
begin

Application of polynomial as a function.

primrec (**in** *semiring-0*) *poly* :: *'a list* => *'a* => *'a* **where**
  poly-Nil: *poly* [] *x* = *0*
| *poly-Cons*: *poly* (*h*#*t*) *x* = *h* + *x* * *poly t x*

71.1 Arithmetic Operations on Polynomials

addition

primrec (**in** *semiring-0*) *padd* :: *'a list* ⇒ *'a list* ⇒ *'a list* (**infixl** *+++ 65*)
where
padd-Nil: [] *+++ l2* = *l2*
| *padd-Cons*: (*h*#*t*) *+++ l2* = (*if l2* = [] *then h*#*t*
else (*h* + *hd l2*)#(*t +++ tl l2*))

Multiplication by a constant

primrec (**in** *semiring-0*) *cmult* :: *'a* ⇒ *'a list* ⇒ *'a list* (**infixl** *%** *70*) **where**
cmult-Nil: *c %* [] = []*
| *cmult-Cons*: *c %* (h#t) = (c * h)#(c %* t)*

Multiplication by a polynomial

primrec (**in** *semiring-0*) *pmult* :: *'a list* ⇒ *'a list* ⇒ *'a list* (**infixl** ***** *70*)
where
pmult-Nil: [] **** l2* = []
| *pmult-Cons*: (*h*#*t*) **** l2* = (*if t* = [] *then h %* l2*
else (*h %* l2*) *+++* ((*0*) # (*t *** l2*)))

Repeated multiplication by a polynomial

primrec (**in** *semiring-0*) *mulexp* :: *nat* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
mulexp-zero: *mulexp 0 p q* = *q*
| *mulexp-Suc*: *mulexp* (*Suc n*) *p q* = *p *** mulexp n p q*

Exponential

primrec (**in** *semiring-1*) *pexp* :: *'a list* ⇒ *nat* ⇒ *'a list* (**infixl** *%^ 80*) **where**
pexp-0: *p %^ 0* = [*1*]
| *pexp-Suc*: *p %^* (*Suc n*) = *p **** (*p %^ n*)

Quotient related value of dividing a polynomial by x + a

primrec (**in** *field*) *pquot* :: *'a list* ⇒ *'a* ⇒ *'a list* **where**
pquot-Nil: *pquot* [] *a*= []
| *pquot-Cons*: *pquot* (*h*#*t*) *a* = (*if t* = [] *then* [*h*]
else (*inverse*(*a*) * (*h* − *hd*(*pquot t a*)))#(*pquot t a*))

normalization of polynomials (remove extra 0 coeff)

primrec (**in** *semiring-0*) *pnormalize* :: *'a list* ⇒ *'a list* **where**
pnormalize-Nil: *pnormalize* [] = []
| *pnormalize-Cons*: *pnormalize* (*h*#*p*) = (*if* ((*pnormalize p*) = [])
then (*if* (*h* = *0*) *then* [] *else* [*h*])
else (*h*#(*pnormalize p*)))

definition (**in** *semiring-0*) *pnormal p* = ((*pnormalize p* = *p*) ∧ *p* ≠ [])
definition (**in** *semiring-0*) *nonconstant p* = (*pnormal p* ∧ (∀ *x*. *p* ≠ [*x*]))

Other definitions

definition (**in** *ring-1*)

poly-minus :: *'a list => 'a list* (*-- - [80] 80*) **where**
-- p = (- 1) % p*

definition (**in** *semiring-0*)
divides :: *'a list ⇒ 'a list ⇒ bool* (**infixl** *divides 70*) **where**
[*code del*]: *p1 divides p2 = (∃ q. poly p2 = poly(p1 *** q))*

— order of a polynomial
definition (**in** *ring-1*) *order* :: *'a => 'a list => nat* **where**
order a p = (SOME n. ([−a, 1] %^ n) divides p &
~ (([−a, 1] %^ (Suc n)) divides p))

— degree of a polynomial
definition (**in** *semiring-0*) *degree* :: *'a list => nat* **where**
degree p = length (pnormalize p) − 1

— squarefree polynomials — NB with respect to real roots only.
definition (**in** *ring-1*)
rsquarefree :: *'a list => bool* **where**
rsquarefree p = (poly p ≠ poly [] &
(∀ a. (order a p = 0) | (order a p = 1)))

context *semiring-0*
begin

lemma *padd-Nil2*[*simp*]: *p +++ [] = p*
⟨*proof*⟩

lemma *padd-Cons-Cons*: *(h1 # p1) +++ (h2 # p2) = (h1 + h2) # (p1 +++ p2)*
⟨*proof*⟩

lemma *pminus-Nil*[*simp*]: *-- [] = []*
⟨*proof*⟩

lemma *pmult-singleton*: *[h1] *** p1 = h1 %* p1* ⟨*proof*⟩
end

lemma (**in** *semiring-1*) *poly-ident-mult*[*simp*]: *1 %* t = t* ⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-simple-add-Cons*[*simp*]: *[a] +++ ((0)#t) = (a#t)*
⟨*proof*⟩

Handy general properties

lemma (**in** *comm-semiring-0*) *padd-commut*: *b +++ a = a +++ b*
⟨*proof*⟩

lemma (**in** *comm-semiring-0*) *padd-assoc*: *∀ b c. (a +++ b) +++ c = a +++ (b +++ c)*

⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-cmult-distr*: *a %* (p +++ q) = (a %* p +++ a %* q)*
⟨*proof*⟩

lemma (**in** *ring-1*) *pmult-by-x*[*simp*]: *[0, 1] *** t = ((0)#t)*
⟨*proof*⟩

properties of evaluation of polynomials.

lemma (**in** *semiring-0*) *poly-add*: *poly (p1 +++ p2) x = poly p1 x + poly p2 x*
⟨*proof*⟩

lemma (**in** *comm-semiring-0*) *poly-cmult*: *poly (c %* p) x = c * poly p x*
⟨*proof*⟩

lemma (**in** *comm-semiring-0*) *poly-cmult-map*: *poly (map (op * c) p) x = c*poly p x*
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-minus*: *poly (-- p) x = - (poly p x)*
⟨*proof*⟩

lemma (**in** *comm-semiring-0*) *poly-mult*: *poly (p1 *** p2) x = poly p1 x * poly p2 x*
⟨*proof*⟩

class *idom-char-0 = idom + ring-char-0*

lemma (**in** *comm-ring-1*) *poly-exp*: *poly (p %^ n) x = (poly p x) ^ n*
⟨*proof*⟩

More Polynomial Evaluation Lemmas

lemma (**in** *semiring-0*) *poly-add-rzero*[*simp*]: *poly (a +++ []) x = poly a x*
⟨*proof*⟩

lemma (**in** *comm-semiring-0*) *poly-mult-assoc*: *poly ((a *** b) *** c) x = poly (a *** (b *** c)) x*
⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-mult-Nil2*[*simp*]: *poly (p *** []) x = 0*
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *poly-exp-add*: *poly (p %^ (n + d)) x = poly(p %^ n *** p %^ d) x*
⟨*proof*⟩

71.2 Key Property: if $f\ a = (0::'a)$ then $x - a$ divides $p\ x$

lemma (**in** *comm-ring-1*) *lemma-poly-linear-rem*: $\forall\, h.\ \exists\, q\ r.\ h\#t = [r] +++ [-a,\ 1] \ast\ast\ast\ q$
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-linear-rem*: $\exists\, q\ r.\ h\#t = [r] +++ [-a,\ 1] \ast\ast\ast\ q$
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-linear-divides*: $(poly\ p\ a = 0) = ((p = []) \mid (\exists\, q.\ p = [-a,\ 1] \ast\ast\ast\ q))$
⟨*proof*⟩

lemma (**in** *semiring-0*) *lemma-poly-length-mult*[*simp*]: $\forall\, h\ k\ a.\ length\ (k\ \%\ast\ p +++\ (h\ \#\ (a\ \%\ast\ p))) = Suc\ (length\ p)$
⟨*proof*⟩

lemma (**in** *semiring-0*) *lemma-poly-length-mult2*[*simp*]: $\forall\, h\ k.\ length\ (k\ \%\ast\ p +++\ (h\ \#\ p)) = Suc\ (length\ p)$
⟨*proof*⟩

lemma (**in** *ring-1*) *poly-length-mult*[*simp*]: $length([-a,1] \ast\ast\ast\ q) = Suc\ (length\ q)$
⟨*proof*⟩

71.3 Polynomial length

lemma (**in** *semiring-0*) *poly-cmult-length*[*simp*]: $length\ (a\ \%\ast\ p) = length\ p$
⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-add-length*: $length\ (p1 +++\ p2) = max\ (length\ p1)\ (length\ p2)$
⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-root-mult-length*[*simp*]: $length([a,b] \ast\ast\ast\ p) = Suc\ (length\ p)$
⟨*proof*⟩

lemma (**in** *idom*) *poly-mult-not-eq-poly-Nil*[*simp*]:
$poly\ (p \ast\ast\ast\ q)\ x \neq poly\ []\ x \longleftrightarrow poly\ p\ x \neq poly\ []\ x \wedge poly\ q\ x \neq poly\ []\ x$
⟨*proof*⟩

lemma (**in** *idom*) *poly-mult-eq-zero-disj*: $poly\ (p \ast\ast\ast\ q)\ x = 0 \longleftrightarrow poly\ p\ x = 0 \vee poly\ q\ x = 0$
⟨*proof*⟩

Normalisation Properties

lemma (**in** *semiring-0*) *poly-normalized-nil*: $(pnormalize\ p = []) \dashrightarrow (poly\ p\ x = 0)$
⟨*proof*⟩

A nontrivial polynomial of degree n has no more than n roots

lemma (**in** *idom*) *poly-roots-index-lemma*:
assumes *p*: *poly p x* ≠ *poly* [] *x* **and** *n*: *length p* = *n*
shows ∃ *i*. ∀ *x*. *poly p x* = *0* ⟶ (∃ *m*≤*n*. *x* = *i m*)
⟨*proof*⟩

lemma (**in** *idom*) *poly-roots-index-length*: *poly p x* ≠ *poly* [] *x* ==>
∃ *i*. ∀ *x*. (*poly p x* = *0*) −−> (∃ *n*. *n* ≤ *length p* & *x* = *i n*)
⟨*proof*⟩

lemma (**in** *idom*) *poly-roots-finite-lemma1*: *poly p x* ≠ *poly* [] *x* ==>
∃ *N i*. ∀ *x*. (*poly p x* = *0*) −−> (∃ *n*. (*n*::*nat*) < *N* & *x* = *i n*)
⟨*proof*⟩

lemma (**in** *idom*) *idom-finite-lemma*:
assumes *P*: ∀ *x*. *P x* −−> (∃ *n*. *n* < *length j* & *x* = *j*!*n*)
shows *finite* {*x*. *P x*}
⟨*proof*⟩

lemma (**in** *idom*) *poly-roots-finite-lemma2*: *poly p x* ≠ *poly* [] *x* ==>
∃ *i*. ∀ *x*. (*poly p x* = *0*) −−> *x* ∈ *set i*
⟨*proof*⟩

lemma (**in** *ring-char-0*) *UNIV-ring-char-0-infinte*:
¬ (*finite* (*UNIV*:: *'a set*))
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-roots-finite*: (*poly p* ≠ *poly* []) =
finite {*x*. *poly p x* = *0*}
⟨*proof*⟩

Entirety and Cancellation for polynomials

lemma (**in** *idom-char-0*) *poly-entire-lemma2*:
assumes *p0*: *poly p* ≠ *poly* [] **and** *q0*: *poly q* ≠ *poly* []
shows *poly* (*p****q*) ≠ *poly* []
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-entire*:
poly (*p* *** *q*) = *poly* [] ⟷ *poly p* = *poly* [] ∨ *poly q* = *poly* []
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-entire-neg*: (*poly* (*p* *** *q*) ≠ *poly* []) = ((*poly p* ≠ *poly* []) & (*poly q* ≠ *poly* []))
⟨*proof*⟩

lemma *fun-eq*: (*f* = *g*) = (∀ *x*. *f x* = *g x*)

⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-add-minus-zero-iff*: (*poly* (*p* +++ −− *q*) = *poly* []) = (*poly p* = *poly q*)
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-add-minus-mult-eq*: *poly* (*p* *** *q* +++ −−(*p* *** *r*)) = *poly* (*p* *** (*q* +++ −− *r*))
⟨*proof*⟩

subclass (**in** *idom-char-0*) *comm-ring-1* ⟨*proof*⟩
lemma (**in** *idom-char-0*) *poly-mult-left-cancel*: (*poly* (*p* *** *q*) = *poly* (*p* *** *r*)) = (*poly p* = *poly* [] | *poly q* = *poly r*)
⟨*proof*⟩

lemma (**in** *idom*) *poly-exp-eq-zero*[*simp*]:
(*poly* (*p* %^ *n*) = *poly* []) = (*poly p* = *poly* [] & *n* ≠ *0*)
⟨*proof*⟩

lemma (**in** *semiring-1*) *one-neq-zero*[*simp*]: *1* ≠ *0* ⟨*proof*⟩
lemma (**in** *comm-ring-1*) *poly-prime-eq-zero*[*simp*]: *poly* [*a*,*1*] ≠ *poly* []
⟨*proof*⟩

lemma (**in** *idom*) *poly-exp-prime-eq-zero*: (*poly* ([*a*, *1*] %^ *n*) ≠ *poly* [])
⟨*proof*⟩

A more constructive notion of polynomials being trivial

lemma (**in** *idom-char-0*) *poly-zero-lemma'*: *poly* (*h* # *t*) = *poly* [] ==> *h* = *0* & *poly t* = *poly* []
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-zero*: (*poly p* = *poly* []) = *list-all* (%*c*. *c* = *0*) *p*
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-0*: *list-all* (λ*c*. *c* = *0*) *p* ⟹ *poly p x* = *0*
⟨*proof*⟩

Basics of divisibility.

lemma (**in** *idom*) *poly-primes*: ([*a*, *1*] *divides* (*p* *** *q*)) = ([*a*, *1*] *divides p* | [*a*, *1*] *divides q*)
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *poly-divides-refl*[*simp*]: *p divides p*
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *poly-divides-trans*: [| *p divides q*; *q divides r* |] ==> *p divides r*
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *poly-divides-exp*: $m \leq n \Longrightarrow (p \,\%\hat{}\, m)$ *divides* $(p \,\%\hat{}\, n)$
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *poly-exp-divides*: $[\!|\ (p \,\%\hat{}\, n)$ *divides* $q;\ \ m \leq n\ |\!] \Longrightarrow (p \,\%\hat{}\, m)$ *divides* q
⟨*proof*⟩

lemma (**in** *comm-semiring-0*) *poly-divides-add*:
$[\!|\ p$ *divides* $q;\ p$ *divides* $r\ |\!] \Longrightarrow p$ *divides* $(q +\!\!+\!\!+ r)$
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-divides-diff*:
$[\!|\ p$ *divides* $q;\ p$ *divides* $(q +\!\!+\!\!+ r)\ |\!] \Longrightarrow p$ *divides* r
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-divides-diff2*: $[\!|\ p$ *divides* $r;\ p$ *divides* $(q +\!\!+\!\!+ r)\ |\!] \Longrightarrow p$ *divides* q
⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-divides-zero*: *poly* $p =$ *poly* $[] \Longrightarrow q$ *divides* p
⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-divides-zero2*[*simp*]: q *divides* $[]$
⟨*proof*⟩

At last, we can consider the order of a root.

lemma (**in** *idom-char-0*) *poly-order-exists-lemma*:
assumes *lp*: *length* $p = d$ **and** p: *poly* $p \neq$ *poly* $[]$
shows $\exists n\ q.\ p =$ *mulexp* $n\ [-a,\ 1]\ q \wedge$ *poly* $q\ a \neq 0$
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *poly-mulexp*: *poly* $(mulexp\ n\ p\ q)\ x = (poly\ p\ x)\ \hat{}\ n * poly\ q\ x$
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *divides-left-mult*:
assumes d:$(p{*}{*}{*}q)$ *divides* r **shows** p *divides* $r \wedge q$ *divides* r
⟨*proof*⟩

lemma (**in** *semiring-1*)
zero-power-iff: $0\ \hat{}\ n = (if\ n = 0\ then\ 1\ else\ 0)$
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-order-exists*:
  assumes *lp*: *length p* = *d* **and** *p0*: *poly p* ≠ *poly* []
  shows ∃ *n*. ([−*a*, *1*] %^ *n*) *divides p* & ~(([−*a*, *1*] %^ (*Suc n*)) *divides p*)
⟨*proof*⟩

lemma (**in** *semiring-1*) *poly-one-divides*[*simp*]: [*1*] *divides p*
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-order*: *poly p* ≠ *poly* []
    ==> *EX*! *n*. ([−*a*, *1*] %^ *n*) *divides p* &
            ~(([−*a*, *1*] %^ (*Suc n*)) *divides p*)
⟨*proof*⟩

Order

lemma *some1-equalityD*: [| *n* = (@*n*. *P n*); *EX*! *n*. *P n* |] ==> *P n*
⟨*proof*⟩

lemma (**in** *idom-char-0*) *order*:
    (([−*a*, *1*] %^ *n*) *divides p* &
      ~(([−*a*, *1*] %^ (*Suc n*)) *divides p*)) =
      ((*n* = *order a p*) & ~(*poly p* = *poly* []))
⟨*proof*⟩

lemma (**in** *idom-char-0*) *order2*: [| *poly p* ≠ *poly* [] |]
    ==> ([−*a*, *1*] %^ (*order a p*)) *divides p* &
          ~(([−*a*, *1*] %^ (*Suc*(*order a p*))) *divides p*)
⟨*proof*⟩

lemma (**in** *idom-char-0*) *order-unique*: [| *poly p* ≠ *poly* []; ([−*a*, *1*] %^ *n*) *divides p*;
        ~(([−*a*, *1*] %^ (*Suc n*)) *divides p*)
    |] ==> (*n* = *order a p*)
⟨*proof*⟩

lemma (**in** *idom-char-0*) *order-unique-lemma*: (*poly p* ≠ *poly* [] & ([−*a*, *1*] %^ *n*) *divides p* &
        ~(([−*a*, *1*] %^ (*Suc n*)) *divides p*))
    ==> (*n* = *order a p*)
⟨*proof*⟩

lemma (**in** *ring-1*) *order-poly*: *poly p* = *poly q* ==> *order a p* = *order a q*
⟨*proof*⟩

lemma (**in** *semiring-1*) *pexp-one*[*simp*]: *p* %^ (*Suc 0*) = *p*
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *lemma-order-root*:

$0 < n$ & $[-a, 1] \,\%\hat{}\, n$ *divides* p & $\sim [-a, 1] \,\%\hat{}\, (Suc\ n)$ *divides* p
$\Longrightarrow$ *poly* $p\ a = 0$
⟨*proof*⟩

lemma (**in** *idom-char-0*) *order-root*: (*poly* $p\ a = 0$) = ((*poly* p = *poly* []) | *order* $a\ p \neq 0$)
⟨*proof*⟩

lemma (**in** *idom-char-0*) *order-divides*: (([$-a$, 1] $\%\hat{}\, n$) *divides* p) = ((*poly* p = *poly* []) | $n \leq$ *order* $a\ p$)
⟨*proof*⟩

lemma (**in** *idom-char-0*) *order-decomp*:
poly $p \neq$ *poly* []
==> $\exists q$. (*poly* p = *poly* (([$-a$, 1] $\%\hat{}$ (*order* $a\ p$)) *** q)) &
$\sim$([$-a$, 1] *divides* q)
⟨*proof*⟩

Important composition properties of orders.

lemma *order-mult*: *poly* (p *** q) $\neq$ *poly* []
==> *order* a (p *** q) = *order* $a\ p$ + *order* (a::'a::{*idom-char-0*}) q
⟨*proof*⟩

lemma (**in** *idom-char-0*) *order-mult*:
assumes *pq0*: *poly* (p *** q) $\neq$ *poly* []
shows *order* a (p *** q) = *order* $a\ p$ + *order* $a\ q$
⟨*proof*⟩

lemma (**in** *idom-char-0*) *order-root2*: *poly* $p \neq$ *poly* [] ==> (*poly* $p\ a = 0$) = (*order* $a\ p \neq 0$)
⟨*proof*⟩

lemma (**in** *semiring-1*) *pmult-one*[*simp*]: [1] *** $p = p$ ⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-Nil-zero*: *poly* [] = *poly* [0]
⟨*proof*⟩

lemma (**in** *idom-char-0*) *rsquarefree-decomp*:
[| *rsquarefree* p; *poly* $p\ a = 0$ |]
==> $\exists q$. (*poly* p = *poly* ([$-a$, 1] *** q)) & *poly* $q\ a \neq 0$
⟨*proof*⟩

Normalization of a polynomial.

lemma (**in** *semiring-0*) *poly-normalize*[*simp*]: *poly* (*pnormalize* p) = *poly* p
⟨*proof*⟩

The degree of a polynomial.

lemma (**in** *semiring-0*) *lemma-degree-zero*:
list-all (%c. $c = 0$) $p \longleftrightarrow$ *pnormalize* p = []

⟨*proof*⟩

lemma (**in** *idom-char-0*) *degree-zero*:
assumes *pN*: *poly p* = *poly* [] **shows***degree p* = *0*
⟨*proof*⟩

lemma (**in** *semiring-0*) *pnormalize-sing*: (*pnormalize* [*x*] = [*x*]) ⟷ *x* ≠ *0*
⟨*proof*⟩
lemma (**in** *semiring-0*) *pnormalize-pair*: *y* ≠ *0* ⟷ (*pnormalize* [*x*, *y*] = [*x*, *y*])
⟨*proof*⟩
lemma (**in** *semiring-0*) *pnormal-cons*: *pnormal p* ⟹ *pnormal* (*c*#*p*)
⟨*proof*⟩
lemma (**in** *semiring-0*) *pnormal-tail*: *p*≠[] ⟹ *pnormal* (*c*#*p*) ⟹ *pnormal p*
⟨*proof*⟩

lemma (**in** *semiring-0*) *pnormal-last-nonzero*: *pnormal p* ==> *last p* ≠ *0*
⟨*proof*⟩

lemma (**in** *semiring-0*) *pnormal-length*: *pnormal p* ⟹ *0* < *length p*
⟨*proof*⟩

lemma (**in** *semiring-0*) *pnormal-last-length*: ⟦*0* < *length p* ; *last p* ≠ *0*⟧ ⟹ *pnormal p*
⟨*proof*⟩

lemma (**in** *semiring-0*) *pnormal-id*: *pnormal p* ⟷ (*0* < *length p* ∧ *last p* ≠ *0*)
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-Cons-eq*: *poly* (*c*#*cs*) = *poly* (*d*#*ds*) ⟷ *c*=*d* ∧ *poly cs* = *poly ds* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma (**in** *idom-char-0*) *pnormalize-unique*: *poly p* = *poly q* ⟹ *pnormalize p* = *pnormalize q*
⟨*proof*⟩

lemma (**in** *idom-char-0*) *degree-unique*: **assumes** *pq*: *poly p* = *poly q*
shows *degree p* = *degree q*
⟨*proof*⟩

lemma (**in** *semiring-0*) *pnormalize-length*: *length* (*pnormalize p*) ≤ *length p* ⟨*proof*⟩

lemma (**in** *semiring-0*) *last-linear-mul-lemma*:
last ((*a* %∗ *p*) +++ (*x*#(*b* %∗ *p*))) = (*if p*=[] *then x else b*∗*last p*)

⟨*proof*⟩

lemma (**in** *semiring-1*) *last-linear-mul*: **assumes** *p*:*p*≠[] **shows** *last* ([*a*,*1*] ***** *p*) = *last p*
⟨*proof*⟩

lemma (**in** *semiring-0*) *pnormalize-eq*: *last p* ≠ *0* ⟹ *pnormalize p* = *p*
⟨*proof*⟩

lemma (**in** *semiring-0*) *last-pnormalize*: *pnormalize p* ≠ [] ⟹ *last* (*pnormalize p*) ≠ *0*
⟨*proof*⟩

lemma (**in** *semiring-0*) *pnormal-degree*: *last p* ≠ *0* ⟹ *degree p* = *length p* − *1*
⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-Nil-ext*: *poly* [] = (λ*x*. *0*) ⟨*proof*⟩

lemma (**in** *idom-char-0*) *linear-mul-degree*: **assumes** *p*: *poly p* ≠ *poly* []
shows *degree* ([*a*,*1*] ***** *p*) = *degree p* + *1*
⟨*proof*⟩

lemma (**in** *idom-char-0*) *linear-pow-mul-degree*:
degree([*a*,*1*] *%* ^*n *** p*) = (*if poly p* = *poly* [] *then 0 else degree p* + *n*)
⟨*proof*⟩

lemma (**in** *idom-char-0*) *order-degree*:
assumes *p0*: *poly p* ≠ *poly* []
shows *order a p* ≤ *degree p*
⟨*proof*⟩

Tidier versions of finiteness of roots.

lemma (**in** *idom-char-0*) *poly-roots-finite-set*: *poly p* ≠ *poly* [] ==> *finite* {*x*. *poly p x* = *0*}
⟨*proof*⟩

bound for polynomial.

lemma *poly-mono*: *abs*(*x*) ≤ *k* ==> *abs*(*poly p* (*x*::'*a*::{*linordered-idom*})) ≤ *poly* (*map abs p*) *k*
⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-Sing*: *poly* [*c*] *x* = *c* ⟨*proof*⟩

end

72 While-Combinator: A general "while" combinator

theory *While-Combinator*

imports *Main*
begin

We define the while combinator as the "mother of all tail recursive functions".

function (*tailrec*) *while* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
where
while-unfold[*simp del*]: *while b c s* = (*if b s then while b c* (*c s*) *else s*)
⟨*proof*⟩

declare *while-unfold*[*code*]

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

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*⟩

An application: computation of the *lfp* on finite sets via iteration.

theorem *lfp-conv-while*:
[| *mono f*; *finite U*; *f U* = *U* |] ==>
lfp f = *fst* (*while* (λ(*A*, *fA*). *A* ≠ *fA*) (λ(*A*, *fA*). (*fA*, *f fA*)) ({}, *f* {}))
⟨*proof*⟩

An example of using the *while* combinator.

Cannot use *set-eq-subset* because it leads to looping because the anti-symmetry simproc turns the subset relationship back into equality.

theorem *P* (*lfp* (λ*N*::*int set*. {*0*} ∪ {(*n* + *2*) *mod 6* | *n*. *n* ∈ *N*})) =
P {*0*, *4*, *2*}
⟨*proof*⟩

end

73 Order-Relation: Orders as Relations

theory *Order-Relation*
imports *Main*
begin

73.1 Orders on a set

definition *preorder-on A r* ≡ *refl-on A r* ∧ *trans r*

definition *partial-order-on A r* ≡ *preorder-on A r* ∧ *antisym r*

definition *linear-order-on A r* ≡ *partial-order-on A r* ∧ *total-on A r*

definition *strict-linear-order-on A r* ≡ *trans r* ∧ *irrefl r* ∧ *total-on A r*

definition *well-order-on A r* ≡ *linear-order-on A r* ∧ *wf*(*r* − *Id*)

lemmas *order-on-defs* =
preorder-on-def partial-order-on-def linear-order-on-def
strict-linear-order-on-def well-order-on-def

lemma *preorder-on-empty*[*simp*]: *preorder-on* {} {}
⟨*proof*⟩

lemma *partial-order-on-empty*[*simp*]: *partial-order-on* {} {}
⟨*proof*⟩

lemma *lnear-order-on-empty*[*simp*]: *linear-order-on* {} {}
⟨*proof*⟩

lemma *well-order-on-empty*[*simp*]: *well-order-on* {} {}
⟨*proof*⟩

lemma *preorder-on-converse*[*simp*]: *preorder-on A* (*r*^−*1*) = *preorder-on A r*
⟨*proof*⟩

lemma *partial-order-on-converse*[*simp*]:
partial-order-on A (*r*^−*1*) = *partial-order-on A r*
⟨*proof*⟩

lemma *linear-order-on-converse*[*simp*]:
linear-order-on A (*r*^−*1*) = *linear-order-on A r*
⟨*proof*⟩

lemma *strict-linear-order-on-diff-Id*:
linear-order-on A r ⟹ *strict-linear-order-on A* (*r*−*Id*)

⟨*proof*⟩

73.2 Orders on the field

abbreviation *Refl r* ≡ *refl-on* (*Field r*) *r*

abbreviation *Preorder r* ≡ *preorder-on* (*Field r*) *r*

abbreviation *Partial-order r* ≡ *partial-order-on* (*Field r*) *r*

abbreviation *Total r* ≡ *total-on* (*Field r*) *r*

abbreviation *Linear-order r* ≡ *linear-order-on* (*Field r*) *r*

abbreviation *Well-order r* ≡ *well-order-on* (*Field r*) *r*

lemma *subset-Image-Image-iff*:
⟦ *Preorder r*; *A* ⊆ *Field r*; *B* ⊆ *Field r*⟧ ⟹
r `` *A* ⊆ *r* `` *B* ⟷ (∀ *a*∈*A*.∃ *b*∈*B*. (*b*,*a*):*r*)
⟨*proof*⟩

lemma *subset-Image1-Image1-iff*:
⟦ *Preorder r*; *a* : *Field r*; *b* : *Field r*⟧ ⟹ *r* `` {*a*} ⊆ *r* `` {*b*} ⟷ (*b*,*a*):*r*
⟨*proof*⟩

lemma *Refl-antisym-eq-Image1-Image1-iff*:
⟦*Refl r*; *antisym r*; *a*:*Field r*; *b*:*Field r*⟧ ⟹ *r* `` {*a*} = *r* `` {*b*} ⟷ *a*=*b*
⟨*proof*⟩

lemma *Partial-order-eq-Image1-Image1-iff*:
⟦*Partial-order r*; *a*:*Field r*; *b*:*Field r*⟧ ⟹ *r* `` {*a*} = *r* `` {*b*} ⟷ *a*=*b*
⟨*proof*⟩

73.3 Orders on a type

abbreviation *strict-linear-order* ≡ *strict-linear-order-on UNIV*

abbreviation *linear-order* ≡ *linear-order-on UNIV*

abbreviation *well-order r* ≡ *well-order-on UNIV*

end

74 Zorn: Zorn's Lemma

theory *Zorn*
imports *Order-Relation Main*

begin

definition *chain-subset* :: *'a set set ⇒ bool* (*chain⊆*) **where**
chain⊆ C ≡ ∀A∈C.∀B∈C. A ⊆ B ∨ B ⊆ A

The lemma and section numbers refer to an unpublished article [1].

definition
chain :: *'a set set => 'a set set set* **where**
chain S = {F. F ⊆ S & chain⊆ F}

definition
super :: *['a set set,'a set set] => 'a set set set* **where**
super S c = {d. d ∈ chain S & c ⊂ d}

definition
maxchain :: *'a set set => 'a set set set* **where**
maxchain S = {c. c ∈ chain S & super S c = {}}

definition
succ :: *['a set set,'a set set] => 'a set set* **where**
succ S c =
(if c ∉ chain S | c ∈ maxchain S
then c else SOME c'. c' ∈ super S c)

inductive-set
TFin :: 'a set set => 'a set set set
for *S :: 'a set set*
where
succI: *x ∈ TFin S ==> succ S x ∈ TFin S*
| *Pow-UnionI:* *Y ∈ Pow(TFin S) ==> Union(Y) ∈ TFin S*

74.1 Mathematical Preamble

lemma *Union-lemma0*:
(∀x ∈ C. x ⊆ A | B ⊆ x) ==> Union(C) ⊆ A | B ⊆ Union(C)
⟨*proof*⟩

This is theorem *increasingD2* of ZF/Zorn.thy

lemma *Abrial-axiom1*: *x ⊆ succ S x*
⟨*proof*⟩

lemmas *TFin-UnionI = TFin.Pow-UnionI* [*OF PowI*]

lemma *TFin-induct*:
assumes *H*: *n ∈ TFin S*
and *I*: *!!x. x ∈ TFin S ==> P x ==> P (succ S x)*
!!Y. Y ⊆ TFin S ==> Ball Y P ==> P(Union Y)
shows *P n* ⟨*proof*⟩

lemma *succ-trans*: $x \subseteq y \Longrightarrow x \subseteq succ\ S\ y$
⟨*proof*⟩

Lemma 1 of section 3.1

lemma *TFin-linear-lemma1*:
$[\!|\ n \in TFin\ S;\ \ m \in TFin\ S;$
$\forall x \in TFin\ S.\ x \subseteq m \longrightarrow x = m \mid succ\ S\ x \subseteq m$
$|\!] \Longrightarrow n \subseteq m \mid succ\ S\ m \subseteq n$
⟨*proof*⟩

Lemma 2 of section 3.2

lemma *TFin-linear-lemma2*:
$m \in TFin\ S \Longrightarrow \forall n \in TFin\ S.\ n \subseteq m \longrightarrow n{=}m \mid succ\ S\ n \subseteq m$
⟨*proof*⟩

Re-ordering the premises of Lemma 2

lemma *TFin-subsetD*:
$[\!|\ n \subseteq m;\ \ m \in TFin\ S;\ \ n \in TFin\ S\ |\!] \Longrightarrow n{=}m \mid succ\ S\ n \subseteq m$
⟨*proof*⟩

Consequences from section 3.3 – Property 3.2, the ordering is total

lemma *TFin-subset-linear*: $[\!|\ m \in TFin\ S;\ \ n \in TFin\ S|\!] \Longrightarrow n \subseteq m \mid m \subseteq n$
⟨*proof*⟩

Lemma 3 of section 3.3

lemma *eq-succ-upper*: $[\!|\ n \in TFin\ S;\ \ m \in TFin\ S;\ \ m = succ\ S\ m\ |\!] \Longrightarrow n \subseteq m$
⟨*proof*⟩

Property 3.3 of section 3.3

lemma *equal-succ-Union*: $m \in TFin\ S \Longrightarrow (m = succ\ S\ m) = (m = Union(TFin\ S))$
⟨*proof*⟩

74.2 Hausdorff's Theorem: Every Set Contains a Maximal Chain.

NB: We assume the partial ordering is $\subseteq$, the subset relation!

lemma *empty-set-mem-chain*: $(\{\} :: 'a\ set\ set) \in chain\ S$
⟨*proof*⟩

lemma *super-subset-chain*: $super\ S\ c \subseteq chain\ S$
⟨*proof*⟩

lemma *maxchain-subset-chain*: $maxchain\ S \subseteq chain\ S$
⟨*proof*⟩

lemma *mem-super-Ex*: $c \in chain\ S - maxchain\ S \Longrightarrow EX\ d.\ d \in super\ S\ c$

⟨*proof*⟩

lemma *select-super*:
$c \in chain\ S - maxchain\ S \Longrightarrow (\epsilon\, c'.\ c' \colon super\ S\ c) \colon super\ S\ c$
⟨*proof*⟩

lemma *select-not-equals*:
$c \in chain\ S - maxchain\ S \Longrightarrow (\epsilon\, c'.\ c' \colon super\ S\ c) \neq c$
⟨*proof*⟩

lemma *succI3*: $c \in chain\ S - maxchain\ S \Longrightarrow succ\ S\ c = (\epsilon\, c'.\ c' \colon super\ S\ c)$
⟨*proof*⟩

lemma *succ-not-equals*: $c \in chain\ S - maxchain\ S \Longrightarrow succ\ S\ c \neq c$
⟨*proof*⟩

lemma *TFin-chain-lemma4*: $c \in TFin\ S \Longrightarrow (c :: 'a\ set\ set) \colon chain\ S$
⟨*proof*⟩

theorem *Hausdorff*: $\exists\, c.\ (c :: 'a\ set\ set) \colon maxchain\ S$
⟨*proof*⟩

74.3 Zorn's Lemma: If All Chains Have Upper Bounds Then There Is a Maximal Element

lemma *chain-extend*:
$[\!|\ c \in chain\ S;\ z \in S;\ \forall x \in c.\ x \subseteq (z :: 'a\ set)\ |\!] \Longrightarrow \{z\}\ Un\ c \in chain\ S$
⟨*proof*⟩

lemma *chain-Union-upper*: $[\!|\ c \in chain\ S;\ x \in c\ |\!] \Longrightarrow x \subseteq Union(c)$
⟨*proof*⟩

lemma *chain-ball-Union-upper*: $c \in chain\ S \Longrightarrow \forall x \in c.\ x \subseteq Union(c)$
⟨*proof*⟩

lemma *maxchain-Zorn*:
$[\!|\ c \in maxchain\ S;\ u \in S;\ Union(c) \subseteq u\ |\!] \Longrightarrow Union(c) = u$
⟨*proof*⟩

theorem *Zorn-Lemma*:
$\forall c \in chain\ S.\ Union(c) \colon S \Longrightarrow \exists y \in S.\ \forall z \in S.\ y \subseteq z \longrightarrow y = z$
⟨*proof*⟩

74.4 Alternative version of Zorn's Lemma

lemma *Zorn-Lemma2*:
$\forall c \in chain\ S.\ \exists y \in S.\ \forall x \in c.\ x \subseteq y$
$\Longrightarrow \exists y \in S.\ \forall x \in S.\ (y :: 'a\ set) \subseteq x \longrightarrow y = x$
⟨*proof*⟩

Various other lemmas

lemma *chainD*: $[|\ c \in chain\ S;\ x \in c;\ y \in c\ |] ==> x \subseteq y \mid y \subseteq x$
⟨*proof*⟩

lemma *chainD2*: $!!(c :: 'a\ set\ set).\ c \in chain\ S ==> c \subseteq S$
⟨*proof*⟩

definition *Chain* :: $('a*'a)set \Rightarrow 'a\ set\ set$ **where**
$Chain\ r \equiv \{A.\ \forall a{\in}A.\forall b{\in}A.\ (a,b) : r \vee (b,a) \in r\}$

lemma *mono-Chain*: $r \subseteq s \Longrightarrow Chain\ r \subseteq Chain\ s$
⟨*proof*⟩

Zorn's lemma for partial orders:

lemma *Zorns-po-lemma*:
assumes *po*: $Partial\text{-}order\ r$ **and** *u*: $\forall C{\in}Chain\ r.\ \exists u{\in}Field\ r.\ \forall a{\in}C.\ (a,u):r$
shows $\exists m{\in}Field\ r.\ \forall a{\in}Field\ r.\ (m,a):r \longrightarrow a=m$
⟨*proof*⟩

definition *init-seg-of* :: $((('a*'a)set * ('a*'a)set)set$ **where**
$init\text{-}seg\text{-}of == \{(r,s).\ r \subseteq s \wedge (\forall a\ b\ c.\ (a,b):s \wedge (b,c):r \longrightarrow (a,b):r)\}$

abbreviation *initialSegmentOf* :: $('a*'a)set \Rightarrow ('a*'a)set \Rightarrow bool$
(**infix** *initial'-segment'-of* 55) **where**
$r\ initial\text{-}segment\text{-}of\ s == (r,s):init\text{-}seg\text{-}of$

lemma *refl-on-init-seg-of*[*simp*]: $r\ initial\text{-}segment\text{-}of\ r$
⟨*proof*⟩

lemma *trans-init-seg-of*:
$r\ initial\text{-}segment\text{-}of\ s \Longrightarrow s\ initial\text{-}segment\text{-}of\ t \Longrightarrow r\ initial\text{-}segment\text{-}of\ t$
⟨*proof*⟩

lemma *antisym-init-seg-of*:
$r\ initial\text{-}segment\text{-}of\ s \Longrightarrow s\ initial\text{-}segment\text{-}of\ r \Longrightarrow r=s$
⟨*proof*⟩

lemma *Chain-init-seg-of-Union*:
$R \in Chain\ init\text{-}seg\text{-}of \Longrightarrow r{\in}R \Longrightarrow r\ initial\text{-}segment\text{-}of\ \bigcup R$
⟨*proof*⟩

lemma *chain-subset-trans-Union*:
$chain_{\subseteq}\ R \Longrightarrow \forall r{\in}R.\ trans\ r \Longrightarrow trans(\bigcup R)$
⟨*proof*⟩

lemma *chain-subset-antisym-Union*:

$chain_{\subseteq}\ R \Longrightarrow \forall\, r{\in}R.\ antisym\ r \Longrightarrow antisym(\bigcup R)$
⟨*proof*⟩

lemma *chain-subset-Total-Union*:
assumes $chain_{\subseteq}\ R\ \forall\, r{\in}R.\ Total\ r$
shows $Total\ (\bigcup R)$
⟨*proof*⟩

lemma *wf-Union-wf-init-segs*:
assumes $R \in Chain\ init\text{-}seg\text{-}of$ **and** $\forall\, r{\in}R.\ wf\ r$ **shows** $wf(\bigcup R)$
⟨*proof*⟩

lemma *initial-segment-of-Diff*:
$p\ initial\text{-}segment\text{-}of\ q \Longrightarrow p - s\ initial\text{-}segment\text{-}of\ q - s$
⟨*proof*⟩

lemma *Chain-inits-DiffI*:
$R \in Chain\ init\text{-}seg\text{-}of \Longrightarrow \{r - s\ |r.\ r \in R\} \in Chain\ init\text{-}seg\text{-}of$
⟨*proof*⟩

theorem *well-ordering*: $\exists\, r{::}('a*'a)set.\ Well\text{-}order\ r \wedge Field\ r = UNIV$
⟨*proof*⟩

corollary *well-order-on*: $\exists\, r{::}('a*'a)set.\ well\text{-}order\text{-}on\ A\ r$
⟨*proof*⟩

end

75 List-Prefix: List prefixes and postfixes

theory *List-Prefix*
imports *List Main*
begin

75.1 Prefix order on lists

instantiation *list* :: (*type*) *order*
begin

definition
$prefix\text{-}def\ [code\ del]{:}\ xs \le ys = (\exists\, zs.\ ys = xs\ @\ zs)$

definition
$strict\text{-}prefix\text{-}def\ [code\ del]{:}\ xs < ys = (xs \le ys \wedge xs \neq (ys{::}'a\ list))$

instance
⟨*proof*⟩

end

lemma *prefixI* [*intro?*]: $ys = xs \mathbin{@} zs \Longrightarrow xs \leq ys$
⟨*proof*⟩

lemma *prefixE* [*elim?*]:
assumes $xs \leq ys$
obtains *zs* **where** $ys = xs \mathbin{@} zs$
⟨*proof*⟩

lemma *strict-prefixI'* [*intro?*]: $ys = xs \mathbin{@} z \# zs \Longrightarrow xs < ys$
⟨*proof*⟩

lemma *strict-prefixE'* [*elim?*]:
assumes $xs < ys$
obtains *z zs* **where** $ys = xs \mathbin{@} z \# zs$
⟨*proof*⟩

lemma *strict-prefixI* [*intro?*]: $xs \leq ys \Longrightarrow xs \neq ys \Longrightarrow xs < (ys::'a\ list)$
⟨*proof*⟩

lemma *strict-prefixE* [*elim?*]:
fixes *xs ys* :: *'a list*
assumes $xs < ys$
obtains $xs \leq ys$ **and** $xs \neq ys$
⟨*proof*⟩

75.2 Basic properties of prefixes

theorem *Nil-prefix* [*iff*]: $[] \leq xs$
⟨*proof*⟩

theorem *prefix-Nil* [*simp*]: $(xs \leq []) = (xs = [])$
⟨*proof*⟩

lemma *prefix-snoc* [*simp*]: $(xs \leq ys \mathbin{@} [y]) = (xs = ys \mathbin{@} [y] \vee xs \leq ys)$
⟨*proof*⟩

lemma *Cons-prefix-Cons* [*simp*]: $(x \# xs \leq y \# ys) = (x = y \wedge xs \leq ys)$
⟨*proof*⟩

lemma *same-prefix-prefix* [*simp*]: $(xs \mathbin{@} ys \leq xs \mathbin{@} zs) = (ys \leq zs)$
⟨*proof*⟩

lemma *same-prefix-nil* [*iff*]: $(xs \mathbin{@} ys \leq xs) = (ys = [])$
⟨*proof*⟩

lemma *prefix-prefix* [*simp*]: $xs \leq ys \Longrightarrow xs \leq ys \mathbin{@} zs$
⟨*proof*⟩

lemma *append-prefixD*: $xs \mathbin{@} ys \leq zs \Longrightarrow xs \leq zs$
⟨*proof*⟩

theorem *prefix-Cons*: $(xs \leq y \# ys) = (xs = [] \vee (\exists zs.\ xs = y \# zs \wedge zs \leq ys))$
⟨*proof*⟩

theorem *prefix-append*:
$(xs \leq ys \mathbin{@} zs) = (xs \leq ys \vee (\exists us.\ xs = ys \mathbin{@} us \wedge us \leq zs))$
⟨*proof*⟩

lemma *append-one-prefix*:
$xs \leq ys \Longrightarrow length\ xs < length\ ys \Longrightarrow xs \mathbin{@} [ys\ !\ length\ xs] \leq ys$
⟨*proof*⟩

theorem *prefix-length-le*: $xs \leq ys \Longrightarrow length\ xs \leq length\ ys$
⟨*proof*⟩

lemma *prefix-same-cases*:
$(xs_1 :: 'a\ list) \leq ys \Longrightarrow xs_2 \leq ys \Longrightarrow xs_1 \leq xs_2 \vee xs_2 \leq xs_1$
⟨*proof*⟩

lemma *set-mono-prefix*: $xs \leq ys \Longrightarrow set\ xs \subseteq set\ ys$
⟨*proof*⟩

lemma *take-is-prefix*: $take\ n\ xs \leq xs$
⟨*proof*⟩

lemma *map-prefixI*: $xs \leq ys \Longrightarrow map\ f\ xs \leq map\ f\ ys$
⟨*proof*⟩

lemma *prefix-length-less*: $xs < ys \Longrightarrow length\ xs < length\ ys$
⟨*proof*⟩

lemma *strict-prefix-simps* [*simp*]:
$xs < [] = False$
$[] < (x \# xs) = True$
$(x \# xs) < (y \# ys) = (x = y \wedge xs < ys)$
⟨*proof*⟩

lemma *take-strict-prefix*: $xs < ys \Longrightarrow take\ n\ xs < ys$
⟨*proof*⟩

lemma *not-prefix-cases*:
assumes *pfx*: $\neg\ ps \leq ls$
obtains
(*c1*) $ps \neq []$ **and** $ls = []$
| (*c2*) *a as x xs* **where** $ps = a \# as$ **and** $ls = x \# xs$ **and** $x = a$ **and** $\neg\ as \leq xs$
| (*c3*) *a as x xs* **where** $ps = a \# as$ **and** $ls = x \# xs$ **and** $x \neq a$

⟨*proof*⟩

lemma *not-prefix-induct* [*consumes 1*, *case-names Nil Neq Eq*]:
assumes *np*: $\neg$ *ps* $\leq$ *ls*
and *base*: $\bigwedge$*x xs*. *P* (*x*#*xs*) []
and *r1*: $\bigwedge$*x xs y ys*. *x* $\neq$ *y* $\Longrightarrow$ *P* (*x*#*xs*) (*y*#*ys*)
and *r2*: $\bigwedge$*x xs y ys*. ⟦ *x* = *y*; $\neg$ *xs* $\leq$ *ys*; *P xs ys* ⟧ $\Longrightarrow$ *P* (*x*#*xs*) (*y*#*ys*)
shows *P ps ls* ⟨*proof*⟩

75.3 Parallel lists

definition
parallel :: *'a list* => *'a list* => *bool* (**infixl** ∥ *50*) **where**
(*xs* ∥ *ys*) = ($\neg$ *xs* $\leq$ *ys* $\wedge$ $\neg$ *ys* $\leq$ *xs*)

lemma *parallelI* [*intro*]: $\neg$ *xs* $\leq$ *ys* ==> $\neg$ *ys* $\leq$ *xs* ==> *xs* ∥ *ys*
⟨*proof*⟩

lemma *parallelE* [*elim*]:
assumes *xs* ∥ *ys*
obtains $\neg$ *xs* $\leq$ *ys* $\wedge$ $\neg$ *ys* $\leq$ *xs*
⟨*proof*⟩

theorem *prefix-cases*:
obtains *xs* $\leq$ *ys* | *ys* < *xs* | *xs* ∥ *ys*
⟨*proof*⟩

theorem *parallel-decomp*:
xs ∥ *ys* ==> ∃ *as b bs c cs*. *b* $\neq$ *c* $\wedge$ *xs* = *as* @ *b* # *bs* $\wedge$ *ys* = *as* @ *c* # *cs*
⟨*proof*⟩

lemma *parallel-append*: *a* ∥ *b* $\Longrightarrow$ *a* @ *c* ∥ *b* @ *d*
⟨*proof*⟩

lemma *parallel-appendI*: *xs* ∥ *ys* $\Longrightarrow$ *x* = *xs* @ *xs'* $\Longrightarrow$ *y* = *ys* @ *ys'* $\Longrightarrow$ *x* ∥ *y*
⟨*proof*⟩

lemma *parallel-commute*: *a* ∥ *b* ⟷ *b* ∥ *a*
⟨*proof*⟩

75.4 Postfix order on lists

definition
postfix :: *'a list* => *'a list* => *bool* ((-/ >>= -) [*51*, *50*] *50*) **where**
(*xs* >>= *ys*) = (∃ *zs*. *xs* = *zs* @ *ys*)

lemma *postfixI* [*intro?*]: *xs* = *zs* @ *ys* ==> *xs* >>= *ys*
⟨*proof*⟩

lemma *postfixE* [*elim?*]:

assumes *xs >>= ys*
obtains *zs* **where** *xs = zs @ ys*
⟨*proof*⟩

lemma *postfix-refl* [*iff*]: *xs >>= xs*
⟨*proof*⟩
lemma *postfix-trans*: ⟦*xs >>= ys*; *ys >>= zs*⟧ ⟹ *xs >>= zs*
⟨*proof*⟩
lemma *postfix-antisym*: ⟦*xs >>= ys*; *ys >>= xs*⟧ ⟹ *xs = ys*
⟨*proof*⟩

lemma *Nil-postfix* [*iff*]: *xs >>= []*
⟨*proof*⟩
lemma *postfix-Nil* [*simp*]: *([] >>= xs) = (xs = [])*
⟨*proof*⟩

lemma *postfix-ConsI*: *xs >>= ys* ⟹ *x#xs >>= ys*
⟨*proof*⟩
lemma *postfix-ConsD*: *xs >>= y#ys* ⟹ *xs >>= ys*
⟨*proof*⟩

lemma *postfix-appendI*: *xs >>= ys* ⟹ *zs @ xs >>= ys*
⟨*proof*⟩
lemma *postfix-appendD*: *xs >>= zs @ ys* ⟹ *xs >>= ys*
⟨*proof*⟩

lemma *postfix-is-subset*: *xs >>= ys ==> set ys ⊆ set xs*
⟨*proof*⟩

lemma *postfix-ConsD2*: *x#xs >>= y#ys ==> xs >>= ys*
⟨*proof*⟩

lemma *postfix-to-prefix*: *xs >>= ys* ⟷ *rev ys ≤ rev xs*
⟨*proof*⟩

lemma *distinct-postfix*: *distinct xs* ⟹ *xs >>= ys* ⟹ *distinct ys*
⟨*proof*⟩

lemma *postfix-map*: *xs >>= ys* ⟹ *map f xs >>= map f ys*
⟨*proof*⟩

lemma *postfix-drop*: *as >>= drop n as*
⟨*proof*⟩

lemma *postfix-take*: *xs >>= ys* ⟹ *xs = take (length xs − length ys) xs @ ys*
⟨*proof*⟩

lemma *parallelD1*: *x ∥ y* ⟹ ¬ *x ≤ y*
⟨*proof*⟩

lemma *parallelD2*: $x \parallel y \Longrightarrow \neg\ y \leq x$
⟨*proof*⟩

lemma *parallel-Nil1* [*simp*]: $\neg\ x \parallel []$
⟨*proof*⟩

lemma *parallel-Nil2* [*simp*]: $\neg\ [] \parallel x$
⟨*proof*⟩

lemma *Cons-parallelI1*: $a \neq b \Longrightarrow a \# as \parallel b \# bs$
⟨*proof*⟩

lemma *Cons-parallelI2*: $[\![\ a = b;\ as \parallel bs\ ]\!] \Longrightarrow a \# as \parallel b \# bs$
⟨*proof*⟩

lemma *not-equal-is-parallel*:
assumes *neq*: $xs \neq ys$
and *len*: $length\ xs = length\ ys$
shows $xs \parallel ys$
⟨*proof*⟩

75.5 Executable code

lemma *less-eq-code* [*code*]:
$([]::'a::\{eq,\ ord\}\ list) \leq xs \longleftrightarrow True$
$(x::'a::\{eq,\ ord\})\ \#\ xs \leq [] \longleftrightarrow False$
$(x::'a::\{eq,\ ord\})\ \#\ xs \leq y\ \#\ ys \longleftrightarrow x = y \wedge xs \leq ys$
⟨*proof*⟩

lemma *less-code* [*code*]:
$xs < ([]::'a::\{eq,\ ord\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{eq,\ ord\})\#\ xs \longleftrightarrow True$
$(x::'a::\{eq,\ ord\})\ \#\ xs < y\ \#\ ys \longleftrightarrow x = y \wedge xs < ys$
⟨*proof*⟩

lemmas [*code*] = *postfix-to-prefix*

end

76 List-lexord: Lexicographic order on lists

theory *List-lexord*
imports *List Main*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def [*code del*]: (*xs*::('*a*::*ord*) *list*) < *ys* ⟷ (*xs*, *ys*) ∈ *lexord* {(*u*,*v*). *u* < *v*}

definition
list-le-def [*code del*]: (*xs*::('*a*::*ord*) *list*) ≤ *ys* ⟷ (*xs* < *ys* ∨ *xs* = *ys*)

instance ⟨*proof*⟩

end

instance *list* :: (*order*) *order*
⟨*proof*⟩

instance *list* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition
[*code del*]: (*inf* :: '*a list* ⇒ -) = *min*

definition
[*code del*]: (*sup* :: '*a list* ⇒ -) = *max*

instance
⟨*proof*⟩

end

lemma *not-less-Nil* [*simp*]: ¬ (*x* < [])
⟨*proof*⟩

lemma *Nil-less-Cons* [*simp*]: [] < *a* # *x*
⟨*proof*⟩

lemma *Cons-less-Cons* [*simp*]: *a* # *x* < *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* < *y*
⟨*proof*⟩

lemma *le-Nil* [*simp*]: *x* ≤ [] ⟷ *x* = []
⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: [] ≤ *x*
⟨*proof*⟩

lemma *Cons-le-Cons* [*simp*]: *a* # *x* ≤ *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* ≤ *y*
⟨*proof*⟩

lemma *less-code* [*code*]:
$xs < ([]::'a::\{eq, order\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{eq, order\}) \# xs \longleftrightarrow True$
$(x::'a::\{eq, order\}) \# xs < y \# ys \longleftrightarrow x < y \vee x = y \wedge xs < ys$
⟨*proof*⟩

lemma *less-eq-code* [*code*]:
$x \# xs \leq ([]::'a::\{eq, order\}\ list) \longleftrightarrow False$
$[] \leq (xs::'a::\{eq, order\}\ list) \longleftrightarrow True$
$(x::'a::\{eq, order\}) \# xs \leq y \# ys \longleftrightarrow x < y \vee x = y \wedge xs \leq ys$
⟨*proof*⟩

end

77 Sublist-Order: Sublist Ordering

theory *Sublist-Order*
imports *Main*
begin

This theory defines sublist ordering on lists. A list *ys* is a sublist of a list *xs*, iff one obtains *ys* by erasing some elements from *xs*.

77.1 Definitions and basic lemmas

instantiation *list* :: (*type*) *ord*
begin

inductive *less-eq-list* **where**
empty [*simp*, *intro*!]: $[] \leq xs$
| *drop*: $ys \leq xs \Longrightarrow ys \leq x \# xs$
| *take*: $ys \leq xs \Longrightarrow x \# ys \leq x \# xs$

definition
[*code del*]: $(xs :: 'a\ list) < ys \longleftrightarrow xs \leq ys \wedge \neg\ ys \leq xs$

instance ⟨*proof*⟩

end

lemma *le-list-length*: $xs \leq ys \Longrightarrow length\ xs \leq length\ ys$
⟨*proof*⟩

lemma *le-list-same-length*: $xs \leq ys \Longrightarrow length\ xs = length\ ys \Longrightarrow xs = ys$
⟨*proof*⟩

lemma *not-le-list-length*[*simp*]: $length\ ys < length\ xs \Longrightarrow \sim\ xs <= ys$

⟨*proof*⟩

lemma *le-list-below-empty* [*simp*]: $xs \leq [] \longleftrightarrow xs = []$
⟨*proof*⟩

lemma *le-list-drop-many*: $xs \leq ys \Longrightarrow xs \leq zs \mathbin{@} ys$
⟨*proof*⟩

lemma [*code*]: $[] <= xs \longleftrightarrow True$
⟨*proof*⟩

lemma [*code*]: $(x\#xs) <= [] \longleftrightarrow False$
⟨*proof*⟩

lemma *le-list-drop-Cons*: **assumes** $x\#xs <= ys$ **shows** $xs <= ys$
⟨*proof*⟩

lemma *le-list-drop-Cons2*:
assumes $x\#xs <= x\#ys$ **shows** $xs <= ys$
⟨*proof*⟩

lemma *le-list-drop-Cons-neq*: **assumes** $x \# xs <= y \# ys$
shows $x \mathbin{\tilde{}}= y \Longrightarrow x \# xs <= ys$
⟨*proof*⟩

lemma *le-list-Cons2-iff* [*simp*,*code*]: $(x\#xs) <= (y\#ys) \longleftrightarrow$
$(if\ x=y\ then\ xs <= ys\ else\ (x\#xs) <= ys)$
⟨*proof*⟩

lemma *le-list-take-many-iff*: $zs \mathbin{@} xs \leq zs \mathbin{@} ys \longleftrightarrow xs \leq ys$
⟨*proof*⟩

lemma *le-list-Cons-EX*:
assumes $x \# ys <= zs$ **shows** $EX\ us\ vs.\ zs = us \mathbin{@} x \# vs \mathbin{\&} ys <= vs$
⟨*proof*⟩

instantiation *list* :: (*type*) *order*
begin

instance ⟨*proof*⟩

end

lemma *le-list-append-le-same-iff*: $xs \mathbin{@} ys <= ys \longleftrightarrow xs=[]$
⟨*proof*⟩

lemma *le-list-append-mono*: $[\![\ xs <= xs';\ ys <= ys'\ ]\!] \Longrightarrow xs@ys <= xs'@ys'$
⟨*proof*⟩

lemma *less-list-length*: $xs < ys \Longrightarrow length\ xs < length\ ys$
⟨*proof*⟩

lemma *less-list-empty* [*simp*]: $[] < xs \longleftrightarrow xs \neq []$
⟨*proof*⟩

lemma *less-list-below-empty*[*simp*]: $xs < [] \longleftrightarrow False$
⟨*proof*⟩

lemma *less-list-drop*: $xs < ys \Longrightarrow xs < x \# ys$
⟨*proof*⟩

lemma *less-list-take-iff*: $x \# xs < x \# ys \longleftrightarrow xs < ys$
⟨*proof*⟩

lemma *less-list-drop-many*: $xs < ys \Longrightarrow xs < zs\ @\ ys$
⟨*proof*⟩

lemma *less-list-take-many-iff*: $zs\ @\ xs < zs\ @\ ys \longleftrightarrow xs < ys$
⟨*proof*⟩

77.2 Appending elements

lemma *le-list-rev-take-iff*[*simp*]: $xs\ @\ zs \leq ys\ @\ zs \longleftrightarrow xs \leq ys$ (**is** $?L = ?R$)
⟨*proof*⟩

lemma *less-list-rev-take*: $xs\ @\ zs < ys\ @\ zs \longleftrightarrow xs < ys$
⟨*proof*⟩

lemma *le-list-rev-drop-many*: $xs \leq ys \Longrightarrow xs \leq ys\ @\ zs$
⟨*proof*⟩

77.3 Relation to standard list operations

lemma *le-list-map*: $xs \leq ys \Longrightarrow map\ f\ xs \leq map\ f\ ys$
⟨*proof*⟩

lemma *le-list-filter-left*[*simp*]: $filter\ f\ xs \leq xs$
⟨*proof*⟩

lemma *le-list-filter*: $xs \leq ys \Longrightarrow filter\ f\ xs \leq filter\ f\ ys$
⟨*proof*⟩

lemma $xs \leq ys \longleftrightarrow (EX\ N.\ xs = sublist\ ys\ N)$ (**is** $?L = ?R$)
⟨*proof*⟩

end

References

[1] Abrial and Laffitte. Towards the mechanization of the proofs of some classical theorems of set theory. Unpublished.

[2] J. Avigad and K. Donnelly. Formalizing O notation in Isabelle/HOL. In D. Basin and M. Rusiowitch, editors, *Automated Reasoning: second international conference, IJCAR 2004*, pages 357–371. Springer, 2004.

[3] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.