The Supplemental Isabelle/HOL Library

April 19, 2009

Contents

1 Abstract-Rat: Abstract rational numbers

theory *Abstract-Rat*
imports *GCD 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 ∧ zgcd a b = 1*))

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

declare *zgcd-zdvd1*[*presburger*]
declare *zgcd-zdvd2*[*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 = zgcd* (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*::{*ring-char-0*,*field*, *division-by-zero*}) = *INum y*) = (*x* = *y*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

lemma *of-int-div-aux*: *d* ~= *0* ==> ((*of-int x*)::′*a*::{*field*, *ring-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*, *ring-char-0*}) = *of-int n* / *of-int d*
⟨*proof*⟩

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

lemma *INum-normNum-iff*: (*INum x* ::′*a*::{*field*, *division-by-zero*, *ring-char-0*}) = *INum y* ⟷ *normNum x* = *normNum y* (**is** *?lhs* = *?rhs*)

⟨*proof*⟩

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

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

⟨*proof*⟩

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

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

lemma *Ninv*[*simp*]: *INum* (*Ninv* x) = ($1::'a ::$ {*division-by-zero*,*field*}) / (*INum* x)
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *Nmul-assoc*:
assumes *SORT-CONSTRAINT*('*a*::{*ring-char-0*,*division-by-zero*,*field*})
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*::{*ring-char-0*,*division-by-zero*,*field*})
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*::{*ring-char-0*,*division-by-zero*,*field*})
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 AssocList: Map operations implemented on association lists

theory *AssocList*
imports *Map Main*
begin

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

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

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

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

primrec
merge :: (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where
merge qs [] = *qs*
| *merge qs* (*p*#*ps*) = *update* (*fst p*) (*snd p*) (*merge qs ps*)

lemma *length-delete-le*: *length* (*delete k al*) ≤ *length al*
⟨*proof*⟩

lemma *compose-hint* [*simp*]:
length (*delete k al*) < *Suc* (*length al*)
⟨*proof*⟩

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

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

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

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

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

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

2.1 *delete*

lemma *delete-eq*:
delete k xs = *filter* (λ*p*. *fst p* ≠ *k*) *xs*
⟨*proof*⟩

lemma *delete-id* [*simp*]: *k* ∉ *fst* ‘ *set al* ⟹ *delete k al* = *al*
⟨*proof*⟩

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

lemma *delete-conv'*: *map-of* (*delete k al*) = ((*map-of al*)(*k* := *None*))
⟨*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 *distinct-delete*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*delete k al*))
⟨*proof*⟩

lemma *delete-twist*: *delete x* (*delete y al*) = *delete y* (*delete x al*)
⟨*proof*⟩

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

2.2 *clearjunk*

lemma *insert-fst-filter*:
insert a(*fst* ‘ {*x* ∈ *set ps*. *fst x* ≠ *a*}) = *insert a* (*fst* ‘ *set ps*)
⟨*proof*⟩

lemma *dom-clearjunk*: *fst ‘ set* (*clearjunk al*) = *fst ‘ set al*
⟨*proof*⟩

lemma *notin-filter-fst*: *a* ∉ *fst ‘* {*x* ∈ *set ps*. *fst x* ≠ *a*}
⟨*proof*⟩

lemma *distinct-clearjunk* [*simp*]: *distinct* (*map fst* (*clearjunk al*))
⟨*proof*⟩

lemma *map-of-filter*: *k* ≠ *a* ⟹ *map-of* [*q*←*ps* . *fst q* ≠ *a*] *k* = *map-of ps k*
⟨*proof*⟩

lemma *map-of-clearjunk*: *map-of* (*clearjunk al*) = *map-of al*
⟨*proof*⟩

lemma *length-clearjunk*: *length* (*clearjunk al*) ≤ *length al*
⟨*proof*⟩

lemma *notin-fst-filter*: *a* ∉ *fst ‘ set ps* ⟹ [*q*←*ps* . *fst q* ≠ *a*] = *ps*
⟨*proof*⟩

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

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

2.3 *dom* and *ran*

lemma *dom-map-of′*: *fst ‘ set al* = *dom* (*map-of al*)
⟨*proof*⟩

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

lemma *ran-clearjunk*: *ran* (*map-of* (*clearjunk al*)) = *ran* (*map-of al*)
⟨*proof*⟩

lemma *ran-distinct*:
assumes *dist*: *distinct* (*map fst al*)
shows *ran* (*map-of al*) = *snd ‘ set al*
⟨*proof*⟩

lemma *ran-map-of*: *ran* (*map-of al*) = *snd ‘ set* (*clearjunk al*)
⟨*proof*⟩

2.4 *update*

lemma *update-conv*: *map-of* (*update k v al*) *k′* = ((*map-of al*)(*k*↦*v*)) *k′*
⟨*proof*⟩

lemma *update-conv'*: *map-of* (*update k v al*) = ((*map-of al*)(*k*↦*v*))
⟨*proof*⟩

lemma *dom-update*: *fst* ' *set* (*update k v al*) = {*k*} ∪ *fst* ' *set al*
⟨*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 *clearjunk-update*: *clearjunk* (*update k v al*) = *update k v* (*clearjunk al*)
⟨*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*⟩

2.5 *updates*

lemma *updates-conv*: *map-of* (*updates ks vs al*) *k* = ((*map-of al*)(*ks*[↦]*vs*)) *k*

⟨*proof*⟩

lemma *updates-conv'*: *map-of* (*updates ks vs al*) = ((*map-of al*)(*ks*[↦]*vs*))
⟨*proof*⟩

lemma *distinct-updates*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*updates ks vs al*))
⟨*proof*⟩

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

lemma *updates-empty*[*simp*]: *updates vs* [] *al* = *al*
⟨*proof*⟩

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

2.6 *map-ran*

lemma *map-ran-conv*: *map-of (map-ran f al) k = Option.map (f k) (map-of al k)*
⟨*proof*⟩

lemma *dom-map-ran*: *fst ‘ set (map-ran f al) = fst ‘ set al*
⟨*proof*⟩

lemma *distinct-map-ran*: *distinct (map fst al) ⟹ distinct (map fst (map-ran f al))*
⟨*proof*⟩

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

lemma *clearjunk-map-ran*: *clearjunk (map-ran f al) = map-ran f (clearjunk al)*
⟨*proof*⟩

2.7 *merge*

lemma *dom-merge*: *fst ‘ set (merge xs ys) = fst ‘ set xs ∪ 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) k = (map-of xs ++ map-of ys) k*
⟨*proof*⟩

lemma *merge-conv'*: *map-of (merge xs ys) = (map-of xs ++ map-of ys)*
⟨*proof*⟩

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

lemma *merge-find-right*[*simp*]: *map-of n k* = *Some v* ⟹ *map-of* (*merge m n*) *k* = *Some v*
⟨*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*⟩

2.8 *compose*

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*⟩

2.9 *restrict*

lemma *restrict-eq*:
restrict A = *filter* (λ*p*. *fst p* ∈ *A*)
⟨*proof*⟩

lemma *distinct-restr*: *distinct* (*map fst al*) ⟹ *distinct* (*map fst* (*restrict A al*))
⟨*proof*⟩

lemma *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k*
⟨*proof*⟩

lemma *restr-conv'*: *map-of* (*restrict A al*) = ((*map-of al*)|‘ *A*)
⟨*proof*⟩

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

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

lemma *restr-out* [*simp*]: $x \notin A \Longrightarrow$ *map-of* (*restrict A al*) x = *None*
⟨*proof*⟩

lemma *dom-restr* [*simp*]: *fst* ' *set* (*restrict A al*) = *fst* ' *set al* ∩ *A*
⟨*proof*⟩

lemma *restr-upd-same* [*simp*]: *restrict* (−{*x*}) (*update x y al*) = *restrict* (−{*x*}) *al*
⟨*proof*⟩

lemma *restr-restr* [*simp*]: *restrict A* (*restrict B al*) = *restrict* (*A*∩*B*) *al*
⟨*proof*⟩

lemma *restr-update*[*simp*]:
map-of (*restrict D* (*update x y al*)) =
map-of ((*if x* ∈ *D then* (*update x y* (*restrict* (*D*−{*x*}) *al*)) *else restrict D al*))
⟨*proof*⟩

lemma *restr-delete* [*simp*]:
(*delete x* (*restrict D al*)) =
(*if x*∈ *D then restrict* (*D* − {*x*}) *al else restrict D al*)
⟨*proof*⟩

lemma *update-restr*:
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D*−{*x*}) *al*))
⟨*proof*⟩

lemma *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*⟩

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

end

3 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*.

3.1 Basic definitions

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

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** $\otimes$ *70*) **where**
$A \otimes B == \{c.\ EX\ a{:}A.\ EX\ b{:}B.\ c = a * b\}$

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

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

3.2 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* ⊕ *E* <= *D* ⊕ *F*
⟨*proof*⟩

lemma *set-plus-mono3* [*intro*]: *a* : *C* ==> *a* +*o* *D* <= *C* ⊕ *D*

⟨*proof*⟩

lemma *set-plus-mono4* [*intro*]: (*a*::'*a*::*comm-monoid-add*) : *C* ==>
a +*o* *D* <= *D* ⊕ *C*
⟨*proof*⟩

lemma *set-plus-mono5*: *a*:*C* ==> *B* <= *D* ==> *a* +*o* *B* <= *C* ⊕ *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* ⊕ *E* ==>
x : *D* ⊕ *F*
⟨*proof*⟩

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

lemma *set-plus-mono4-b*: (*a*::'*a*::*comm-monoid-add*) : *C* ==>
x : *a* +*o* *D* ==> *x* : *D* ⊕ *C*
⟨*proof*⟩

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

lemma *set-zero-plus2*: (*0*::'*a*::*comm-monoid-add*) : *A* ==> *B* <= *A* ⊕ *B*
⟨*proof*⟩

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

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

lemma *set-minus-plus*: ((*a*::'*a*::*ab-group-add*) − *b* : *C*) = (*a* : *b* +*o* *C*)
⟨*proof*⟩

lemma *set-times-intro* [*intro*]: *a* : *C* ==> *b* : *D* ==> *a* ∗ *b* : *C* ⊗ *D*
⟨*proof*⟩

lemma *set-times-intro2* [*intro*!]: *b* : *C* ==> *a* ∗ *b* : *a* ∗*o* *C*
⟨*proof*⟩

lemma *set-times-rearrange*: ((*a*::'*a*::*comm-monoid-mult*) ∗*o* *C*) ⊗
(*b* ∗*o* *D*) = (*a* ∗ *b*) ∗*o* (*C* ⊗ *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*]: $C <= D ==> a *o C <= a *o D$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *set-times-mono4-b*: $(a::'a::comm\text{-}monoid\text{-}mult) : C ==> x : a *o D ==> x : D \otimes C$
⟨*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*: $(a::'a::ring\text{-}1) : (-1) *o C ==>$
$- a : C$
⟨*proof*⟩

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

end

4 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*].

4.1 Definitions

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

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

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

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

lemma *bigo-plus-subset2* [*intro*]: *A* <= *O*(*f*) ==> *B* <= *O*(*f*) ==> *A* ⊕ *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*: $ALL\ x.\ f\ x \sim= 0 ==>$

$O(f * g) <= (f::'a => ('b::ordered\text{-}field)) *o O(g)$
⟨*proof*⟩

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

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

lemma *bigo-mult8*: $ALL\ x.\ f\ x \sim= 0 ==> O(f * g) = O(f::'a => ('b::ordered\text{-}field)) \otimes O(g)$
⟨*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*: $(c::'a::ordered\text{-}field) \neq 0 \Longrightarrow (\%x.\ 1) : O(\%x.\ c)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

4.2 Setsum

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

⟨*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*⟩

4.3 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*: $(c::'a::ordered\text{-}field)\ \sim= 0 ==>$
$(\%x.\ c) * f =o\ O(h) ==> f =o\ O(h)$
⟨*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*⟩

4.4 Less than or equal to

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

lemma *bigo-lesseq1*: $f =o O(h) ==> ALL\ x.\ abs\ (g\ x) <= abs\ (f\ x) ==>$
$g =o O(h)$
⟨*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*: $f <o g =o O(k::'a=>'b::ordered\text{-}field) ==>$
$g =o h +o O(k) ==> f <o h =o O(k)$

⟨*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

5 Binomial: Binomial Coefficients

theory *Binomial*
imports *Fact SetInterval Presburger Main*
begin

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

primrec *binomial* :: $nat \Rightarrow nat \Rightarrow nat$ (**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*: $k \leq n$ ==> (*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*. $k \leq n$ ==> *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*:
$k \leq n$ ==> (*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*:
[|$k \leq n$; *0*<*k*|] ==> (*n choose k*) * *k* = *n* * ((*n* − *1*) *choose* (*k* − *1*))
⟨*proof*⟩

5.1 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* ==> $x \notin M$
==> {*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 $\Longrightarrow$ (!!*A*. *A*<=*B* $\Longrightarrow$ *finite*{*x*. *P x A*}) $\Longrightarrow$ *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*⟩

5.2 Pochhammer's symbol : generalized raising factorial

definition *pochhammer* (*a*::'*a*::*comm-semiring-1*) *n* = (*if n* = *0 then 1 else setprod*
(λ*n*. *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* (λ*n*. *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 *fact-setprod*: *fact n* = *setprod id* {*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*⟩

5.3 Generalized binomial coefficients

definition *gbinomial* :: 'a::{*field*, *recpower*,*ring-char-0*} ⇒ *nat* ⇒ 'a (**infixl** *gchoose* *65*)
where *a gchoose n* = (*if n* = *0 then 1 else* (*setprod* (λ*i*. *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* ≤ *n*
shows (*of-nat* (*n choose k*) :: 'a::{*field*, *ring-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-nat n * (a gchoose n) + of-nat (Suc n) * (a gchoose (Suc n))*
⟨*proof*⟩

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

lemma *gbinomial-mult-fact*:
*(of-nat (fact (Suc k)) :: 'a) * ((a::'a::{field, ring-char-0,recpower}) gchoose (Suc k)) = (setprod (λi. a − of-nat i) {0 .. k})*
⟨*proof*⟩

lemma *gbinomial-mult-fact'*:
*((a::'a::{field, ring-char-0,recpower}) gchoose (Suc k)) * (of-nat (fact (Suc k)) :: 'a) = (setprod (λi. a − of-nat i) {0 .. k})*
⟨*proof*⟩

lemma *gbinomial-Suc-Suc*: *((a::'a::{field,recpower, ring-char-0}) + 1) gchoose (Suc k) = a gchoose k + (a gchoose (Suc k))*
⟨*proof*⟩

end

6 Bit: The Field of Integers mod 2

theory *Bit*
imports *Main*
begin

6.1 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*⟩

6.2 Type *bit* forms a field

instantiation *bit* :: {*field*, *division-by-zero*}
begin

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

definition *times-bit-def*:
*x * y* = (*case x of 0* ⇒ *0* | *1* ⇒ *y*)

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*]: $x * y = (1 :: bit) \longleftrightarrow x = 1 \wedge y = 1$
⟨*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*⟩

6.3 Numerals at type *bit*

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

definition *number-of-bit-def*:
(*number-of w* :: *bit*) = *of-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-of* (*Int.Bit0 w*) = (*0* :: *bit*)
⟨*proof*⟩

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

end

7 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*]: $x \sqcup \sim x = \mathbf{1}$
begin

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

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

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

7.1 Complement

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

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*⟩

7.2 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

7.3 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

7.4 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

7.5 Symmetric Difference

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

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

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

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

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

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

8 Product-ord: Order on product types

theory *Product-ord*
imports *Main*
begin

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

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

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

instance ⟨*proof*⟩

end

lemma [*code*]:
(*x1*::'*a*::{*ord*, *eq*}, *y1*) ≤ (*x2*, *y2*) ⟷ *x1* < *x2* ∨ *x1* = *x2* ∧ *y1* ≤ *y2*
(*x1*::'*a*::{*ord*, *eq*}, *y1*) < (*x2*, *y2*) ⟷ *x1* < *x2* ∨ *x1* = *x2* ∧ *y1* < *y2*
⟨*proof*⟩

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

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

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

definition
inf-prod-def: (*inf* :: '*a* × '*b* ⇒ - ⇒ -) = *min*

definition
sup-prod-def: (*sup* :: '*a* × '*b* ⇒ - ⇒ -) = *max*

instance
⟨*proof*⟩

end

end

9 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*]

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 List

code-modulename *OCaml*
Char-nat List

code-modulename *Haskell*
Char-nat List

end

10 Char-ord: Order on characters

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

instantiation *nibble* :: *linorder*
begin

definition
nibble-less-eq-def: *n* ≤ *m* ⟷ *nat-of-nibble n* ≤ *nat-of-nibble m*

definition
nibble-less-def: *n* < *m* ⟷ *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

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

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

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

⟨*ML*⟩

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

code-reserved *SML*
char

code-reserved *OCaml*
char

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

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

end

12 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*)

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*)

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

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

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

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

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

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

code-const *pdivmod*
(*SML* (*fn k => fn l =>/ IntInf.divMod/* (*IntInf.abs k*,*/ IntInf.abs l*)))
(*OCaml* (*fun k -> fun l ->/ Big'-int.quomod'-big'-int/* (*Big'-int.abs'-big'-int k*)*/* (*Big'-int.abs'-big'-int l*)))
(*Haskell* (*k l ->/ divMod/* (*abs k*)*/* (*abs l*)))

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

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

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

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

Evaluation

lemma [*code*, *code del*]:
(*Code-Eval.term-of* :: *int* ⇒ *term*) = *Code-Eval.term-of* ⟨*proof*⟩

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

end

13 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*⟩

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

lemma [*code*, *code inline*]:
char-rec f c = *split f* (*nibble-pair-of-nat* (*nat-of-char c*))
⟨*proof*⟩

lemma [*code*, *code inline*]:
char-case f c = *split f* (*nibble-pair-of-nat* (*nat-of-char c*))
⟨*proof*⟩

lemma [*code*]:
size (*c*::*char*) = *0*
⟨*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* | *k* < *256* = *toEnum k* :: *Char in chr* . *fromInteger*))

end

14 Code-Index: Type of indices

theory *Code-Index*
imports *Main*
begin

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

14.1 Datatype of indices

typedef (**open**) *index* = *UNIV* :: *nat set*
morphisms *nat-of of-nat* ⟨*proof*⟩

lemma *of-nat-nat-of* [*simp*]:
of-nat (*nat-of k*) = *k*
⟨*proof*⟩

lemma *nat-of-of-nat* [*simp*]:

nat-of (*of-nat n*) = *n*
⟨*proof*⟩

lemma [*measure-function*]:
is-measure nat-of ⟨*proof*⟩

lemma *index*:
(⋀*n*::*index*. *PROP P n*) ≡ (⋀*n*::*nat*. *PROP P* (*of-nat n*))
⟨*proof*⟩

lemma *index-case*:
assumes ⋀*n*. *k* = *of-nat n* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *index-induct-raw*:
assumes ⋀*n*. *P* (*of-nat n*)
shows *P k*
⟨*proof*⟩

lemma *nat-of-inject* [*simp*]:
nat-of k = *nat-of l* ⟷ *k* = *l*
⟨*proof*⟩

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

instantiation *index* :: *zero*
begin

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

instance ⟨*proof*⟩

end

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

rep-datatype *0* :: *index Suc-index*
⟨*proof*⟩

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

lemma *index-decr* [*termination-simp*]:
k ≠ *Code-Index.of-nat 0* ⟹ *Code-Index.nat-of k* − *Suc 0* < *Code-Index.nat-of*

k
⟨*proof*⟩

lemma [*simp*, *code*]:
index-size = *nat-of*
⟨*proof*⟩

lemma [*simp*, *code*]:
size = *nat-of*
⟨*proof*⟩

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

lemma [*code*]:
eq-class.eq k l ⟷ *eq-class.eq* (*nat-of k*) (*nat-of l*)
⟨*proof*⟩

lemma [*code nbe*]:
eq-class.eq (*k*::*index*) *k* ⟷ *True*
⟨*proof*⟩

14.2 Indices as datatype of ints

instantiation *index* :: *number*
begin

definition
number-of = *of-nat o nat*

instance ⟨*proof*⟩

end

lemma *nat-of-number* [*simp*]:
nat-of (*number-of k*) = *number-of k*
⟨*proof*⟩

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

14.3 Basic arithmetic

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

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

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

definition [*simp*, *code del*]:
$n - m = \textit{of-nat}\ (\textit{nat-of}\ n - \textit{nat-of}\ m)$

definition [*simp*, *code del*]:
$n * m = \textit{of-nat}\ (\textit{nat-of}\ n * \textit{nat-of}\ m)$

definition [*simp*, *code del*]:
$n\ \textit{div}\ m = \textit{of-nat}\ (\textit{nat-of}\ n\ \textit{div}\ \textit{nat-of}\ m)$

definition [*simp*, *code del*]:
$n\ \textit{mod}\ m = \textit{of-nat}\ (\textit{nat-of}\ n\ \textit{mod}\ \textit{nat-of}\ m)$

definition [*simp*, *code del*]:
$n \leq m \longleftrightarrow \textit{nat-of}\ n \leq \textit{nat-of}\ m$

definition [*simp*, *code del*]:
$n < m \longleftrightarrow \textit{nat-of}\ n < \textit{nat-of}\ m$

instance ⟨*proof*⟩

end

lemma *zero-index-code* [*code inline*, *code*]:
$(0{::}\textit{index}) = \textit{Numeral0}$
⟨*proof*⟩
lemma [*code post*]: $\textit{Numeral0} = (0{::}\textit{index})$
⟨*proof*⟩

lemma *one-index-code* [*code inline*, *code*]:
$(1{::}\textit{index}) = \textit{Numeral1}$
⟨*proof*⟩
lemma [*code post*]: $\textit{Numeral1} = (1{::}\textit{index})$
⟨*proof*⟩

lemma *plus-index-code* [*code nbe*]:
$\textit{of-nat}\ n + \textit{of-nat}\ m = \textit{of-nat}\ (n + m)$
⟨*proof*⟩

definition *subtract-index* :: $\textit{index} \Rightarrow \textit{index} \Rightarrow \textit{index}$ **where**
[*simp*, *code del*]: $\textit{subtract-index} = \textit{op} -$

lemma *subtract-index-code* [*code nbe*]:
$\textit{subtract-index}\ (\textit{of-nat}\ n)\ (\textit{of-nat}\ m) = \textit{of-nat}\ (n - m)$
⟨*proof*⟩

lemma *minus-index-code* [*code*]:
$n - m = \textit{subtract-index}\ n\ m$
⟨*proof*⟩

lemma *times-index-code* [*code nbe*]:
of-nat n ∗ *of-nat m* = *of-nat* (*n* ∗ *m*)
⟨*proof*⟩

lemma *less-eq-index-code* [*code nbe*]:
of-nat n ≤ *of-nat m* ⟷ *n* ≤ *m*
⟨*proof*⟩

lemma *less-index-code* [*code nbe*]:
of-nat n < *of-nat m* ⟷ *n* < *m*
⟨*proof*⟩

lemma *Suc-index-minus-one*: *Suc-index n* − *1* = *n* ⟨*proof*⟩

lemma *of-nat-code* [*code*]:
of-nat = *Nat.of-nat*
⟨*proof*⟩

lemma *index-not-eq-zero*: *i* ≠ *of-nat 0* ⟷ *i* ≥ *1*
⟨*proof*⟩

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

lemma *nat-of-aux-code* [*code*]:
nat-of-aux i n = (*if i* = *0 then n else nat-of-aux* (*i* − *1*) (*Suc n*))
⟨*proof*⟩

lemma *nat-of-code* [*code*]:
nat-of i = *nat-of-aux i 0*
⟨*proof*⟩

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

lemma [*code*]:
div-mod-index n m = (*if m* = *0 then* (*0*, *n*) *else* (*n div m*, *n mod m*))
⟨*proof*⟩

lemma [*code*]:
n div m = *fst* (*div-mod-index n m*)
⟨*proof*⟩

lemma [*code*]:
n mod m = *snd* (*div-mod-index n m*)
⟨*proof*⟩

hide (**open**) *const of-nat nat-of*

14.4 ML interface

⟨*ML*⟩

14.5 Code generator setup

Implementation of indices by bounded integers

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

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

⟨*ML*⟩

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

code-const *op* + :: *index* ⇒ *index* ⇒ *index*
(*SML Int.+*/ ((-),/ (-)))
(*OCaml Pervasives.*(+))
(*Haskell* **infixl** *6* +)

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

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

code-const *div-mod-index*
(*SML* (*fn n => fn m =>*/ *if m = 0*/ *then* (*0*, *n*) *else*/ (*n div m*, *n mod m*)))
(*OCaml* (*fun n -> fun m ->*/ *if m = 0*/ *then* (*0*, *n*) *else*/ (*n* ′/ *m*, *n mod m*)))
(*Haskell divMod*)

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

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

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

Evaluation

lemma [*code*, *code del*]:
(*Code-Eval.term-of* :: *index* ⇒ *term*) = *Code-Eval.term-of* ⟨*proof*⟩

code-const *Code-Eval.term-of* :: *index* ⇒ *term*
(*SML HOLogic.mk'-number/ HOLogic.indexT/ (IntInf.fromInt/ -)*)

end

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

theory *Coinductive-List*
imports *List Main*
begin

15.1 List constructors over the datatype universe

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

lemma *CONS-not-NIL* [*iff*]: *CONS M N* ≠ *NIL*
and *NIL-not-CONS* [*iff*]: *NIL* ≠ *CONS M N*
and *CONS-inject* [*iff*]: (*CONS K M*) = (*CONS L N*) = (*K* = *L* ∧ *M* = *N*)
⟨*proof*⟩

lemma *CONS-mono*: *M* ⊆ *M'* ⟹ *N* ⊆ *N'* ⟹ *CONS M N* ⊆ *CONS M' N'*
⟨*proof*⟩

lemma *CONS-UN1*: *CONS M* (⋃*x*. *f x*) = (⋃*x*. *CONS M* (*f x*))
— A continuity result?
⟨*proof*⟩

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

lemma *List-case-NIL* [*simp*]: *List-case c h NIL* = *c*
and *List-case-CONS* [*simp*]: *List-case c h* (*CONS M N*) = *h M N*
⟨*proof*⟩

15.2 Corecursive lists

coinductive-set *LList* **for** *A*

where *NIL* [*intro*]: $NIL \in LList\ A$
| *CONS* [*intro*]: $a \in A \Longrightarrow M \in LList\ A \Longrightarrow CONS\ a\ M \in LList\ A$

lemma *LList-mono*:
assumes *subset*: $A \subseteq B$
shows $LList\ A \subseteq LList\ B$
— This justifies using *LList* in other recursive type definitions.
⟨*proof*⟩

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

definition $LList\text{-}corec\ a\ f = (\bigcup k.\ LList\text{-}corec\text{-}aux\ k\ f\ a)$

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

lemma *LList-corec*:
$LList\text{-}corec\ a\ f =$
(*case f a of None* ⇒ *NIL* | *Some* (z, w) ⇒ $CONS\ z\ (LList\text{-}corec\ w\ f)$)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *LList-corec-type*: $LList\text{-}corec\ a\ f \in LList\ UNIV$
⟨*proof*⟩

15.3 Abstract type definition

typedef $'a\ llist = LList\ (range\ Datatype.Leaf) :: 'a\ Datatype.item\ set$
⟨*proof*⟩

lemma *NIL-type*: $NIL \in llist$
⟨*proof*⟩

lemma *CONS-type*: $a \in range\ Datatype.Leaf \Longrightarrow$
$M \in llist \Longrightarrow CONS\ a\ M \in llist$
⟨*proof*⟩

lemma *llistI*: $x \in LList\ (range\ Datatype.Leaf) \Longrightarrow x \in llist$
⟨*proof*⟩

lemma *llistD*: $x \in llist \Longrightarrow x \in LList\ (range\ Datatype.Leaf)$

⟨*proof*⟩

lemma *Rep-llist-UNIV*: *Rep-llist x* ∈ *LList UNIV*
⟨*proof*⟩

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

code-datatype *LNil LCons*

lemma *LCons-not-LNil* [*iff*]: *LCons x xs* ≠ *LNil*
⟨*proof*⟩

lemma *LNil-not-LCons* [*iff*]: *LNil* ≠ *LCons x xs*
⟨*proof*⟩

lemma *LCons-inject* [*iff*]: (*LCons x xs* = *LCons y ys*) = (*x* = *y* ∧ *xs* = *ys*)
⟨*proof*⟩

lemma *Rep-llist-LNil*: *Rep-llist LNil* = *NIL*
⟨*proof*⟩

lemma *Rep-llist-LCons*: *Rep-llist* (*LCons x l*) =
CONS (*Datatype.Leaf x*) (*Rep-llist l*)
⟨*proof*⟩

lemma *llist-cases* [*cases type*: *llist*]:
obtains
(*LNil*) *l* = *LNil*
| (*LCons*) *x l′* **where** *l* = *LCons x l′*
⟨*proof*⟩

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

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

lemma *llist-case-LNil* [*simp*, *code*]: *llist-case c d LNil* = *c*
⟨*proof*⟩

lemma *llist-case-LCons* [*simp*, *code*]: *llist-case c d* (*LCons M N*) = *d M N*
⟨*proof*⟩

lemma *llist-case-cert*:
assumes $CASE \equiv$ *llist-case c d*
shows $(CASE\ LNil \equiv c)$ &&& $(CASE\ (LCons\ M\ N) \equiv d\ M\ N)$
⟨*proof*⟩

⟨*ML*⟩

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

lemma *LList-corec-type2*:
LList-corec a
(λ*z*. *case f z of None* ⇒ *None*
| *Some* (*v*, *w*) ⇒ *Some* (*Datatype.Leaf v*, *w*)) ∈ *llist*
(**is** *?corec a* ∈ -)
⟨*proof*⟩

lemma *llist-corec* [*code*]:
llist-corec a f =
(*case f a of None* ⇒ *LNil* | *Some* (*z*, *w*) ⇒ *LCons z* (*llist-corec w f*))
⟨*proof*⟩

15.4 Equality as greatest fixed-point – the bisimulation principle

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

lemma *EqLList-unfold*:
EqLList r = *dsum* (*Id-on* {*Datatype.Numb 0*}) (*dprod r* (*EqLList r*))
⟨*proof*⟩

lemma *EqLList-implies-ntrunc-equality*:
(*M*, *N*) ∈ *EqLList* (*Id-on A*) ⟹ *ntrunc k M* = *ntrunc k N*
⟨*proof*⟩

lemma *Domain-EqLList*: *Domain* (*EqLList* (*Id-on A*)) ⊆ *LList A*
⟨*proof*⟩

lemma *EqLList-Id-on*: *EqLList* (*Id-on A*) = *Id-on* (*LList A*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *EqLList-Id-on-iff* [*iff*]: $(p \in EqLList\ (Id\text{-}on\ A)) = (p \in Id\text{-}on\ (LList\ A))$
⟨*proof*⟩

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

lemma *LList-equalityI*
[*consumes 1*, *case-names EqLList*, *case-conclusion EqLList EqNIL EqCONS*]:
assumes r: $(M, N) \in r$
and *step*: $\bigwedge M\ N.\ (M, N) \in r \Longrightarrow$
$M = NIL \wedge N = NIL \vee$
$(\exists\, a\ b\ M'\ N'.$
$M = CONS\ a\ M' \wedge N = CONS\ b\ N' \wedge (a, b) \in Id\text{-}on\ A \wedge$
$((M', N') \in r \vee (M', N') \in EqLList\ (Id\text{-}on\ A)))$
shows $M = N$
⟨*proof*⟩

lemma *LList-fun-equalityI*
[*consumes 1*, *case-names NIL-type NIL CONS*, *case-conclusion CONS EqNIL EqCONS*]:
assumes M: $M \in LList\ A$
and *fun-NIL*: $g\ NIL \in LList\ A\quad f\ NIL = g\ NIL$
and *fun-CONS*: $\bigwedge x\ l.\ x \in A \Longrightarrow l \in LList\ A \Longrightarrow$
$(f\ (CONS\ x\ l), g\ (CONS\ x\ l)) = (NIL, NIL) \vee$
$(\exists\, M\ N\ a\ b.$
$(f\ (CONS\ x\ l), g\ (CONS\ x\ l)) = (CONS\ a\ M, CONS\ b\ N) \wedge$
$(a, b) \in Id\text{-}on\ A \wedge$
$(M, N) \in \{(f\ u, g\ u) \mid u.\ u \in LList\ A\} \cup Id\text{-}on\ (LList\ A))$
(**is** $\bigwedge x\ l.$ - $\Longrightarrow$ - $\Longrightarrow$ *?fun-CONS x l*)
shows $f\ M = g\ M$
⟨*proof*⟩

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

lemma *equals-LList-corec*:
assumes h: $\bigwedge x.\ h\ x =$
$(case\ f\ x\ of\ None \Rightarrow NIL \mid Some\ (z, w) \Rightarrow CONS\ z\ (h\ w))$
shows $h\ x = (\lambda x.\ LList\text{-}corec\ x\ f)\ x$
⟨*proof*⟩

lemma *llist-equalityI*
[*consumes 1*, *case-names Eqllist*, *case-conclusion Eqllist EqLNil EqLCons*]:
assumes r: $(l1, l2) \in r$
and *step*: $\bigwedge q.\ q \in r \Longrightarrow$
$q = (LNil, LNil) \vee$
$(\exists\, l1\ l2\ a\ b.$
$q = (LCons\ a\ l1, LCons\ b\ l2) \wedge a = b \wedge$
$((l1, l2) \in r \vee l1 = l2))$
(**is** $\bigwedge q.$ - $\Longrightarrow$ *?EqLNil q* $\vee$ *?EqLCons q*)
shows $l1 = l2$

⟨*proof*⟩

lemma *llist-fun-equalityI*
[*case-names LNil LCons*, *case-conclusion LCons EqLNil EqLCons*]:
assumes *fun-LNil*: *f LNil* = *g LNil*
and *fun-LCons*: ⋀*x l*.
(*f* (*LCons x l*), *g* (*LCons x l*)) = (*LNil*, *LNil*) ∨
(∃ *l1 l2 a b*.
(*f* (*LCons x l*), *g* (*LCons x l*)) = (*LCons a l1*, *LCons b l2*) ∧
a = *b* ∧ ((*l1*, *l2*) ∈ {(*f u*, *g u*) | *u*. *True*} ∨ *l1* = *l2*))
(**is** ⋀*x l*. *?fun-LCons x l*)
shows *f l* = *g l*
⟨*proof*⟩

15.5 Derived operations – both on the set and abstract type

15.5.1 *Lconst*

definition *Lconst M* ≡ *lfp* (λ*N*. *CONS M N*)

lemma *Lconst-fun-mono*: *mono* (*CONS M*)
⟨*proof*⟩

lemma *Lconst*: *Lconst M* = *CONS M* (*Lconst M*)
⟨*proof*⟩

lemma *Lconst-type*:
assumes *M* ∈ *A*
shows *Lconst M* ∈ *LList A*
⟨*proof*⟩

lemma *Lconst-eq-LList-corec*: *Lconst M* = *LList-corec M* (λ*x*. *Some* (*x*, *x*))
⟨*proof*⟩

lemma *gfp-Lconst-eq-LList-corec*:
gfp (λ*N*. *CONS M N*) = *LList-corec M* (λ*x*. *Some*(*x*, *x*))
⟨*proof*⟩

15.5.2 *Lmap* and *lmap*

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

lemma *Lmap-NIL* [*simp*]: *Lmap f NIL* = *NIL*
and *Lmap-CONS* [*simp*]: *Lmap f* (*CONS M N*) = *CONS* (*f M*) (*Lmap f N*)

⟨*proof*⟩

lemma *Lmap-type*:
assumes *M*: $M \in LList\ A$
and *f*: $\bigwedge x.\ x \in A \Longrightarrow f\ x \in B$
shows $Lmap\ f\ M \in LList\ B$
⟨*proof*⟩

lemma *Lmap-compose*:
assumes *M*: $M \in LList\ A$
shows $Lmap\ (f\ o\ g)\ M = Lmap\ f\ (Lmap\ g\ M)$ (**is** *?lhs M* = *?rhs M*)
⟨*proof*⟩

lemma *Lmap-ident*:
assumes *M*: $M \in LList\ A$
shows $Lmap\ (\lambda x.\ x)\ M = M$ (**is** *?lmap M* = -)
⟨*proof*⟩

lemma *lmap-LNil* [*simp*]: $lmap\ f\ LNil = LNil$
and *lmap-LCons* [*simp*]: $lmap\ f\ (LCons\ M\ N) = LCons\ (f\ M)\ (lmap\ f\ N)$
⟨*proof*⟩

lemma *lmap-compose* [*simp*]: $lmap\ (f\ o\ g)\ l = lmap\ f\ (lmap\ g\ l)$
⟨*proof*⟩

lemma *lmap-ident* [*simp*]: $lmap\ (\lambda x.\ x)\ l = l$
⟨*proof*⟩

15.5.3 *Lappend*

definition
$Lappend\ M\ N = LList\text{-}corec\ (M,\ N)$
(*split* (*List-case*
(*List-case None* (λ*N1 N2*. *Some* (*N1*, (*NIL*, *N2*))))
(λ*M1 M2 N*. *Some* (*M1*, (*M2*, *N*)))))
definition
$lappend\ l\ n = llist\text{-}corec\ (l,\ n)$
(*split* (*llist-case*
(*llist-case None* (λ*n1 n2*. *Some* (*n1*, (*LNil*, *n2*))))
(λ*l1 l2 n*. *Some* (*l1*, (*l2*, *n*)))))

lemma *Lappend-NIL-NIL* [*simp*]:
$Lappend\ NIL\ NIL = NIL$
and *Lappend-NIL-CONS* [*simp*]:
$Lappend\ NIL\ (CONS\ N\ N') = CONS\ N\ (Lappend\ NIL\ N')$
and *Lappend-CONS* [*simp*]:
$Lappend\ (CONS\ M\ M')\ N = CONS\ M\ (Lappend\ M'\ N)$
⟨*proof*⟩

lemma *Lappend-NIL* [*simp*]: $M \in LList\ A \Longrightarrow Lappend\ NIL\ M = M$
⟨*proof*⟩

lemma *Lappend-NIL2*: $M \in LList\ A \Longrightarrow Lappend\ M\ NIL = M$
⟨*proof*⟩

lemma *Lappend-type*:
assumes *M*: $M \in LList\ A$ **and** *N*: $N \in LList\ A$
shows $Lappend\ M\ N \in LList\ A$
⟨*proof*⟩

lemma *lappend-LNil-LNil* [*simp*]: $lappend\ LNil\ LNil = LNil$
and *lappend-LNil-LCons* [*simp*]: $lappend\ LNil\ (LCons\ l\ l') = LCons\ l\ (lappend\ LNil\ l')$
and *lappend-LCons* [*simp*]: $lappend\ (LCons\ l\ l')\ m = LCons\ l\ (lappend\ l'\ m)$
⟨*proof*⟩

lemma *lappend-LNil1* [*simp*]: $lappend\ LNil\ l = l$
⟨*proof*⟩

lemma *lappend-LNil2* [*simp*]: $lappend\ l\ LNil = l$
⟨*proof*⟩

lemma *lappend-assoc*: $lappend\ (lappend\ l1\ l2)\ l3 = lappend\ l1\ (lappend\ l2\ l3)$
⟨*proof*⟩

lemma *lmap-lappend-distrib*: $lmap\ f\ (lappend\ l\ n) = lappend\ (lmap\ f\ l)\ (lmap\ f\ n)$
⟨*proof*⟩

15.6 iterates

llist-fun-equalityI cannot be used here!

definition
$iterates :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ llist$ **where**
$iterates\ f\ a = llist\text{-}corec\ a\ (\lambda x.\ Some\ (x, f\ x))$

lemma *iterates*: $iterates\ f\ x = LCons\ x\ (iterates\ f\ (f\ x))$
⟨*proof*⟩

lemma *lmap-iterates*: $lmap\ f\ (iterates\ f\ x) = iterates\ f\ (f\ x)$
⟨*proof*⟩

lemma *iterates-lmap*: $iterates\ f\ x = LCons\ x\ (lmap\ f\ (iterates\ f\ x))$
⟨*proof*⟩

15.7 A rather complex proof about iterates – cf. Andy Pitts

lemma *funpow-lmap*:
fixes $f :: 'a \Rightarrow 'a$

shows (*lmap f ^ n*) (*LCons b l*) = *LCons* ((*f ^ n*) *b*) ((*lmap f ^ n*) *l*)
⟨*proof*⟩

lemma *iterates-equality*:
assumes *h*: ⋀*x*. *h x* = *LCons x* (*lmap f* (*h x*))
shows *h* = *iterates f*
⟨*proof*⟩

lemma *lappend-iterates*: *lappend* (*iterates f x*) *l* = *iterates f x*
⟨*proof*⟩

end

16 Commutative-Ring: Proving equalities in commutative rings

theory *Commutative-Ring*
imports *List Parity Main*
uses (*comm-ring.ML*)
begin

Syntax of multivariate polynomials (pol) and polynomial expressions.

datatype *'a pol* =
Pc 'a
| *Pinj nat 'a pol*
| *PX 'a pol nat 'a pol*

datatype *'a polex* =
Pol 'a pol
| *Add 'a polex 'a polex*
| *Sub 'a polex 'a polex*
| *Mul 'a polex 'a polex*
| *Pow 'a polex nat*
| *Neg 'a polex*

Interpretation functions for the shadow syntax.

fun
Ipol :: *'a*::{*comm-ring*,*recpower*} *list* ⇒ *'a pol* ⇒ *'a*
where
Ipol l (*Pc c*) = *c*
| *Ipol l* (*Pinj i P*) = *Ipol* (*drop i l*) *P*
| *Ipol l* (*PX P x Q*) = *Ipol l P* ∗ (*hd l*) ^*x* + *Ipol* (*drop 1 l*) *Q*

fun
Ipolex :: *'a*::{*comm-ring*,*recpower*} *list* ⇒ *'a polex* ⇒ *'a*
where

```
    Ipolex l (Pol P) = Ipol l P
  | Ipolex l (Add P Q) = Ipolex l P + Ipolex l Q
  | Ipolex l (Sub P Q) = Ipolex l P - Ipolex l Q
  | Ipolex l (Mul P Q) = Ipolex l P * Ipolex l Q
  | Ipolex l (Pow p n) = Ipolex l p ^ n
  | Ipolex l (Neg P) = - Ipolex l P
```

Create polynomial normalized polynomials given normalized inputs.

```
definition
  mkPinj :: nat ⇒ 'a pol ⇒ 'a pol where
  mkPinj x P = (case P of
    Pc c ⇒ Pc c |
    Pinj y P ⇒ Pinj (x + y) P |
    PX p1 y p2 ⇒ Pinj x P)

definition
  mkPX :: 'a::{comm-ring,recpower} pol ⇒ nat ⇒ 'a pol ⇒ 'a pol where
  mkPX P i Q = (case P of
    Pc c ⇒ (if (c = 0) then (mkPinj 1 Q) else (PX P i Q)) |
    Pinj j R ⇒ PX P i Q |
    PX P2 i2 Q2 ⇒ (if (Q2 = (Pc 0)) then (PX P2 (i+i2) Q) else (PX P i Q))
)
```

Defining the basic ring operations on normalized polynomials

```
function
  add :: 'a::{comm-ring,recpower} pol ⇒ 'a pol ⇒ 'a pol (infixl ⊕ 65)
where
    Pc a ⊕ Pc b = Pc (a + b)
  | Pc c ⊕ Pinj i P = Pinj i (P ⊕ Pc c)
  | Pinj i P ⊕ Pc c = Pinj i (P ⊕ Pc c)
  | Pc c ⊕ PX P i Q = PX P i (Q ⊕ Pc c)
  | PX P i Q ⊕ Pc c = PX P i (Q ⊕ Pc c)
  | Pinj x P ⊕ Pinj y Q =
      (if x = y then mkPinj x (P ⊕ Q)
       else (if x > y then mkPinj y (Pinj (x - y) P ⊕ Q)
         else mkPinj x (Pinj (y - x) Q ⊕ P)))
  | Pinj x P ⊕ PX Q y R =
      (if x = 0 then P ⊕ PX Q y R
       else (if x = 1 then PX Q y (R ⊕ P)
         else PX Q y (R ⊕ Pinj (x - 1) P)))
  | PX P x R ⊕ Pinj y Q =
      (if y = 0 then PX P x R ⊕ Q
       else (if y = 1 then PX P x (R ⊕ Q)
         else PX P x (R ⊕ Pinj (y - 1) Q)))
  | PX P1 x P2 ⊕ PX Q1 y Q2 =
      (if x = y then mkPX (P1 ⊕ Q1) x (P2 ⊕ Q2)
       else (if x > y then mkPX (PX P1 (x - y) (Pc 0) ⊕ Q1) y (P2 ⊕ Q2)
         else mkPX (PX Q1 (y-x) (Pc 0) ⊕ P1) x (P2 ⊕ Q2)))
⟨proof⟩
```

termination ⟨*proof*⟩

function
mul :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol* ⇒ *'a pol* (**infixl** ⊗ *70*)
where
Pc a ⊗ *Pc b* = *Pc* (*a* ∗ *b*)
| *Pc c* ⊗ *Pinj i P* =
(*if c* = *0 then Pc 0 else mkPinj i* (*P* ⊗ *Pc c*))
| *Pinj i P* ⊗ *Pc c* =
(*if c* = *0 then Pc 0 else mkPinj i* (*P* ⊗ *Pc c*))
| *Pc c* ⊗ *PX P i Q* =
(*if c* = *0 then Pc 0 else mkPX* (*P* ⊗ *Pc c*) *i* (*Q* ⊗ *Pc c*))
| *PX P i Q* ⊗ *Pc c* =
(*if c* = *0 then Pc 0 else mkPX* (*P* ⊗ *Pc c*) *i* (*Q* ⊗ *Pc c*))
| *Pinj x P* ⊗ *Pinj y Q* =
(*if x* = *y then mkPinj x* (*P* ⊗ *Q*) *else*
(*if x* > *y then mkPinj y* (*Pinj* (*x*−*y*) *P* ⊗ *Q*)
else mkPinj x (*Pinj* (*y* − *x*) *Q* ⊗ *P*)))
| *Pinj x P* ⊗ *PX Q y R* =
(*if x* = *0 then P* ⊗ *PX Q y R else*
(*if x* = *1 then mkPX* (*Pinj x P* ⊗ *Q*) *y* (*R* ⊗ *P*)
else mkPX (*Pinj x P* ⊗ *Q*) *y* (*R* ⊗ *Pinj* (*x* − *1*) *P*)))
| *PX P x R* ⊗ *Pinj y Q* =
(*if y* = *0 then PX P x R* ⊗ *Q else*
(*if y* = *1 then mkPX* (*Pinj y Q* ⊗ *P*) *x* (*R* ⊗ *Q*)
else mkPX (*Pinj y Q* ⊗ *P*) *x* (*R* ⊗ *Pinj* (*y* − *1*) *Q*)))
| *PX P1 x P2* ⊗ *PX Q1 y Q2* =
mkPX (*P1* ⊗ *Q1*) (*x* + *y*) (*P2* ⊗ *Q2*) ⊕
(*mkPX* (*P1* ⊗ *mkPinj 1 Q2*) *x* (*Pc 0*) ⊕
(*mkPX* (*Q1* ⊗ *mkPinj 1 P2*) *y* (*Pc 0*)))
⟨*proof*⟩
termination ⟨*proof*⟩

Negation

fun
neg :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol*
where
neg (*Pc c*) = *Pc* (−*c*)
| *neg* (*Pinj i P*) = *Pinj i* (*neg P*)
| *neg* (*PX P x Q*) = *PX* (*neg P*) *x* (*neg Q*)

Substraction

definition
sub :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol* ⇒ *'a pol* (**infixl** ⊖ *65*)
where
sub P Q = *P* ⊕ *neg Q*

Square for Fast Exponentation

fun

sqr :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol*
where
sqr (*Pc c*) = *Pc* (*c* ∗ *c*)
| *sqr* (*Pinj i P*) = *mkPinj i* (*sqr P*)
| *sqr* (*PX A x B*) = *mkPX* (*sqr A*) (*x* + *x*) (*sqr B*) ⊕
mkPX (*Pc* (*1* + *1*) ⊗ *A* ⊗ *mkPinj 1 B*) *x* (*Pc 0*)

Fast Exponentation

fun
pow :: *nat* ⇒ *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol*
where
pow 0 P = *Pc 1*
| *pow n P* = (*if even n then pow* (*n div 2*) (*sqr P*)
else P ⊗ *pow* (*n div 2*) (*sqr P*))

lemma *pow-if*:
pow n P =
(*if n* = *0 then Pc 1 else if even n then pow* (*n div 2*) (*sqr P*)
else P ⊗ *pow* (*n div 2*) (*sqr P*))
⟨*proof*⟩

Normalization of polynomial expressions

fun
norm :: *'a*::{*comm-ring*,*recpower*} *polex* ⇒ *'a pol*
where
norm (*Pol P*) = *P*
| *norm* (*Add P Q*) = *norm P* ⊕ *norm Q*
| *norm* (*Sub P Q*) = *norm P* ⊖ *norm Q*
| *norm* (*Mul P Q*) = *norm P* ⊗ *norm Q*
| *norm* (*Pow P n*) = *pow n* (*norm P*)
| *norm* (*Neg P*) = *neg* (*norm P*)

mkPinj preserve semantics

lemma *mkPinj-ci*: *Ipol l* (*mkPinj a B*) = *Ipol l* (*Pinj a B*)
⟨*proof*⟩

mkPX preserves semantics

lemma *mkPX-ci*: *Ipol l* (*mkPX A b C*) = *Ipol l* (*PX A b C*)
⟨*proof*⟩

Correctness theorems for the implemented operations

Negation

lemma *neg-ci*: *Ipol l* (*neg P*) = −(*Ipol l P*)
⟨*proof*⟩

Addition

lemma *add-ci*: *Ipol l* (*P* ⊕ *Q*) = *Ipol l P* + *Ipol l Q*
⟨*proof*⟩

Multiplication

lemma *mul-ci*: *Ipol l* (*P* ⊗ *Q*) = *Ipol l P* ∗ *Ipol l Q*
⟨*proof*⟩

Substraction

lemma *sub-ci*: *Ipol l* (*P* ⊖ *Q*) = *Ipol l P* − *Ipol l Q*
⟨*proof*⟩

Square

lemma *sqr-ci*: *Ipol ls* (*sqr P*) = *Ipol ls P* ∗ *Ipol ls P*
⟨*proof*⟩

Power

lemma *even-pow*:*even n* ⟹ *pow n P* = *pow* (*n div 2*) (*sqr P*)
⟨*proof*⟩

lemma *pow-ci*: *Ipol ls* (*pow n P*) = *Ipol ls P* ^ *n*
⟨*proof*⟩

Normalization preserves semantics

lemma *norm-ci*: *Ipolex l Pe* = *Ipol l* (*norm Pe*)
⟨*proof*⟩

Reflection lemma: Key to the (incomplete) decision procedure

lemma *norm-eq*:
assumes *norm P1* = *norm P2*
shows *Ipolex l P1* = *Ipolex l P2*
⟨*proof*⟩

⟨*ML*⟩

end

17 Continuity: Continuity and iterations (of set transformers)

theory *Continuity*
imports *Relation-Power Main*
begin

17.1 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*.

17.2 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*⟩

17.3 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*⟩

17.4 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

18 ContNotDenum: Non-denumerability of the Continuum.

theory *ContNotDenum*
imports *Complex-Main*
begin

18.1 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.

18.2 Closed Intervals

This section formalises some properties of closed intervals.

18.2.1 Definition

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

18.2.2 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* ∈ *closed-int a b* ∧ (∀ *x* ∈ *closed-int a b*. *a* ≤ *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*⟩

18.3 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*⟩

18.4 Generating the intervals

18.4.1 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*⟩

18.5 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 $(f\ 0) \notin newInt\ 0\ f$.

18.5.1 Definition

primrec *newInt* :: $nat \Rightarrow (nat \Rightarrow real) \Rightarrow (real\ set)$ **where**
  $newInt\ 0\ f = closed\text{-}int\ (f\ 0 + 1)\ (f\ 0 + 2)$
  | $newInt\ (Suc\ n)\ f =$
    $(SOME\ e.\ (\exists\ e1\ e2.$
     $e1 < e2 \wedge$
     $e = closed\text{-}int\ e1\ e2 \wedge$
     $e \subseteq (newInt\ n\ f) \wedge$
     $(f\ (Suc\ n)) \notin e)$
)

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

18.5.2 Properties

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

lemma *newInt-ex*:
  $\exists\ a\ b.\ a < b \wedge$
   $newInt\ (Suc\ n)\ f = closed\text{-}int\ a\ b \wedge$
   $newInt\ (Suc\ n)\ f \subseteq newInt\ n\ f \wedge$
   $f\ (Suc\ n) \notin newInt\ (Suc\ n)\ f$
⟨*proof*⟩

lemma *newInt-subset*:
  $newInt\ (Suc\ n)\ f \subseteq newInt\ n\ f$
  ⟨*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*:
  $\forall\ n.\ f\ n \notin (\bigcap n.\ newInt\ n\ f)$
⟨*proof*⟩

lemma *newInt-notempty*:
  $(\bigcap n.\ newInt\ n\ f) \neq \{\}$
⟨*proof*⟩

18.6 Final Theorem

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

end

19 Nat-Int-Bij: Bijections $\mathbb{N} \to \mathbb{N}^2$ and $\mathbb{N} \to \mathbb{Z}$

theory *Nat-Int-Bij*
imports *Main*
begin

19.1 A bijection between $\mathbb{N}$ and $\mathbb{N}^2$

Definition and proofs are from [3, page 85].

definition *nat2-to-nat*:: $(nat * nat) \Rightarrow nat$ **where**
nat2-to-nat pair = (*let* (*n*,*m*) = *pair in* (*n*+*m*) * *Suc* (*n*+*m*) *div 2* + *n*)
definition *nat-to-nat2*:: $nat \Rightarrow (nat * nat)$ **where**
nat-to-nat2 = *inv nat2-to-nat*

lemma *dvd2-a-x-suc-a*: *2 dvd a* * (*Suc a*)
⟨*proof*⟩

lemma
assumes *eq*: *nat2-to-nat* (*u*,*v*) = *nat2-to-nat* (*x*,*y*)
shows *nat2-to-nat-help*: *u*+*v* ≤ *x*+*y*
⟨*proof*⟩

theorem *nat2-to-nat-inj*: *inj nat2-to-nat*
⟨*proof*⟩

lemma *nat-to-nat2-surj*: *surj nat-to-nat2*
⟨*proof*⟩

lemma *gauss-sum-nat-upto*: *2* * (∑ *i*≤*n*::*nat*. *i*) = *n* * (*n* + *1*)
⟨*proof*⟩

lemma *nat2-to-nat-surj*: *surj nat2-to-nat*
⟨*proof*⟩

19.2 A bijection between $\mathbb{N}$ and $\mathbb{Z}$

definition *nat-to-int-bij* :: $nat \Rightarrow int$ **where**
nat-to-int-bij n = (*if 2 dvd n then int*(*n div 2*) *else* −*int*(*Suc n div 2*))

definition *int-to-nat-bij* :: $int \Rightarrow nat$ **where**
int-to-nat-bij i = (*if 0*<=*i then 2*∗*nat*(*i*) *else 2*∗*nat*(−*i*) − *1*)

lemma *i2n-n2i-id*: *int-to-nat-bij* (*nat-to-int-bij n*) = *n*
⟨*proof*⟩

lemma *n2i-i2n-id*: *nat-to-int-bij*(*int-to-nat-bij i*) = *i*
⟨*proof*⟩

lemma *inv-nat-to-int-bij*: *inv nat-to-int-bij* = *int-to-nat-bij*
⟨*proof*⟩

lemma *inv-int-to-nat-bij*: *inv int-to-nat-bij* = *nat-to-int-bij*
⟨*proof*⟩

lemma *surj-nat-to-int-bij*: *surj nat-to-int-bij*
⟨*proof*⟩

lemma *surj-int-to-nat-bij*: *surj int-to-nat-bij*
⟨*proof*⟩

lemma *inj-nat-to-int-bij*: *inj nat-to-int-bij*
⟨*proof*⟩

lemma *inj-int-to-nat-bij*: *inj int-to-nat-bij*
⟨*proof*⟩

end

20 Countable: Encoding (almost) everything into natural numbers

theory *Countable*
imports
~~/*src*/*HOL*/*List*
~~/*src*/*HOL*/*Hilbert-Choice*
~~/*src*/*HOL*/*Nat-Int-Bij*
~~/*src*/*HOL*/*Rational*
Main
begin

20.1 The class of countable types

class *countable* =
assumes *ex-inj*: ∃ *to-nat* :: *'a* ⇒ *nat*. *inj to-nat*

lemma *countable-classI*:
fixes *f* :: *'a* ⇒ *nat*
assumes ⋀*x y*. *f x* = *f y* ⟹ *x* = *y*
shows *OFCLASS*(*'a*, *countable-class*)
⟨*proof*⟩

20.2 Conversion functions

definition *to-nat* :: *'a*::*countable* ⇒ *nat* **where**
to-nat = (*SOME f*. *inj f*)

definition *from-nat* :: *nat* ⇒ *'a*::*countable* **where**
from-nat = *inv* (*to-nat* :: *'a* ⇒ *nat*)

lemma *inj-to-nat* [*simp*]: *inj to-nat*
⟨*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*⟩

20.3 Countable types

instance *nat* :: *countable*
⟨*proof*⟩

subclass (**in** *finite*) *countable*
⟨*proof*⟩

Pairs

primrec *sum* :: *nat* ⇒ *nat*
where
sum 0 = *0*
| *sum* (*Suc n*) = *Suc n* + *sum n*

lemma *sum-arith*: *sum n* = *n* ∗ *Suc n div 2*
⟨*proof*⟩

lemma *sum-mono*: *n* ≥ *m* ⟹ *sum n* ≥ *sum m*
⟨*proof*⟩

definition
pair-encode = (λ(*m*, *n*). *sum* (*m* + *n*) + *m*)

lemma *inj-pair-cencode*: *inj pair-encode*
⟨*proof*⟩

instance ∗ :: (*countable*, *countable*) *countable*
⟨*proof*⟩

Sums

instance +:: (*countable*, *countable*) *countable*
⟨*proof*⟩

Integers

lemma *int-cases*: (*i*::*int*) = *0* ∨ *i* < *0* ∨ *i* > *0*
⟨*proof*⟩

lemma *int-pos-neg-zero*:
obtains (*zero*) (*z*::*int*) = *0* *sgn z* = *0* *abs z* = *0*
| (*pos*) *n* **where** *z* = *of-nat n* *sgn z* = *1* *abs z* = *of-nat n*
| (*neg*) *n* **where** *z* = − (*of-nat n*) *sgn z* = −*1* *abs z* = *of-nat n*
⟨*proof*⟩

instance *int* :: *countable*
⟨*proof*⟩

Options

instance *option* :: (*countable*) *countable*
⟨*proof*⟩

Lists

lemma *from-nat-to-nat-map* [*simp*]: *map from-nat* (*map to-nat xs*) = *xs*
⟨*proof*⟩

primrec
list-encode :: '*a*::*countable list* ⇒ *nat*
where
list-encode [] = *0*
| *list-encode* (*x*#*xs*) = *Suc* (*to-nat* (*x*, *list-encode xs*))

instance *list* :: (*countable*) *countable*
⟨*proof*⟩

Functions

instance *fun* :: (*finite*, *countable*) *countable*
⟨*proof*⟩

20.4 The Rationals are Countably Infinite

definition *nat-to-rat-surj* :: *nat* ⇒ *rat* **where**
nat-to-rat-surj n = (*let* (*a*,*b*) = *nat-to-nat2 n*
in Fract (*nat-to-int-bij a*) (*nat-to-int-bij 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*:
$\mathbb{Q} = range\ (of\text{-}rat\ o\ nat\text{-}to\text{-}rat\text{-}surj)$
⟨*proof*⟩

lemma *surj-of-rat-nat-to-rat-surj*:
$r \in \mathbb{Q} \Longrightarrow \exists\, n.\ r = of\text{-}rat(nat\text{-}to\text{-}rat\text{-}surj\ n)$
⟨*proof*⟩

end

instance *rat* :: *countable*
⟨*proof*⟩

end

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

theory *Dense-Linear-Order*
imports *Main*
uses
"~~/src/HOL/Tools/Qelim/langford-data.ML"
"~~/src/HOL/Tools/Qelim/ferrante-rackoff-data.ML"
("~~/src/HOL/Tools/Qelim/langford.ML")
("~~/src/HOL/Tools/Qelim/ferrante-rackoff.ML")
begin

⟨*ML*⟩

context *linorder*
begin

lemma *less-not-permute*[*noatp*]: $\neg\ (x < y \wedge y < x)$ ⟨*proof*⟩

lemma *gather-simps*[*noatp*]:
shows
$(\exists\, x.\ (\forall\, y \in L.\ y < x) \wedge (\forall\, y \in U.\ x < y) \wedge x < u \wedge P\ x) \longleftrightarrow (\exists\, x.\ (\forall\, y \in L.\ y < x) \wedge (\forall\, y \in (insert\ u\ U).\ x < y) \wedge P\ x)$
and $(\exists\, x.\ (\forall\, y \in L.\ y < x) \wedge (\forall\, y \in U.\ x < y) \wedge l < x \wedge P\ x) \longleftrightarrow (\exists\, x.\ (\forall\, y \in (insert\ l\ L).\ y < x) \wedge (\forall\, y \in U.\ x < y) \wedge P\ x)$
$(\exists\, x.\ (\forall\, y \in L.\ y < x) \wedge (\forall\, y \in U.\ x < y) \wedge x < u) \longleftrightarrow (\exists\, x.\ (\forall\, y \in L.\ y < x) \wedge (\forall\, y \in (insert\ u\ U).\ x < y))$
and $(\exists\, x.\ (\forall\, y \in L.\ y < x) \wedge (\forall\, y \in U.\ x < y) \wedge l < x) \longleftrightarrow (\exists\, x.\ (\forall\, y \in (insert$

$l\ L).\ y < x) \wedge (\forall\, y \in U.\ x < y))$ ⟨*proof*⟩

lemma
gather-start[*noatp*]: $(\exists\, x.\ P\ x) \equiv (\exists\, x.\ (\forall\, y \in \{\}.\ y < x) \wedge (\forall\, y \in \{\}.\ x < y) \wedge P\ x)$
⟨*proof*⟩

Theorems for $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (P\ x \longleftrightarrow P_{-\infty})$

lemma *minf-lt*[*noatp*]: $\exists\, z\ .\ \forall\, x.\ x < z \longrightarrow (x < t \longleftrightarrow True)$ ⟨*proof*⟩
lemma *minf-gt*[*noatp*]: $\exists\, z\ .\ \forall\, x.\ x < z \longrightarrow\ (t < x \longleftrightarrow\ False)$
⟨*proof*⟩

lemma *minf-le*[*noatp*]: $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (x \leq t \longleftrightarrow True)$ ⟨*proof*⟩
lemma *minf-ge*[*noatp*]: $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (t \leq x \longleftrightarrow False)$
⟨*proof*⟩
lemma *minf-eq*[*noatp*]: $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (x = t \longleftrightarrow False)$ ⟨*proof*⟩
lemma *minf-neq*[*noatp*]: $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (x \neq t \longleftrightarrow True)$ ⟨*proof*⟩
lemma *minf-P*[*noatp*]: $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (P \longleftrightarrow P)$ ⟨*proof*⟩

Theorems for $\exists\, z.\ \forall\, x.\ x < z \longrightarrow (P\ x \longleftrightarrow P_{+\infty})$

lemma *pinf-gt*[*noatp*]: $\exists\, z\ .\ \forall\, x.\ z < x \longrightarrow (t < x \longleftrightarrow True)$ ⟨*proof*⟩
lemma *pinf-lt*[*noatp*]: $\exists\, z\ .\ \forall\, x.\ z < x \longrightarrow\ (x < t \longleftrightarrow\ False)$
⟨*proof*⟩

lemma *pinf-ge*[*noatp*]: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (t \leq x \longleftrightarrow True)$ ⟨*proof*⟩
lemma *pinf-le*[*noatp*]: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (x \leq t \longleftrightarrow False)$
⟨*proof*⟩
lemma *pinf-eq*[*noatp*]: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (x = t \longleftrightarrow False)$ ⟨*proof*⟩
lemma *pinf-neq*[*noatp*]: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (x \neq t \longleftrightarrow True)$ ⟨*proof*⟩
lemma *pinf-P*[*noatp*]: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (P \longleftrightarrow P)$ ⟨*proof*⟩

lemma *nmi-lt*[*noatp*]: $t \in U \Longrightarrow \forall\, x.\ \neg True \wedge x < t \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)$
⟨*proof*⟩
lemma *nmi-gt*[*noatp*]: $t \in U \Longrightarrow \forall\, x.\ \neg False \wedge t < x \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)$
⟨*proof*⟩
lemma *nmi-le*[*noatp*]: $t \in U \Longrightarrow \forall\, x.\ \neg True \wedge x{\leq}\ t \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)$
⟨*proof*⟩
lemma *nmi-ge*[*noatp*]: $t \in U \Longrightarrow \forall\, x.\ \neg False \wedge t{\leq}\ x \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)$
⟨*proof*⟩
lemma *nmi-eq*[*noatp*]: $t \in U \Longrightarrow \forall\, x.\ \neg False \wedge\ \ x = t \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)$
⟨*proof*⟩
lemma *nmi-neq*[*noatp*]: $t \in U \Longrightarrow \forall\, x.\ \neg True \wedge x \neq t \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)$
⟨*proof*⟩
lemma *nmi-P*[*noatp*]: $\forall\ x.\ {}^{\sim}P \wedge P \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)$ ⟨*proof*⟩
lemma *nmi-conj*[*noatp*]: $[\![\forall\, x.\ \neg P1\,' \wedge P1\ x \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)\ ;$
$\forall\, x.\ \neg P2\,' \wedge P2\ x \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)]\!] \Longrightarrow$
$\forall\, x.\ \neg(P1\,' \wedge P2\,') \wedge (P1\ x \wedge P2\ x) \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)$ ⟨*proof*⟩
lemma *nmi-disj*[*noatp*]: $[\![\forall\, x.\ \neg P1\,' \wedge P1\ x \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)\ ;$
$\forall\, x.\ \neg P2\,' \wedge P2\ x \longrightarrow\ (\exists\ u{\in}\ U.\ u \leq x)]\!] \Longrightarrow$

$\forall x.\ \neg(P1'\vee P2')\wedge(P1\ x\vee P2\ x)\longrightarrow\ (\exists\ u{\in}\ U.\ u\leq x)$ ⟨*proof*⟩

lemma *npi-lt*[*noatp*]: $t\in U\Longrightarrow\forall x.\ \neg False\wedge\ \ x<t\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)$ ⟨*proof*⟩
lemma *npi-gt*[*noatp*]: $t\in U\Longrightarrow\forall x.\ \neg True\wedge t<x\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)$ ⟨*proof*⟩
lemma *npi-le*[*noatp*]: $t\in U\Longrightarrow\forall x.\ \neg False\wedge\ \ x\leq t\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)$ ⟨*proof*⟩
lemma *npi-ge*[*noatp*]: $t\in U\Longrightarrow\forall x.\ \neg True\wedge t\leq x\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)$ ⟨*proof*⟩
lemma *npi-eq*[*noatp*]: $t\in U\Longrightarrow\forall x.\ \neg False\wedge\ \ x=t\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)$ ⟨*proof*⟩
lemma *npi-neq*[*noatp*]: $t\in U\Longrightarrow\forall x.\ \neg True\wedge x\neq t\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u\ )$ ⟨*proof*⟩
lemma *npi-P*[*noatp*]: $\forall\ x.\ {}^{\sim}P\wedge P\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)$ ⟨*proof*⟩
lemma *npi-conj*[*noatp*]: $[\![\forall x.\ \neg P1'\wedge P1\ x\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)\ ;\ \forall x.\ \neg P2'\wedge P2\ x\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)]\!]$
$\Longrightarrow\ \forall x.\ \neg(P1'\wedge P2')\wedge(P1\ x\wedge P2\ x)\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)$ ⟨*proof*⟩
lemma *npi-disj*[*noatp*]: $[\![\forall x.\ \neg P1'\wedge P1\ x\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)\ ;\forall x.\ \neg P2'\wedge P2\ x\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)]\!]$
$\Longrightarrow\forall x.\ \neg(P1'\vee P2')\wedge(P1\ x\vee P2\ x)\longrightarrow\ (\exists\ u{\in}\ U.\ x\leq u)$ ⟨*proof*⟩

lemma *lin-dense-lt*[*noatp*]: $t\in U\Longrightarrow\forall x\ l\ u.\ (\forall\ t.\ l<t\wedge t<u\longrightarrow t\notin U)\wedge l<x\wedge x<u\wedge x<t\longrightarrow(\forall\ y.\ l<y\wedge y<u\longrightarrow y<t)$
⟨*proof*⟩

lemma *lin-dense-gt*[*noatp*]: $t\in U\Longrightarrow\forall x\ l\ u.\ (\forall\ t.\ l<t\wedge t<u\longrightarrow t\notin U)\wedge l<x\wedge x<u\wedge t<x\longrightarrow(\forall\ y.\ l<y\wedge y<u\longrightarrow t<y)$
⟨*proof*⟩

lemma *lin-dense-le*[*noatp*]: $t\in U\Longrightarrow\forall x\ l\ u.\ (\forall\ t.\ l<t\wedge t<u\longrightarrow t\notin U)\wedge l<x\wedge x<u\wedge x\leq t\longrightarrow(\forall\ y.\ l<y\wedge y<u\longrightarrow y\leq t)$
⟨*proof*⟩

lemma *lin-dense-ge*[*noatp*]: $t\in U\Longrightarrow\forall x\ l\ u.\ (\forall\ t.\ l<t\wedge t<u\longrightarrow t\notin U)\wedge l<x\wedge x<u\wedge t\leq x\longrightarrow(\forall\ y.\ l<y\wedge y<u\longrightarrow t\leq y)$
⟨*proof*⟩
lemma *lin-dense-eq*[*noatp*]: $t\in U\Longrightarrow\forall x\ l\ u.\ (\forall\ t.\ l<t\wedge t<u\longrightarrow t\notin U)\wedge l<x\wedge x<u\wedge x=t\ \longrightarrow(\forall\ y.\ l<y\wedge y<u\longrightarrow y=t)$ ⟨*proof*⟩
lemma *lin-dense-neq*[*noatp*]: $t\in U\Longrightarrow\forall x\ l\ u.\ (\forall\ t.\ l<t\wedge t<u\longrightarrow t\notin U)\wedge l<x\wedge x<u\wedge x\neq t\ \longrightarrow(\forall\ y.\ l<y\wedge y<u\longrightarrow y\neq t)$ ⟨*proof*⟩
lemma *lin-dense-P*[*noatp*]: $\forall x\ l\ u.\ (\forall\ t.\ l<t\wedge t<u\longrightarrow t\notin U)\wedge l<x\wedge x<u\wedge P\ \longrightarrow(\forall\ y.\ l<y\wedge y<u\longrightarrow P)$ ⟨*proof*⟩

lemma *lin-dense-conj*[*noatp*]:
$[\![\forall x\ l\ u.\ (\forall\ t.\ l<t\wedge t<u\longrightarrow t\notin U)\wedge l<x\wedge x<u\wedge P1\ x$
$\longrightarrow(\forall\ y.\ l<y\wedge y<u\longrightarrow P1\ y)\ ;$
$\forall x\ l\ u.\ (\forall\ t.\ l<t\wedge t<u\longrightarrow t\notin U)\wedge l<x\wedge x<u\wedge P2\ x$
$\longrightarrow(\forall\ y.\ l<y\wedge y<u\longrightarrow P2\ y)]\!]\Longrightarrow$

$\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge (P1\ x \wedge P2\ x)$
$\longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow (P1\ y \wedge P2\ y))$
⟨*proof*⟩

lemma *lin-dense-disj*[*noatp*]:
$[\![\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P1\ x$
$\longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow P1\ y)\ ;$
$\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P2\ x$
$\longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow P2\ y)]\!] \Longrightarrow$
$\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge (P1\ x \vee P2\ x)$
$\longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow (P1\ y \vee P2\ y))$
⟨*proof*⟩

lemma *npmibnd*[*noatp*]: $[\![\forall x.\ \neg\ MP \wedge P\ x \longrightarrow (\exists\ u \in U.\ u \leq x);\ \forall x.\ \neg PP \wedge P\ x \longrightarrow (\exists\ u \in U.\ x \leq u)]\!]$
$\Longrightarrow \forall x.\ \neg\ MP \wedge \neg PP \wedge P\ x \longrightarrow (\exists\ u \in U.\ \exists\ u' \in U.\ u \leq x \wedge x \leq u')$
⟨*proof*⟩

lemma *finite-set-intervals*[*noatp*]:
assumes *px*: $P\ x$ **and** *lx*: $l \leq x$ **and** *xu*: $x \leq u$ **and** *linS*: $l \in S$
and *uinS*: $u \in S$ **and** *fS*:*finite S* **and** *lS*: $\forall\ x \in S.\ l \leq x$ **and** *Su*: $\forall\ x \in S.\ x \leq u$
shows $\exists\ a \in S.\ \exists\ b \in S.\ (\forall\ y.\ a < y \wedge y < b \longrightarrow y \notin S) \wedge a \leq x \wedge x \leq b \wedge P\ x$
⟨*proof*⟩

lemma *finite-set-intervals2*[*noatp*]:
assumes *px*: $P\ x$ **and** *lx*: $l \leq x$ **and** *xu*: $x \leq u$ **and** *linS*: $l \in S$
and *uinS*: $u \in S$ **and** *fS*:*finite S* **and** *lS*: $\forall\ x \in S.\ l \leq x$ **and** *Su*: $\forall\ x \in S.\ x \leq u$
shows $(\exists\ s \in S.\ P\ s) \vee (\exists\ a \in S.\ \exists\ b \in S.\ (\forall\ y.\ a < y \wedge y < b \longrightarrow y \notin S) \wedge a < x \wedge x < b \wedge P\ x)$
⟨*proof*⟩

end

22 The classical QE after Langford for dense linear orders

context *dense-linear-order*
begin

lemma *interval-empty-iff*:
$\{y.\ x < y \wedge y < z\} = \{\} \longleftrightarrow \neg\ x < z$
⟨*proof*⟩

lemma *dlo-qe-bnds*[*noatp*]:
assumes *ne*: $L \neq \{\}$ **and** *neU*: $U \neq \{\}$ **and** *fL*: *finite L* **and** *fU*: *finite U*
shows $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y)) \equiv (\forall\ l \in L.\ \forall u \in U.\ l < u)$

⟨*proof*⟩

lemma *dlo-qe-noub*[*noatp*]:
  assumes *ne*: $L \neq \{\}$ **and** *fL*: *finite L*
  shows $(\exists\, x.\ (\forall\, y \in L.\ y < x) \wedge (\forall\, y \in \{\}.\ x < y)) \equiv \mathit{True}$
⟨*proof*⟩

lemma *dlo-qe-nolb*[*noatp*]:
  assumes *ne*: $U \neq \{\}$ **and** *fU*: *finite U*
  shows $(\exists\, x.\ (\forall\, y \in \{\}.\ y < x) \wedge (\forall\, y \in U.\ x < y)) \equiv \mathit{True}$
⟨*proof*⟩

lemma *exists-neq*[*noatp*]: $\exists (x::'a).\ x \neq t\ \exists (x::'a).\ t \neq x$
  ⟨*proof*⟩

lemmas *dlo-simps*[*noatp*] = *order-refl less-irrefl not-less not-le exists-neq*
  le-less neq-iff linear less-not-permute

lemma *axiom*[*noatp*]: *dense-linear-order* (*op* ≤) (*op* <) ⟨*proof*⟩
lemma *atoms*[*noatp*]:
  shows *TERM* (*less* :: *'a* ⇒ -)
    and *TERM* (*less-eq* :: *'a* ⇒ -)
    and *TERM* (*op* = :: *'a* ⇒ -) ⟨*proof*⟩

declare *axiom*[*langford qe*: *dlo-qe-bnds dlo-qe-nolb dlo-qe-noub gather*: *gather-start*
gather-simps atoms: *atoms*]
declare *dlo-simps*[*langfordsimp*]

end

lemma *dnf*[*noatp*]:
  $(P\ \&\ (Q \mid R)) = ((P\&Q) \mid (P\&R))$
  $((Q \mid R)\ \&\ P) = ((Q\&P) \mid (R\&P))$
  ⟨*proof*⟩

lemmas *weak-dnf-simps*[*noatp*] = *simp-thms dnf*

lemma *nnf-simps*[*noatp*]:
    $(\neg(P \wedge Q)) = (\neg P \vee \neg Q)\ (\neg(P \vee Q)) = (\neg P \wedge \neg Q)\ (P \longrightarrow Q) = (\neg P \vee Q)$
    $(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg\ Q))\ (\neg\ \neg(P)) = P$
  ⟨*proof*⟩

lemma *ex-distrib*[*noatp*]: $(\exists\, x.\ P\ x \vee Q\ x) \longleftrightarrow ((\exists\, x.\ P\ x) \vee (\exists\, x.\ Q\ x))$ ⟨*proof*⟩

lemmas *dnf-simps*[*noatp*] = *weak-dnf-simps nnf-simps ex-distrib*

⟨*ML*⟩

23 Contructive dense linear orders yield QE for linear arithmetic over ordered Fields

Linear order without upper bounds

locale *linorder-stupid-syntax* = *linorder*
begin
notation
less-eq (*op* $\sqsubseteq$) **and**
less-eq ((-/ $\sqsubseteq$ -) [*51*, *51*] *50*) **and**
less (*op* $\sqsubset$) **and**
less ((-/ $\sqsubset$ -) [*51*, *51*] *50*)

end

locale *linorder-no-ub* = *linorder-stupid-syntax* +
assumes *gt-ex*: $\exists\, y.$ *less x y*
begin
lemma *ge-ex*[*noatp*]: $\exists\, y.\ x \sqsubseteq y$ ⟨*proof*⟩

Theorems for $\exists\, z.\ \forall\, x.\ z \sqsubset x \longrightarrow (P\ x \longleftrightarrow P_{+\infty})$

lemma *pinf-conj*[*noatp*]:
assumes *ex1*: $\exists\, z1.\ \forall\, x.\ z1 \sqsubset x \longrightarrow (P1\ x \longleftrightarrow P1')$
and *ex2*: $\exists\, z2.\ \forall\, x.\ z2 \sqsubset x \longrightarrow (P2\ x \longleftrightarrow P2')$
shows $\exists\, z.\ \forall\, x.\ z \sqsubset x \longrightarrow ((P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2'))$
⟨*proof*⟩

lemma *pinf-disj*[*noatp*]:
assumes *ex1*: $\exists\, z1.\ \forall\, x.\ z1 \sqsubset x \longrightarrow (P1\ x \longleftrightarrow P1')$
and *ex2*: $\exists\, z2.\ \forall\, x.\ z2 \sqsubset x \longrightarrow (P2\ x \longleftrightarrow P2')$
shows $\exists\, z.\ \forall\, x.\ z \sqsubset x \longrightarrow ((P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2'))$
⟨*proof*⟩

lemma *pinf-ex*[*noatp*]: **assumes** *ex*:$\exists\, z.\ \forall\, x.\ z \sqsubset x \longrightarrow (P\ x \longleftrightarrow P1)$ **and** *p1*: *P1* **shows** $\exists\ x.\ P\ x$
⟨*proof*⟩

end

Linear order without upper bounds

locale *linorder-no-lb* = *linorder-stupid-syntax* +
assumes *lt-ex*: $\exists\, y.$ *less y x*
begin
lemma *le-ex*[*noatp*]: $\exists\, y.\ y \sqsubseteq x$ ⟨*proof*⟩

Theorems for $\exists\, z.\ \forall\, x.\ x \sqsubset z \longrightarrow (P\ x \longleftrightarrow P_{-\infty})$

lemma *minf-conj*[*noatp*]:
assumes *ex1*: $\exists\, z1.\ \forall\, x.\ x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$
and *ex2*: $\exists\, z2.\ \forall\, x.\ x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$

shows $\exists z.\ \forall x.\ x \sqsubset\ z \longrightarrow ((P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2'))$
⟨*proof*⟩

lemma *minf-disj*[*noatp*]:
assumes *ex1*: $\exists z1.\ \forall x.\ x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$
and *ex2*: $\exists z2.\ \forall x.\ x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$
shows $\exists z.\ \forall x.\ x \sqsubset\ z \longrightarrow ((P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2'))$
⟨*proof*⟩

lemma *minf-ex*[*noatp*]: **assumes** *ex*:$\exists z.\ \forall x.\ x \sqsubset z \longrightarrow (P\ x \longleftrightarrow P1)$ **and** *p1*: *P1* **shows** $\exists\ x.\ P\ x$
⟨*proof*⟩

end

locale *constr-dense-linear-order* = *linorder-no-lb* + *linorder-no-ub* +
fixes *between*
assumes *between-less*: *less x y* $\Longrightarrow$ *less x* (*between x y*) $\wedge$ *less* (*between x y*) *y*
and *between-same*: *between x x* = *x*

sublocale *constr-dense-linear-order* < *dense-linear-order*
⟨*proof*⟩

context *constr-dense-linear-order*
begin

lemma *rinf-U*[*noatp*]:
assumes *fU*: *finite U*
and *lin-dense*: $\forall x\ l\ u.\ (\forall\ t.\ l \sqsubset t \wedge t \sqsubset\ u \longrightarrow t \notin U) \wedge l \sqsubset x \wedge x \sqsubset u \wedge P\ x$
$\longrightarrow (\forall\ y.\ l \sqsubset y \wedge y \sqsubset u \longrightarrow P\ y\ )$
and *nmpiU*: $\forall x.\ \neg\ MP \wedge \neg PP \wedge P\ x \longrightarrow (\exists\ u \in\ U.\ \exists\ u' \in\ U.\ u \sqsubseteq x \wedge x \sqsubseteq u')$
and *nmi*: $\neg\ MP$ **and** *npi*: $\neg\ PP$ **and** *ex*: $\exists\ x.\ \ P\ x$
shows $\exists\ u \in\ U.\ \exists\ u' \in\ U.\ P\ (between\ u\ u')$
⟨*proof*⟩
term *linorder.Min less-eq*
⟨*proof*⟩

theorem *fr-eq*[*noatp*]:
assumes *fU*: *finite U*
and *lin-dense*: $\forall x\ l\ u.\ (\forall\ t.\ l \sqsubset t \wedge t \sqsubset\ u \longrightarrow t \notin U) \wedge l \sqsubset x \wedge x \sqsubset u \wedge P\ x$
$\longrightarrow (\forall\ y.\ l \sqsubset y \wedge y \sqsubset u \longrightarrow P\ y\ )$
and *nmibnd*: $\forall x.\ \neg\ MP \wedge P\ x \longrightarrow (\exists\ u \in\ U.\ u \sqsubseteq x)$
and *npibnd*: $\forall x.\ \neg PP \wedge P\ x \longrightarrow (\exists\ u \in\ U.\ x \sqsubseteq u)$
and *mi*: $\exists z.\ \forall x.\ x \sqsubset z \longrightarrow (P\ x = MP)$ **and** *pi*: $\exists z.\ \forall x.\ z \sqsubset x \longrightarrow (P\ x = PP)$
shows $(\exists\ x.\ P\ x) \equiv (MP \vee PP \vee (\exists\ u \in U.\ \exists\ u' \in U.\ P\ (between\ u\ u')))$
(**is** - $\equiv$ (- $\vee$ - $\vee$ *?F*) **is** *?E* $\equiv$ *?D*)

⟨*proof*⟩

lemmas *minf-thms*[*noatp*] = *minf-conj minf-disj minf-eq minf-neq minf-lt minf-le minf-gt minf-ge minf-P*
lemmas *pinf-thms*[*noatp*] = *pinf-conj pinf-disj pinf-eq pinf-neq pinf-lt pinf-le pinf-gt pinf-ge pinf-P*

lemmas *nmi-thms*[*noatp*] = *nmi-conj nmi-disj nmi-eq nmi-neq nmi-lt nmi-le nmi-gt nmi-ge nmi-P*
lemmas *npi-thms*[*noatp*] = *npi-conj npi-disj npi-eq npi-neq npi-lt npi-le npi-gt npi-ge npi-P*
lemmas *lin-dense-thms*[*noatp*] = *lin-dense-conj lin-dense-disj lin-dense-eq lin-dense-neq lin-dense-lt lin-dense-le lin-dense-gt lin-dense-ge lin-dense-P*

lemma *ferrack-axiom*[*noatp*]: *constr-dense-linear-order less-eq less between*
⟨*proof*⟩
lemma *atoms*[*noatp*]:
shows *TERM* (*less* :: *'a* ⇒ -)
and *TERM* (*less-eq* :: *'a* ⇒ -)
and *TERM* (*op* = :: *'a* ⇒ -) ⟨*proof*⟩

declare *ferrack-axiom* [*ferrack minf*: *minf-thms pinf*: *pinf-thms*
nmi: *nmi-thms npi*: *npi-thms lindense*:
lin-dense-thms qe: *fr-eq atoms*: *atoms*]

⟨*ML*⟩

end

⟨*ML*⟩

23.1 Ferrante and Rackoff algorithm over ordered fields

lemma *neg-prod-lt*:(*c*::*'a*::*ordered-field*) < *0* ⟹ ((*c*∗*x* < *0*) == (*x* > *0*))
⟨*proof*⟩

lemma *pos-prod-lt*:(*c*::*'a*::*ordered-field*) > *0* ⟹ ((*c*∗*x* < *0*) == (*x* < *0*))
⟨*proof*⟩

lemma *neg-prod-sum-lt*: (*c*::*'a*::*ordered-field*) < *0* ⟹ ((*c*∗*x* + *t*< *0*) == (*x* > (− *1*/*c*)∗*t*))
⟨*proof*⟩

lemma *pos-prod-sum-lt*:(*c*::*'a*::*ordered-field*) > *0* ⟹ ((*c*∗*x* + *t* < *0*) == (*x* < (− *1*/*c*)∗*t*))
⟨*proof*⟩

lemma *sum-lt*:((*x*::*'a*::*pordered-ab-group-add*) + *t* < *0*) == (*x* < − *t*)
⟨*proof*⟩

lemma *neg-prod-le*:$(c::'a::ordered\text{-}field) < 0 \Longrightarrow ((c*x <= 0) == (x >= 0))$
⟨*proof*⟩

lemma *pos-prod-le*:$(c::'a::ordered\text{-}field) > 0 \Longrightarrow ((c*x <= 0) == (x <= 0))$
⟨*proof*⟩

lemma *neg-prod-sum-le*: $(c::'a::ordered\text{-}field) < 0 \Longrightarrow ((c*x + t <= 0) == (x >= (-1/c)*t))$
⟨*proof*⟩

lemma *pos-prod-sum-le*:$(c::'a::ordered\text{-}field) > 0 \Longrightarrow ((c*x + t <= 0) == (x <= (-1/c)*t))$
⟨*proof*⟩

lemma *sum-le*:$((x::'a::pordered\text{-}ab\text{-}group\text{-}add) + t <= 0) == (x <= -t)$
⟨*proof*⟩

lemma *nz-prod-eq*:$(c::'a::ordered\text{-}field) \neq 0 \Longrightarrow ((c*x = 0) == (x = 0))$ ⟨*proof*⟩
lemma *nz-prod-sum-eq*: $(c::'a::ordered\text{-}field) \neq 0 \Longrightarrow ((c*x + t = 0) == (x = (-1/c)*t))$
⟨*proof*⟩
lemma *sum-eq*:$((x::'a::pordered\text{-}ab\text{-}group\text{-}add) + t = 0) == (x = -t)$
⟨*proof*⟩

interpretation *class-ordered-field-dense-linear-order*: *constr-dense-linear-order*
op <= *op <*
$\lambda\ x\ y.\ 1/2 * ((x::'a::\{ordered\text{-}field, recpower, number\text{-}ring\}) + y)$
⟨*proof*⟩

⟨*ML*⟩

lemma *upper-bound-finite-set*:
assumes *fS*: *finite S*
shows $\exists (a::'a::linorder).\ \forall x \in S.\ f\ x \leq a$
⟨*proof*⟩

lemma *lower-bound-finite-set*:
assumes *fS*: *finite S*
shows $\exists (a::'a::linorder).\ \forall x \in S.\ f\ x \geq a$
⟨*proof*⟩

lemma *bound-finite-set*: **assumes** *f*: *finite S*
shows $\exists\, a.\ \forall x \in S.\ (f\ x::\ 'a::linorder) \leq a$
⟨*proof*⟩

end

24 Finite-Cartesian-Product: Definition of finite Cartesian product types.

theory *Finite-Cartesian-Product*
imports *Main*
begin

definition *hassize* (**infixr** *hassize 12*) **where**
(*S hassize n*) = (*finite S* ∧ *card S* = *n*)

lemma *hassize-image-inj*: **assumes** *f*: *inj-on f S* **and** *S*: *S hassize n*
shows *f* ‘ *S hassize n*
⟨*proof*⟩

24.1 Finite Cartesian products, with indexing and lambdas.

typedef (**open** *Cart*)
('*a*, '*b*) ^ (**infixl** ^ *15*)
= *UNIV* :: ('*b* ⇒ '*a*) *set*
morphisms *Cart-nth Cart-lambda* ⟨*proof*⟩

notation *Cart-nth* (**infixl** $ *90*)

notation (*xsymbols*) *Cart-lambda* (**binder** χ *10*)

lemma *stupid-ext*: (∀ *x*. *f x* = *g x*) ⟷ (*f* = *g*)
⟨*proof*⟩

lemma *Cart-eq*: ((*x*:: '*a* ^ '*b*) = *y*) ⟷ (∀ *i*. *x*$*i* = *y*$*i*)
⟨*proof*⟩

lemma *Cart-lambda-beta* [*simp*]: *Cart-lambda g* $ *i* = *g i*
⟨*proof*⟩

lemma *Cart-lambda-unique*:
fixes *f* :: '*a* ^ '*b*
shows (∀ *i*. *f*$*i* = *g i*) ⟷ *Cart-lambda g* = *f*
⟨*proof*⟩

lemma *Cart-lambda-eta*: (χ *i*. (*g*$*i*)) = *g*
⟨*proof*⟩

A non-standard sum to "paste" Cartesian products.

definition *pastecart* :: '*a* ^ '*m* ⇒ '*a* ^ '*n* ⇒ '*a* ^ ('*m* + '*n*) **where**
pastecart f g = (χ *i*. *case i of Inl a* ⇒ *f*$*a* | *Inr b* ⇒ *g*$*b*)

definition *fstcart*:: *'a ^('m + 'n) ⇒ 'a ^ 'm* **where**
fstcart f = (χ i. (f$(Inl i)))

definition *sndcart*:: *'a ^('m + 'n) ⇒ 'a ^ 'n* **where**
sndcart f = (χ i. (f$(Inr i)))

lemma *nth-pastecart-Inl* [*simp*]: *pastecart f g $ Inl a = f$a*
⟨*proof*⟩

lemma *nth-pastecart-Inr* [*simp*]: *pastecart f g $ Inr b = g$b*
⟨*proof*⟩

lemma *nth-fstcart* [*simp*]: *fstcart f $ i = f $ Inl i*
⟨*proof*⟩

lemma *nth-sndtcart* [*simp*]: *sndcart f $ i = f $ Inr i*
⟨*proof*⟩

lemma *finite-sum-image*: *(UNIV::('a + 'b) set) = range Inl ∪ range Inr*
⟨*proof*⟩

lemma *fstcart-pastecart*: *fstcart (pastecart (x::'a ^'m) (y:: 'a ^ 'n)) = x*
⟨*proof*⟩

lemma *sndcart-pastecart*: *sndcart (pastecart (x::'a ^'m) (y:: 'a ^ 'n)) = y*
⟨*proof*⟩

lemma *pastecart-fst-snd*: *pastecart (fstcart z) (sndcart z) = z*
⟨*proof*⟩

lemma *pastecart-eq*: *(x = y) ⟷ (fstcart x = fstcart y) ∧ (sndcart x = sndcart y)*
⟨*proof*⟩

lemma *forall-pastecart*: *(∀ p. P p) ⟷ (∀ x y. P (pastecart x y))*
⟨*proof*⟩

lemma *exists-pastecart*: *(∃ p. P p) ⟷ (∃ x y. P (pastecart x y))*
⟨*proof*⟩

end

25 Glbs: Definitions of Lower Bounds and Greatest Lower Bounds, analogous to Lubs

theory *Glbs*

imports *Lubs*
begin

definition
greatestP :: [*'a* =>*bool*,*'a*::*ord*] => *bool* **where**
greatestP P x = (*P x* & *Collect P* ∗<= *x*)

definition
isLb :: [*'a set*, *'a set*, *'a*::*ord*] => *bool* **where**
isLb R S x = (*x* <=∗ *S* & *x*: *R*)

definition
isGlb :: [*'a set*, *'a set*, *'a*::*ord*] => *bool* **where**
isGlb R S x = *greatestP* (*isLb R S*) *x*

definition
lbs :: [*'a set*, *'a*::*ord set*] => *'a set* **where**
lbs R S = *Collect* (*isLb R S*)

25.1 Rules about the Operators *greatestP*, *isLb* and *isGlb*

lemma *leastPD1*: *greatestP P x* ==> *P x*
⟨*proof*⟩

lemma *greatestPD2*: *greatestP P x* ==> *Collect P* ∗<= *x*
⟨*proof*⟩

lemma *greatestPD3*: [| *greatestP P x*; *y*: *Collect P* |] ==> *x* >= *y*
⟨*proof*⟩

lemma *isGlbD1*: *isGlb R S x* ==> *x* <=∗ *S*
⟨*proof*⟩

lemma *isGlbD1a*: *isGlb R S x* ==> *x*: *R*
⟨*proof*⟩

lemma *isGlb-isLb*: *isGlb R S x* ==> *isLb R S x*
⟨*proof*⟩

lemma *isGlbD2*: [| *isGlb R S x*; *y* : *S* |] ==> *y* >= *x*
⟨*proof*⟩

lemma *isGlbD3*: *isGlb R S x* ==> *greatestP*(*isLb R S*) *x*
⟨*proof*⟩

lemma *isGlbI1*: *greatestP*(*isLb R S*) *x* ==> *isGlb R S x*
⟨*proof*⟩

lemma *isGlbI2*: [| *isLb R S x*; *Collect* (*isLb R S*) ∗<= *x* |] ==> *isGlb R S x*

⟨*proof*⟩

lemma *isLbD*: [| *isLb R S x*; *y* : *S* |] ==> *y* >= *x*
⟨*proof*⟩

lemma *isLbD2*: *isLb R S x* ==> *x* <=* *S*
⟨*proof*⟩

lemma *isLbD2a*: *isLb R S x* ==> *x*: *R*
⟨*proof*⟩

lemma *isLbI*: [| *x* <=* *S* ; *x*: *R* |] ==> *isLb R S x*
⟨*proof*⟩

lemma *isGlb-le-isLb*: [| *isGlb R S x*; *isLb R S y* |] ==> *x* >= *y*
⟨*proof*⟩

lemma *isGlb-ubs*: *isGlb R S x* ==> *lbs R S* *<= *x*
⟨*proof*⟩

end

26 Infinite-Set: Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

26.1 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* $S \Longrightarrow$ *infinite* $(S \cup T)$
⟨*proof*⟩

lemma *infinite-super*:
assumes T: $S \subseteq T$ **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* [*simp*]: *infinite* $(UNIV$ :: *nat set*$)$
⟨*proof*⟩

lemma *nat-not-finite* [*elim*]: *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* ⇒ *'a*) ⟹ *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* ≠ *f y*
shows *inj f*
⟨*proof*⟩

lemma *infinite-countable-subset*:
assumes *inf*: *infinite* (*S*::*'a set*)
shows ∃*f*. *inj* (*f*::*nat* ⇒ *'a*) ∧ *range f* ⊆ *S*
⟨*proof*⟩

lemma *infinite-iff-countable-subset*:
infinite S = (∃*f*. *inj* (*f*::*nat* ⇒ *'a*) ∧ *range f* ⊆ *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 ∃ *y* ∈ *f*`*A*. *infinite* (*f* −` {*y*})
⟨*proof*⟩

lemma *inf-img-fin-domE*:
assumes *finite* (*f*`*A*) **and** *infinite A*
obtains *y* **where** *y* ∈ *f*`*A* **and** *infinite* (*f* −` {*y*})
⟨*proof*⟩

26.2 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**
$Inf\text{-}many\ P = infinite\ \{x.\ P\ x\}$

definition
Alm-all :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** *MOST* *10*) **where**
$Alm\text{-}all\ P = (\neg\ (INFM\ x.\ \neg\ P\ x))$

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-EX*:
$(\exists_\infty x.\ P\ x) \Longrightarrow (\exists x.\ P\ x)$
⟨*proof*⟩

lemma *MOST-iff-finiteNeg*: $(\forall_\infty x.\ P\ x) = finite\ \{x.\ \neg\ P\ x\}$
⟨*proof*⟩

lemma *ALL-MOST*: $\forall x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$
⟨*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 *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-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 *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-nat*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall\, m.\ \exists\, n.\ m<n \wedge P\ n)$
⟨*proof*⟩

lemma *INFM-nat-le*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall\, m.\ \exists\, n.\ m\leq n \wedge P\ n)$
⟨*proof*⟩

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n::nat)) = (\exists\, m.\ \forall\, n.\ m<n \longrightarrow P\ n)$
⟨*proof*⟩

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n::nat)) = (\exists\, m.\ \forall\, n.\ m\leq n \longrightarrow P\ n)$
⟨*proof*⟩

26.3 Enumeration of an Infinite Set

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

consts
enumerate :: $'a::wellorder\ set \Rightarrow (nat \Rightarrow 'a::wellorder)$
primrec
enumerate-0: $enumerate\ S\ 0 = (LEAST\ n.\ n \in S)$
enumerate-Suc: $enumerate\ S\ (Suc\ n) = enumerate\ (S - \{LEAST\ n.\ n \in S\})\ n$

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

lemma *enumerate-in-set*: $infinite\ S \Longrightarrow enumerate\ S\ n : S$
⟨*proof*⟩

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

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

lemma *enumerate-mono*: $m<n \Longrightarrow infinite\ S \Longrightarrow enumerate\ S\ m < enumerate\ S$

n
⟨*proof*⟩

26.4 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 $S = (\forall x\ y.\ x{\in}S \wedge y{\in}S \longrightarrow x{=}y)$

lemma *atmost-one-empty*: $S = \{\} \Longrightarrow$ *atmost-one* S
⟨*proof*⟩

lemma *atmost-one-singleton*: $S = \{x\} \Longrightarrow$ *atmost-one* S
⟨*proof*⟩

lemma *atmost-one-unique* [*elim*]: *atmost-one* $S \Longrightarrow x \in S \Longrightarrow y \in S \Longrightarrow y = x$
⟨*proof*⟩

end

27 Numeral-Type: Numeral Syntax for Types

theory *Numeral-Type*
imports *Main*
begin

27.1 Preliminary lemmas

lemma (**in** *type-definition*) *univ*:
UNIV = *Abs* ' *A*
⟨*proof*⟩

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

27.2 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*]: $CARD('a \times 'b) = CARD('a::finite) * CARD('b::finite)$
⟨*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*]: $CARD('a\ set) = 2\ \hat{}\ CARD('a::finite)$
⟨*proof*⟩

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

27.3 Classes with at least 1 and 2

Class finite already captures "at least 1"

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

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

Class for cardinality "at least 2"

class *card2* = *finite* +
assumes *two-le-card*: $2 \leq CARD('a)$

lemma *one-less-card*: $Suc\ 0 < CARD('a::card2)$
⟨*proof*⟩

lemma *one-less-int-card*: $1 < int\ CARD('a::card2)$
⟨*proof*⟩

27.4 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*⟩

27.5 Locale for modular arithmetic subtypes

locale *mod-type* =
fixes *n* :: *int*
and *Rep* :: '*a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*,*power*} ⇒ *int*
and *Abs* :: *int* ⇒ '*a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*,*power*}
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*)
and *power-def*: *x* ^ *k* = *Abs* (*Rep x* ^ *k mod n*)
begin

lemma *size0*: *0* < *n*
⟨*proof*⟩

lemmas *definitions* =
zero-def one-def add-def mult-def minus-def diff-def power-def

lemma *Rep-less-n*: *Rep x* < *n*
⟨*proof*⟩

lemma *Rep-le-n*: *Rep x* ≤ *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-ring-1-class*)
⟨*proof*⟩

lemma *recpower*: *OFCLASS*('*a*, *recpower-class*)
⟨*proof*⟩

end

locale *mod-ring* = *mod-type* +
constrains n :: *int*
and *Rep* :: '*a*::{*number-ring*,*power*} ⇒ *int*
and *Abs* :: *int* ⇒ '*a*::{*number-ring*,*power*}
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*: ⋀*z*. ⟦(*x*::'*a*) = *of-int z*; *0* ≤ *z*; *z* < *n*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *induct*:
(⋀*z*. ⟦*0* ≤ *z*; *z* < *n*⟧ ⟹ *P* (*of-int z*)) ⟹ *P* (*x*::'*a*)
⟨*proof*⟩

end

27.6 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*,*recpower*}
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*,*power*}
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 $x - y = \mathit{Abs\text{-}bit0'}\ (\mathit{Rep\text{-}bit0}\ x - \mathit{Rep\text{-}bit0}\ y)$
definition $- x = \mathit{Abs\text{-}bit0'}\ (- \mathit{Rep\text{-}bit0}\ x)$
definition $x\ \hat{}\ k = \mathit{Abs\text{-}bit0'}\ (\mathit{Rep\text{-}bit0}\ x\ \hat{}\ k)$

definition $0 = \mathit{Abs\text{-}bit1}\ 0$
definition $1 = \mathit{Abs\text{-}bit1}\ 1$
definition $x + y = \mathit{Abs\text{-}bit1'}\ (\mathit{Rep\text{-}bit1}\ x + \mathit{Rep\text{-}bit1}\ y)$
definition $x * y = \mathit{Abs\text{-}bit1'}\ (\mathit{Rep\text{-}bit1}\ x * \mathit{Rep\text{-}bit1}\ y)$
definition $x - y = \mathit{Abs\text{-}bit1'}\ (\mathit{Rep\text{-}bit1}\ x - \mathit{Rep\text{-}bit1}\ y)$
definition $- x = \mathit{Abs\text{-}bit1'}\ (- \mathit{Rep\text{-}bit1}\ x)$
definition $x\ \hat{}\ k = \mathit{Abs\text{-}bit1'}\ (\mathit{Rep\text{-}bit1}\ x\ \hat{}\ k)$

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*,*recpower*}
  ⟨*proof*⟩

instance *bit1* :: (*finite*) {*comm-ring-1*,*recpower*}
  ⟨*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*

declare *power-Suc* [**where** *?'a='a::finite bit0*, *standard*, *simp*]
declare *power-Suc* [**where** *?'a='a::finite bit1*, *standard*, *simp*]

27.7 Syntax

syntax
-NumeralType :: *num-const* => *type* (-)
-NumeralType0 :: *type* (*0*)
-NumeralType1 :: *type* (*1*)

translations
-NumeralType1 == (*type*) *num1*
-NumeralType0 == (*type*) *num0*

⟨*ML*⟩

27.8 Examples

lemma *CARD(0)* = *0* ⟨*proof*⟩
lemma *CARD(17)* = *17* ⟨*proof*⟩
lemma *8 * 11 ^ 3 - 6* = (*2::5*) ⟨*proof*⟩

end

28 FrechetDeriv: Frechet Derivative

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

definition
fderiv ::
[*'a*::*real-normed-vector* ⇒ *'b*::*real-normed-vector*, *'a*, *'a* ⇒ *'b*] ⇒ *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*⟩

28.1 Addition

lemma *add-diff-add*:
fixes *a b c d* :: *'a*::*ab-group-add*
shows (*a* + *c*) − (*b* + *d*) = (*a* − *b*) + (*c* − *d*)
⟨*proof*⟩

lemma *bounded-linear-add*:
assumes *bounded-linear f*
assumes *bounded-linear g*
shows *bounded-linear* (λ*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*⟩

28.2 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*⟩

28.3 Continuity

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

28.4 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* ⇒ *'b*::*real-normed-vector*

fixes $g :: 'b::real\text{-}normed\text{-}vector \Rightarrow 'c::real\text{-}normed\text{-}vector$
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*⟩

28.5 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 $x :: 'd::real\text{-}normed\text{-}vector$
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*

28.6 Powers

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

28.7 Inverse

lemma *inverse-diff-inverse*:
$[\![(a::'a::division\text{-}ring) \neq 0;\ b \neq 0]\!]$
$\Longrightarrow inverse\ a - inverse\ b = -\ (inverse\ a * (a - b) * inverse\ b)$
⟨*proof*⟩

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*⟩

28.8 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

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

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

29.1 Real inner product spaces

class *real-inner* = *real-vector* + *sgn-div-norm* +
fixes *inner* :: $'a \Rightarrow 'a \Rightarrow real$
assumes *inner-commute*: *inner x y* = *inner y x*
and *inner-left-distrib*: *inner* $(x + y)$ *z* = *inner x z* + *inner y z*
and *inner-scaleR-left*: *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* = $0 \longleftrightarrow x = 0$
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-right-distrib*: *inner x* $(y + z) = inner\ x\ y + inner\ x\ z$
⟨*proof*⟩

lemma *inner-scaleR-right*: *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-distrib = inner-left-distrib inner-right-distrib*
lemmas *inner-diff = inner-diff-left inner-diff-right*
lemmas *inner-scaleR = inner-scaleR-left inner-scaleR-right*

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

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

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

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

29.2 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

29.3 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*⟧
⟹ *GDERIV* (λ*x*. *g* (*f x*)) *x* :> *scaleR dg df*
⟨*proof*⟩

lemma *FDERIV-subst*: ⟦*FDERIV f x* :> *df*; *df* = *d*⟧ ⟹ *FDERIV f x* :> *d*
⟨*proof*⟩

lemma *GDERIV-subst*: ⟦*GDERIV f x* :> *df*; *df* = *d*⟧ ⟹ *GDERIV f x* :> *d*
⟨*proof*⟩

lemma *GDERIV-const*: *GDERIV* (λ*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

30 Euclidean-Space: (Real) Vectors in Euclidean space, and elementary linear algebra.

theory *Euclidean-Space*
imports
Complex-Main ~~/*src*/*HOL*/*Decision-Procs*/*Dense-Linear-Order*
Finite-Cartesian-Product Glbs Infinite-Set Numeral-Type
Inner-Product
uses (*normarith.ML*)
begin

Some common special cases.

lemma *forall-1*: $(\forall\, i{::}1.\ P\ i) \longleftrightarrow P\ 1$
⟨*proof*⟩

lemma *exhaust-2*:
fixes $x :: 2$ **shows** $x = 1 \vee x = 2$
⟨*proof*⟩

lemma *forall-2*: $(\forall\, i{::}2.\ P\ i) \longleftrightarrow P\ 1 \wedge P\ 2$
⟨*proof*⟩

lemma *exhaust-3*:
fixes $x :: 3$ **shows** $x = 1 \vee x = 2 \vee x = 3$
⟨*proof*⟩

lemma *forall-3*: $(\forall\, i{::}3.\ P\ i) \longleftrightarrow P\ 1 \wedge P\ 2 \wedge P\ 3$
⟨*proof*⟩

lemma *UNIV-1*: $UNIV = \{1{::}1\}$
⟨*proof*⟩

lemma *UNIV-2*: $UNIV = \{1{::}2,\ 2{::}2\}$
⟨*proof*⟩

lemma *UNIV-3*: $UNIV = \{1{::}3,\ 2{::}3,\ 3{::}3\}$
⟨*proof*⟩

lemma *setsum-1*: $setsum\ f\ (UNIV{::}1\ set) = f\ 1$
⟨*proof*⟩

lemma *setsum-2*: $setsum\ f\ (UNIV{::}2\ set) = f\ 1 + f\ 2$
⟨*proof*⟩

lemma *setsum-3*: $setsum\ f\ (UNIV{::}3\ set) = f\ 1 + f\ 2 + f\ 3$
⟨*proof*⟩

30.1 Basic componentwise operations on vectors.

instantiation $\hat{}$ :: (*plus*,*type*) *plus*
begin
definition *vector-add-def* : $op + \equiv (\lambda\ x\ y.\ \ (\chi\ i.\ (x\$i) + (y\$i)))$
instance ⟨*proof*⟩
end

instantiation $\hat{}$ :: (*times*,*type*) *times*
begin
definition *vector-mult-def* : $op * \equiv (\lambda\ x\ y.\ \ (\chi\ i.\ (x\$i) * (y\$i)))$
instance ⟨*proof*⟩
end

instantiation ^ :: (*minus*,*type*) *minus* **begin**
definition *vector-minus-def* : *op* − ≡ (λ *x y*. (χ *i*. (*x*$*i*) − (*y*$*i*)))
instance ⟨*proof*⟩
end

instantiation ^ :: (*uminus*,*type*) *uminus* **begin**
definition *vector-uminus-def* : *uminus* ≡ (λ *x*. (χ *i*. − (*x*$*i*)))
instance ⟨*proof*⟩
end
instantiation ^ :: (*zero*,*type*) *zero* **begin**
definition *vector-zero-def* : *0* ≡ (χ *i*. *0*)
instance ⟨*proof*⟩
end

instantiation ^ :: (*one*,*type*) *one* **begin**
definition *vector-one-def* : *1* ≡ (χ *i*. *1*)
instance ⟨*proof*⟩
end

instantiation ^ :: (*ord*,*type*) *ord*
begin
definition *vector-less-eq-def*:
less-eq (*x* :: *'a* ^*'b*) *y* = (*ALL i*. *x*$*i* <= *y*$*i*)
definition *vector-less-def*: *less* (*x* :: *'a* ^*'b*) *y* = (*ALL i*. *x*$*i* < *y*$*i*)

instance ⟨*proof*⟩
end

instantiation ^ :: (*scaleR*, *type*) *scaleR*
begin
definition *vector-scaleR-def*: *scaleR* = (λ *r x*. (χ *i*. *scaleR r* (*x*$*i*)))
instance ⟨*proof*⟩
end

Also the scalar-vector multiplication.

definition *vector-scalar-mult*:: *'a*::*times* ⇒ *'a* ^*'n* ⇒ *'a* ^ *'n* (**infixr** ∗*s* *75*)
where *c* ∗*s* *x* = (χ *i*. *c* ∗ (*x*$*i*))

Constant Vectors

definition *vec x* = (χ *i*. *x*)

Dot products.

definition *dot* :: *'a*::{*comm-monoid-add*, *times*} ^ *'n* ⇒ *'a* ^ *'n* ⇒ *'a* (**infix** · *70*)
where
x · *y* = *setsum* (λ*i*. *x*$*i* ∗ *y*$*i*) *UNIV*

lemma *dot-1*[*simp*]: (*x*::*'a*::{*comm-monoid-add*, *times*} ^*1*) · *y* = (*x*$*1*) ∗ (*y*$*1*)
⟨*proof*⟩

lemma *dot-2*[*simp*]: $(x::'a::\{comm\text{-}monoid\text{-}add, times\}\ \hat{}\ 2) \cdot y = (x\$1) * (y\$1) + (x\$2) * (y\$2)$
⟨*proof*⟩

lemma *dot-3*[*simp*]: $(x::'a::\{comm\text{-}monoid\text{-}add, times\}\ \hat{}\ 3) \cdot y = (x\$1) * (y\$1) + (x\$2) * (y\$2) + (x\$3) * (y\$3)$
⟨*proof*⟩

30.2 A naive proof procedure to lift really trivial arithmetic stuff from the basis of the vector space.

⟨*ML*⟩

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

Obvious "component-pushing".

lemma *vec-component* [*simp*]: $(vec\ x :: 'a\ \hat{}\ 'n)\$i = x$
⟨*proof*⟩

lemma *vector-add-component* [*simp*]:
fixes $x\ y :: 'a::\{plus\}\ \hat{}\ 'n$
shows $(x + y)\$i = x\$i + y\$i$
⟨*proof*⟩

lemma *vector-minus-component* [*simp*]:
fixes $x\ y :: 'a::\{minus\}\ \hat{}\ 'n$
shows $(x - y)\$i = x\$i - y\$i$
⟨*proof*⟩

lemma *vector-mult-component* [*simp*]:
fixes $x\ y :: 'a::\{times\}\ \hat{}\ 'n$
shows $(x * y)\$i = x\$i * y\$i$
⟨*proof*⟩

lemma *vector-smult-component* [*simp*]:
fixes $y :: 'a::\{times\}\ \hat{}\ 'n$
shows $(c *s y)\$i = c * (y\$i)$
⟨*proof*⟩

lemma *vector-uminus-component* [*simp*]:
fixes $x :: 'a::\{uminus\}\ \hat{}\ 'n$
shows $(- x)\$i = - (x\$i)$
⟨*proof*⟩

lemma *vector-scaleR-component* [*simp*]:
fixes $x :: 'a::scaleR\ \hat{}\ 'n$
shows $(scaleR\ r\ x)\$i = scaleR\ r\ (x\$i)$
⟨*proof*⟩

lemma *cond-component*: (*if b then x else y*)$*i* = (*if b then x*$*i else y*$*i*) ⟨*proof*⟩

lemmas *vector-component* =
vec-component vector-add-component vector-mult-component
vector-smult-component vector-minus-component vector-uminus-component
vector-scaleR-component cond-component

30.3 Some frequently useful arithmetic lemmas over vectors.

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

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

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

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

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

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

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

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

instance ^ :: (*real-vector*, *type*) *real-vector*
⟨*proof*⟩

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

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

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

instance ^ :: (*ab-semigroup-idem-mult*,*type*) *ab-semigroup-idem-mult*
⟨*proof*⟩

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

fun *vector-power* :: ('*a*::{*one*,*times*} ^'*n*) ⇒ *nat* ⇒ '*a*^'*n* **where**
vector-power x 0 = *1*
| *vector-power x* (*Suc n*) = *x* ∗ *vector-power x n*

instantiation ^ :: (*recpower*,*type*) *recpower*
begin
definition *vec-power-def*: *op* ^ ≡ *vector-power*
instance
⟨*proof*⟩
end

instance ^ :: (*semiring*,*type*) *semiring*
⟨*proof*⟩

instance ^ :: (*semiring-0*,*type*) *semiring-0*
⟨*proof*⟩
instance ^ :: (*semiring-1*,*type*) *semiring-1*
⟨*proof*⟩
instance ^ :: (*comm-semiring*,*type*) *comm-semiring*
⟨*proof*⟩

instance ^ :: (*comm-semiring-0*,*type*) *comm-semiring-0* ⟨*proof*⟩
instance ^ :: (*cancel-comm-monoid-add*, *type*) *cancel-comm-monoid-add* ⟨*proof*⟩
instance ^ :: (*semiring-0-cancel*,*type*) *semiring-0-cancel* ⟨*proof*⟩
instance ^ :: (*comm-semiring-0-cancel*,*type*) *comm-semiring-0-cancel* ⟨*proof*⟩
instance ^ :: (*ring*,*type*) *ring* ⟨*proof*⟩
instance ^ :: (*semiring-1-cancel*,*type*) *semiring-1-cancel* ⟨*proof*⟩
instance ^ :: (*comm-semiring-1*,*type*) *comm-semiring-1* ⟨*proof*⟩

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

instance ^ :: (*real-algebra*,*type*) *real-algebra*
⟨*proof*⟩

instance ^ :: (*real-algebra-1*,*type*) *real-algebra-1* ⟨*proof*⟩

lemma *of-nat-index*:
(*of-nat n* :: '*a*::*semiring-1* ^'*n*)$*i* = *of-nat n*
⟨*proof*⟩
lemma *zero-index*[*simp*]:
(*0* :: '*a*::*zero* ^'*n*)$*i* = *0* ⟨*proof*⟩

lemma *one-index*[*simp*]:
(*1* :: '*a*::*one* ^'*n*)$*i* = *1* ⟨*proof*⟩

lemma *one-plus-of-nat-neq-0*: $(1::'a::semiring\text{-}char\text{-}0) + of\text{-}nat\ n \neq 0$
⟨*proof*⟩

instance ^ :: (*semiring-char-0*,*type*) *semiring-char-0*
⟨*proof*⟩

instance ^ :: (*comm-ring-1*,*type*) *comm-ring-1* ⟨*proof*⟩
instance ^ :: (*ring-char-0*,*type*) *ring-char-0* ⟨*proof*⟩

lemma *vector-smult-assoc*: $a *s (b *s x) = ((a::'a::semigroup\text{-}mult) * b) *s x$
⟨*proof*⟩
lemma *vector-sadd-rdistrib*: $((a::'a::semiring) + b) *s x = a *s x + b *s x$
⟨*proof*⟩
lemma *vector-add-ldistrib*: $(c::'a::semiring) *s (x + y) = c *s x + c *s y$
⟨*proof*⟩
lemma *vector-smult-lzero*[*simp*]: $(0::'a::mult\text{-}zero) *s x = 0$ ⟨*proof*⟩
lemma *vector-smult-lid*[*simp*]: $(1::'a::monoid\text{-}mult) *s x = x$ ⟨*proof*⟩
lemma *vector-ssub-ldistrib*: $(c::'a::ring) *s (x - y) = c *s x - c *s y$
⟨*proof*⟩
lemma *vector-smult-rneg*: $(c::'a::ring) *s -x = -(c *s x)$ ⟨*proof*⟩
lemma *vector-smult-lneg*: $-(c::'a::ring) *s x = -(c *s x)$ ⟨*proof*⟩
lemma *vector-sneg-minus1*: $-x = (-(1::'a::ring\text{-}1)) *s x$ ⟨*proof*⟩
lemma *vector-smult-rzero*[*simp*]: $c *s 0 = (0::'a::mult\text{-}zero\ ^\ 'n)$ ⟨*proof*⟩
lemma *vector-sub-rdistrib*: $((a::'a::ring) - b) *s x = a *s x - b *s x$
⟨*proof*⟩

lemma *vec-eq*[*simp*]: $(vec\ m = vec\ n) \longleftrightarrow (m = n)$
⟨*proof*⟩

30.4 Square root of sum of squares

definition
$setL2\ f\ A = sqrt\ (\sum i{\in}A.\ (f\ i)^2)$

lemma *setL2-cong*:
$[\![A = B;\ \bigwedge x.\ x \in B \Longrightarrow f\ x = g\ x]\!] \Longrightarrow setL2\ f\ A = setL2\ g\ B$
⟨*proof*⟩

lemma *strong-setL2-cong*:
$[\![A = B;\ \bigwedge x.\ x \in B =simp=> f\ x = g\ x]\!] \Longrightarrow setL2\ f\ A = setL2\ g\ B$
⟨*proof*⟩

lemma *setL2-infinite* [*simp*]: $\neg\ finite\ A \Longrightarrow setL2\ f\ A = 0$
⟨*proof*⟩

lemma *setL2-empty* [*simp*]: $setL2\ f\ \{\} = 0$
⟨*proof*⟩

lemma *setL2-insert* [*simp*]:

$[\![\mathit{finite}\ F;\ a \notin F]\!] \Longrightarrow$
$\mathit{setL2}\ f\ (\mathit{insert}\ a\ F) = \mathit{sqrt}\ ((f\ a)^2 + (\mathit{setL2}\ f\ F)^2)$
⟨*proof*⟩

lemma *setL2-nonneg* [*simp*]: $0 \leq \mathit{setL2}\ f\ A$
⟨*proof*⟩

lemma *setL2-0'*: $\forall a{\in}A.\ f\ a = 0 \Longrightarrow \mathit{setL2}\ f\ A = 0$
⟨*proof*⟩

lemma *setL2-mono*:
assumes $\bigwedge i.\ i \in K \Longrightarrow f\ i \leq g\ i$
assumes $\bigwedge i.\ i \in K \Longrightarrow 0 \leq f\ i$
shows $\mathit{setL2}\ f\ K \leq \mathit{setL2}\ g\ K$
⟨*proof*⟩

lemma *setL2-right-distrib*:
$0 \leq r \Longrightarrow r * \mathit{setL2}\ f\ A = \mathit{setL2}\ (\lambda x.\ r * f\ x)\ A$
⟨*proof*⟩

lemma *setL2-left-distrib*:
$0 \leq r \Longrightarrow \mathit{setL2}\ f\ A * r = \mathit{setL2}\ (\lambda x.\ f\ x * r)\ A$
⟨*proof*⟩

lemma *setsum-nonneg-eq-0-iff*:
fixes $f :: 'a \Rightarrow 'b$::*pordered-ab-group-add*
shows $[\![\mathit{finite}\ A;\ \forall x{\in}A.\ 0 \leq f\ x]\!] \Longrightarrow \mathit{setsum}\ f\ A = 0 \longleftrightarrow (\forall x{\in}A.\ f\ x = 0)$
⟨*proof*⟩

lemma *setL2-eq-0-iff*: $\mathit{finite}\ A \Longrightarrow \mathit{setL2}\ f\ A = 0 \longleftrightarrow (\forall x{\in}A.\ f\ x = 0)$
⟨*proof*⟩

lemma *setL2-triangle-ineq*:
shows $\mathit{setL2}\ (\lambda i.\ f\ i + g\ i)\ A \leq \mathit{setL2}\ f\ A + \mathit{setL2}\ g\ A$
⟨*proof*⟩

lemma *sqrt-sum-squares-le-sum*:
$[\![0 \leq x;\ 0 \leq y]\!] \Longrightarrow \mathit{sqrt}\ (x^2 + y^2) \leq x + y$
⟨*proof*⟩

lemma *setL2-le-setsum* [*rule-format*]:
$(\forall i{\in}A.\ 0 \leq f\ i) \longrightarrow \mathit{setL2}\ f\ A \leq \mathit{setsum}\ f\ A$
⟨*proof*⟩

lemma *sqrt-sum-squares-le-sum-abs*: $\mathit{sqrt}\ (x^2 + y^2) \leq |x| + |y|$
⟨*proof*⟩

lemma *setL2-le-setsum-abs*: $\mathit{setL2}\ f\ A \leq (\sum i{\in}A.\ |f\ i|)$
⟨*proof*⟩

lemma *setL2-mult-ineq-lemma*:
fixes a b c d :: *real*
shows $2 * (a * c) * (b * d) \leq a^2 * d^2 + b^2 * c^2$
⟨*proof*⟩

lemma *setL2-mult-ineq*: $(\sum i{\in}A.\ |f\ i| * |g\ i|) \leq setL2\ f\ A * setL2\ g\ A$
⟨*proof*⟩

lemma *member-le-setL2*: $[\![finite\ A;\ i \in A]\!] \Longrightarrow f\ i \leq setL2\ f\ A$
⟨*proof*⟩

30.5 Norms

instantiation ^ :: (*real-normed-vector*, *finite*) *real-normed-vector*
begin

definition *vector-norm-def*:
$norm\ (x::'a\hat{}'b) = setL2\ (\lambda i.\ norm\ (x\$i))\ UNIV$

definition *vector-sgn-def*:
$sgn\ (x::'a\hat{}'b) = scaleR\ (inverse\ (norm\ x))\ x$

instance ⟨*proof*⟩

end

30.6 Inner products

instantiation ^ :: (*real-inner*, *finite*) *real-inner*
begin

definition *vector-inner-def*:
$inner\ x\ y = setsum\ (\lambda i.\ inner\ (x\$i)\ (y\$i))\ UNIV$

instance ⟨*proof*⟩

end

30.7 Properties of the dot product.

lemma *dot-sym*: $(x::'a::\{comm\text{-}monoid\text{-}add,\ ab\text{-}semigroup\text{-}mult\}\ \hat{}\ 'n) \cdot y = y \cdot x$
⟨*proof*⟩
lemma *dot-ladd*: $((x::'a::ring\ \hat{}\ 'n) + y) \cdot z = (x \cdot z) + (y \cdot z)$
⟨*proof*⟩
lemma *dot-radd*: $x \cdot (y + (z::'a::ring\ \hat{}\ 'n)) = (x \cdot y) + (x \cdot z)$
⟨*proof*⟩
lemma *dot-lsub*: $((x::'a::ring\ \hat{}\ 'n) - y) \cdot z = (x \cdot z) - (y \cdot z)$
⟨*proof*⟩

lemma *dot-rsub*: $(x::'a::ring\ \hat{}\ 'n) \cdot (y - z) = (x \cdot y) - (x \cdot z)$
⟨*proof*⟩
lemma *dot-lmult*: $(c *s x) \cdot y = (c::'a::ring) * (x \cdot y)$ ⟨*proof*⟩
lemma *dot-rmult*: $x \cdot (c *s y) = (c::'a::comm\text{-}ring) * (x \cdot y)$ ⟨*proof*⟩
lemma *dot-lneg*: $(-x) \cdot (y::'a::ring\ \hat{}\ 'n) = -(x \cdot y)$ ⟨*proof*⟩
lemma *dot-rneg*: $(x::'a::ring\ \hat{}\ 'n) \cdot (-y) = -(x \cdot y)$ ⟨*proof*⟩
lemma *dot-lzero*[*simp*]: $0 \cdot x = (0::'a::\{comm\text{-}monoid\text{-}add,\ mult\text{-}zero\})$ ⟨*proof*⟩
lemma *dot-rzero*[*simp*]: $x \cdot 0 = (0::'a::\{comm\text{-}monoid\text{-}add,\ mult\text{-}zero\})$ ⟨*proof*⟩
lemma *dot-pos-le*[*simp*]: $(0::'a::ordered\text{-}ring\text{-}strict) <= x \cdot x$
⟨*proof*⟩

lemma *setsum-squares-eq-0-iff*: **assumes** *fS*: *finite F* **and** *fp*: $\forall x \in F.\ f\ x \geq (0::'a::pordered\text{-}ab\text{-}group\text{-}add)$ **shows** $setsum\ f\ F = 0 \longleftrightarrow (ALL\ x{:}F.\ f\ x = 0)$
⟨*proof*⟩

lemma *dot-eq-0*: $x \cdot x = 0 \longleftrightarrow (x::'a::\{ordered\text{-}ring\text{-}strict,ring\text{-}no\text{-}zero\text{-}divisors\}\ \hat{}\ 'n::finite) = 0$
⟨*proof*⟩

lemma *dot-pos-lt*[*simp*]: $(0 < x \cdot x) \longleftrightarrow (x::'a::\{ordered\text{-}ring\text{-}strict,ring\text{-}no\text{-}zero\text{-}divisors\}\ \hat{}\ 'n::finite) \neq 0$ ⟨*proof*⟩

30.8 The collapse of the general concepts to dimension one.

lemma *vector-one*: $(x::'a\ \hat{}1) = (\chi\ i.\ (x\$1))$
⟨*proof*⟩

lemma *forall-one*: $(\forall (x::'a\ \hat{}1).\ P\ x) \longleftrightarrow (\forall x.\ P(\chi\ i.\ x))$
⟨*proof*⟩

lemma *norm-vector-1*: $norm\ (x :: -\hat{}1) = norm\ (x\$1)$
⟨*proof*⟩

lemma *norm-real*: $norm(x::real\ \hat{}\ 1) = abs(x\$1)$
⟨*proof*⟩

Metric

FIXME: generalize to arbitrary *real-normed-vector* types

definition *dist*:: $real\ \hat{}\ 'n::finite \Rightarrow real\ \hat{}\ 'n \Rightarrow real$ **where**
$dist\ x\ y = norm\ (x - y)$

lemma *dist-real*: $dist(x::real\ \hat{}\ 1)\ y = abs((x\$1) - (y\$1))$
⟨*proof*⟩

30.9 A connectedness or intermediate value lemma with several applications.

lemma *connected-real-lemma*:
fixes $f :: real \Rightarrow real\ \hat{}\ 'n::finite$

assumes *ab*: $a \leq b$ **and** *fa*: $f\ a \in e1$ **and** *fb*: $f\ b \in e2$
and *dst*: $\bigwedge e\ x.\ a <= x \Longrightarrow x <= b \Longrightarrow 0 < e ==> \exists\, d > 0.\ \forall\, y.\ abs(y - x) < d \longrightarrow dist(f\ y)\ (f\ x) < e$
and *e1*: $\forall\, y \in e1.\ \exists\, e > 0.\ \forall\, y'.\ dist\ y'\ y < e \longrightarrow y' \in e1$
and *e2*: $\forall\, y \in e2.\ \exists\, e > 0.\ \forall\, y'.\ dist\ y'\ y < e \longrightarrow y' \in e2$
and *e12*: $\sim(\exists\, x \geq a.\ x <= b \wedge f\ x \in e1 \wedge f\ x \in e2)$
shows $\exists\, x \geq a.\ x <= b \wedge f\ x \notin e1 \wedge f\ x \notin e2$ (**is** $\exists\ x.\ ?P\ x$)
⟨*proof*⟩

One immediately useful corollary is the existence of square roots! — Should help to get rid of all the development of square-root for reals as a special case *real ^ 1*

lemma *square-bound-lemma*: $(x::real) < (1 + x) * (1 + x)$
⟨*proof*⟩

lemma *square-continuous*: $0 < (e::real) ==> \exists\, d.\ 0 < d \wedge (\forall\, y.\ abs(y - x) < d \longrightarrow abs(y * y - x * x) < e)$
⟨*proof*⟩

lemma *real-le-lsqrt*: $0 <= x \Longrightarrow 0 <= y \Longrightarrow x <= y\hat{}2 ==> sqrt\ x <= y$
⟨*proof*⟩

lemma *real-le-rsqrt*: $x\hat{}2 \leq y \Longrightarrow x \leq sqrt\ y$
⟨*proof*⟩

lemma *real-less-rsqrt*: $x\hat{}2 < y \Longrightarrow x < sqrt\ y$
⟨*proof*⟩

lemma *sqrt-even-pow2*: **assumes** *n*: *even n*
shows $sqrt(2\ \hat{}\ n) = 2\ \hat{}\ (n\ div\ 2)$
⟨*proof*⟩

lemma *real-div-sqrt*: $0 <= x ==> x\ /\ sqrt(x) = sqrt(x)$
⟨*proof*⟩

Hence derive more interesting properties of the norm.

This type-specific version is only here to make *normarith.ML* happy.

lemma *norm-0*: $norm\ (0::real\ \hat{}\ -) = 0$
⟨*proof*⟩

lemma *norm-mul*[*simp*]: $norm(a\ *s\ x) = abs(a) * norm\ x$
⟨*proof*⟩
lemma *norm-eq-0-dot*: $(norm\ x = 0) \longleftrightarrow (x \cdot x = (0::real))$
⟨*proof*⟩
lemma *real-vector-norm-def*: $norm\ x = sqrt\ (x \cdot x)$
⟨*proof*⟩
lemma *norm-pow-2*: $norm\ x\ \hat{}\ 2 = x \cdot x$
⟨*proof*⟩
lemma *norm-eq-0-imp*: $norm\ x = 0 ==> x = (0::real\ \hat{}'n::finite)$ ⟨*proof*⟩

lemma *vector-mul-eq-0[simp]*: $(a *s x = 0) \longleftrightarrow a = (0::'a::idom) \vee x = 0$
⟨*proof*⟩
lemma *vector-mul-lcancel[simp]*: $a *s x = a *s y \longleftrightarrow a = (0::real) \vee x = y$
⟨*proof*⟩
lemma *vector-mul-rcancel[simp]*: $a *s x = b *s x \longleftrightarrow (a::real) = b \vee x = 0$
⟨*proof*⟩
lemma *vector-mul-lcancel-imp*: $a \neq (0::real) ==> a *s x = a *s y ==> (x = y)$
⟨*proof*⟩
lemma *vector-mul-rcancel-imp*: $x \neq 0 \Longrightarrow (a::real) *s x = b *s x ==> a = b$
⟨*proof*⟩
lemma *norm-cauchy-schwarz*:
fixes $x\ y :: real\ \hat{}\ 'n::finite$
shows $x \cdot y <= norm\ x * norm\ y$
⟨*proof*⟩

lemma *norm-cauchy-schwarz-abs*:
fixes $x\ y :: real\ \hat{}\ 'n::finite$
shows $|x \cdot y| \leq norm\ x * norm\ y$
⟨*proof*⟩

lemma *norm-triangle-sub*: $norm\ (x::real\ \hat{}\ 'n::finite) <= norm(y) + norm(x - y)$
⟨*proof*⟩
lemma *norm-triangle-le*: $norm(x::real\ \hat{}\ 'n::finite) + norm\ y <= e ==> norm(x + y) <= e$
⟨*proof*⟩
lemma *norm-triangle-lt*: $norm(x::real\ \hat{}\ 'n::finite) + norm(y) < e ==> norm(x + y) < e$
⟨*proof*⟩

lemma *setsum-delta*:
assumes *fS*: *finite S*
shows *setsum* $(\lambda k.$ *if* $k=a$ *then b k else 0*$)\ S =$ (*if* $a \in S$ *then b a else 0*)
⟨*proof*⟩

lemma *component-le-norm*: $|x\$i| <= norm\ (x::real\ \hat{}\ 'n::finite)$
⟨*proof*⟩

lemma *norm-bound-component-le*: $norm(x::real\ \hat{}\ 'n::finite) <= e$
$==> |x\$i| <= e$
⟨*proof*⟩

lemma *norm-bound-component-lt*: $norm(x::real\ \hat{}\ 'n::finite) < e$
$==> |x\$i| < e$
⟨*proof*⟩

lemma *norm-le-l1*: $norm\ (x::\ real\ \hat{}\ 'n::finite) <= setsum(\lambda i.\ |x\$i|)\ UNIV$
⟨*proof*⟩

lemma *real-abs-norm*: $|norm\ x| = norm\ (x :: real\ \hat{}\ -)$
⟨*proof*⟩
lemma *real-abs-sub-norm*: $|norm(x::real\ \hat{}\ 'n::finite) - norm\ y| <= norm(x - y)$
⟨*proof*⟩
lemma *norm-le*: $norm(x::real\ \hat{}\ -) <= norm(y) \longleftrightarrow x \cdot x <= y \cdot y$
⟨*proof*⟩
lemma *norm-lt*: $norm(x::real\ \hat{}\ -) < norm(y) \longleftrightarrow x \cdot x < y \cdot y$
⟨*proof*⟩
lemma *norm-eq*: $norm\ (x::real\ \hat{}\ -) = norm\ y \longleftrightarrow x \cdot x = y \cdot y$
⟨*proof*⟩
lemma *norm-eq-1*: $norm(x::real\ \hat{}\ -) = 1 \longleftrightarrow x \cdot x = 1$
⟨*proof*⟩

Squaring equations and inequalities involving norms.

lemma *dot-square-norm*: $x \cdot x = norm(x)\ \hat{}2$
⟨*proof*⟩

lemma *norm-eq-square*: $norm(x) = a \longleftrightarrow 0 <= a \wedge x \cdot x = a\hat{}2$
⟨*proof*⟩

lemma *real-abs-le-square-iff*: $|x| \leq |y| \longleftrightarrow (x::real)\ \hat{}2 \leq y\hat{}2$
⟨*proof*⟩

lemma *norm-le-square*: $norm(x) <= a \longleftrightarrow 0 <= a \wedge x \cdot x <= a\hat{}2$
⟨*proof*⟩

lemma *norm-ge-square*: $norm(x) >= a \longleftrightarrow a <= 0 \vee x \cdot x >= a\ \hat{}\ 2$
⟨*proof*⟩

lemma *norm-lt-square*: $norm(x) < a \longleftrightarrow 0 < a \wedge x \cdot x < a\hat{}2$
⟨*proof*⟩
lemma *norm-gt-square*: $norm(x) > a \longleftrightarrow a < 0 \vee x \cdot x > a\hat{}2$
⟨*proof*⟩

Dot product in terms of the norm rather than conversely.

lemma *dot-norm*: $x \cdot y = (norm(x + y)\ \hat{}2 - norm\ x\ \hat{}\ 2 - norm\ y\ \hat{}\ 2)\ /\ 2$
⟨*proof*⟩

lemma *dot-norm-neg*: $x \cdot y = ((norm\ x\ \hat{}\ 2 + norm\ y\ \hat{}\ 2) - norm(x - y)\ \hat{}\ 2)\ /\ 2$
⟨*proof*⟩

Equality of vectors in terms of $op \cdot$ products.

lemma *vector-eq*: $(x::\ real\ \hat{}\ 'n::finite) = y \longleftrightarrow x \cdot x = x \cdot y \wedge y \cdot y = x \cdot x$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

30.10 General linear decision procedure for normed spaces.

lemma *norm-cmul-rule-thm*: $b >= norm(x) ==> |c| * b >= norm(c *s x)$

⟨*proof*⟩

lemma *norm-add-rule-thm*: *b1* >= *norm*(*x1* :: *real* ^'*n*::*finite*) ⟹ *b2* >= *norm*(*x2*)
==> *b1* + *b2* >= *norm*(*x1* + *x2*)
⟨*proof*⟩

lemma *ge-iff-diff-ge-0*: (*a*::'*a*::*ordered-ring*) ≥ *b* == *a* − *b* ≥ *0*
⟨*proof*⟩

lemma *pth-1*: (*x*::*real*^'*n*) == *1* ∗*s* *x* ⟨*proof*⟩
lemma *pth-2*: *x* − (*y*::*real*^'*n*) == *x* + −*y* ⟨*proof*⟩
lemma *pth-3*: (−*x*::*real*^'*n*) == −*1* ∗*s* *x* ⟨*proof*⟩
lemma *pth-4*: *0* ∗*s* (*x*::*real*^'*n*) == *0* *c* ∗*s* *0* = (*0*::*real* ^ '*n*) ⟨*proof*⟩
lemma *pth-5*: *c* ∗*s* (*d* ∗*s* *x*) == (*c* ∗ *d*) ∗*s* (*x*::*real* ^ '*n*) ⟨*proof*⟩
lemma *pth-6*: (*c*::*real*) ∗*s* (*x* + *y*) == *c* ∗*s* *x* + *c* ∗*s* *y* ⟨*proof*⟩
lemma *pth-7*: *0* + *x* == (*x*::*real*^'*n*) *x* + *0* == *x* ⟨*proof*⟩
lemma *pth-8*: (*c*::*real*) ∗*s* *x* + *d* ∗*s* *x* == (*c* + *d*) ∗*s* *x* ⟨*proof*⟩
lemma *pth-9*: ((*c*::*real*) ∗*s* *x* + *z*) + *d* ∗*s* *x* == (*c* + *d*) ∗*s* *x* + *z*
c ∗*s* *x* + (*d* ∗*s* *x* + *z*) == (*c* + *d*) ∗*s* *x* + *z*
(*c* ∗*s* *x* + *w*) + (*d* ∗*s* *x* + *z*) == (*c* + *d*) ∗*s* *x* + (*w* + *z*) ⟨*proof*⟩
lemma *pth-a*: (*0*::*real*) ∗*s* *x* + *y* == *y* ⟨*proof*⟩
lemma *pth-b*: (*c*::*real*) ∗*s* *x* + *d* ∗*s* *y* == *c* ∗*s* *x* + *d* ∗*s* *y*
(*c* ∗*s* *x* + *z*) + *d* ∗*s* *y* == *c* ∗*s* *x* + (*z* + *d* ∗*s* *y*)
c ∗*s* *x* + (*d* ∗*s* *y* + *z*) == *c* ∗*s* *x* + (*d* ∗*s* *y* + *z*)
(*c* ∗*s* *x* + *w*) + (*d* ∗*s* *y* + *z*) == *c* ∗*s* *x* + (*w* + (*d* ∗*s* *y* + *z*))
⟨*proof*⟩
lemma *pth-c*: (*c*::*real*) ∗*s* *x* + *d* ∗*s* *y* == *d* ∗*s* *y* + *c* ∗*s* *x*
(*c* ∗*s* *x* + *z*) + *d* ∗*s* *y* == *d* ∗*s* *y* + (*c* ∗*s* *x* + *z*)
c ∗*s* *x* + (*d* ∗*s* *y* + *z*) == *d* ∗*s* *y* + (*c* ∗*s* *x* + *z*)
(*c* ∗*s* *x* + *w*) + (*d* ∗*s* *y* + *z*) == *d* ∗*s* *y* + ((*c* ∗*s* *x* + *w*) + *z*) ⟨*proof*⟩
lemma *pth-d*: *x* + (*0*::*real* ^'*n*) == *x* ⟨*proof*⟩

lemma *norm-imp-pos-and-ge*: *norm* (*x*::*real* ^ -) == *n* ⟹ *norm* *x* ≥ *0* ∧ *n* ≥ *norm* *x*
⟨*proof*⟩

lemma *real-eq-0-iff-le-ge-0*: (*x*::*real*) = *0* == *x* ≥ *0* ∧ −*x* ≥ *0* ⟨*proof*⟩

lemma *norm-pths*:
(*x*::*real* ^'*n*::*finite*) = *y* ⟷ *norm* (*x* − *y*) ≤ *0*
x ≠ *y* ⟷ ¬ (*norm* (*x* − *y*) ≤ *0*)
⟨*proof*⟩

⟨*ML*⟩

Hence more metric properties.

lemma *dist-refl*[*simp*]: *dist* *x* *x* = *0* ⟨*proof*⟩

lemma *dist-sym*: *dist x y = dist y x*⟨*proof*⟩

lemma *dist-pos-le*[*simp*]: *0 <= dist x y* ⟨*proof*⟩

lemma *dist-triangle*: *dist x z <= dist x y + dist y z* ⟨*proof*⟩

lemma *dist-triangle-alt*: *dist y z <= dist x y + dist x z* ⟨*proof*⟩

lemma *dist-eq-0*[*simp*]: *dist x y = 0* ⟷ *x = y* ⟨*proof*⟩

lemma *dist-pos-lt*: *x ≠ y ==> 0 < dist x y* ⟨*proof*⟩
lemma *dist-nz*: *x ≠ y* ⟷ *0 < dist x y* ⟨*proof*⟩

lemma *dist-triangle-le*: *dist x z + dist y z <= e* ⟹ *dist x y <= e* ⟨*proof*⟩

lemma *dist-triangle-lt*: *dist x z + dist y z < e ==> dist x y < e* ⟨*proof*⟩

lemma *dist-triangle-half-l*: *dist x1 y < e / 2* ⟹ *dist x2 y < e / 2 ==> dist x1 x2 < e* ⟨*proof*⟩

lemma *dist-triangle-half-r*: *dist y x1 < e / 2* ⟹ *dist y x2 < e / 2 ==> dist x1 x2 < e* ⟨*proof*⟩

lemma *dist-triangle-add*: *dist (x + y) (x′ + y′) <= dist x x′ + dist y y′*
⟨*proof*⟩

lemma *dist-mul*[*simp*]: *dist (c *s x) (c *s y) = |c| * dist x y*
⟨*proof*⟩

lemma *dist-triangle-add-half*: *dist x x′ < e / 2* ⟹ *dist y y′ < e / 2 ==> dist(x + y) (x′ + y′) < e* ⟨*proof*⟩

lemma *dist-le-0*[*simp*]: *dist x y <= 0* ⟷ *x = y* ⟨*proof*⟩

lemma *setsum-component* [*simp*]:
fixes *f*:: *'a* ⇒ (*'b*::*comm-monoid-add*) ^*'n*
shows (*setsum f S*)$*i* = *setsum* (λ*x*. (*f x*)$*i*) *S*
⟨*proof*⟩

lemma *setsum-eq*: *setsum f S* = (χ *i*. *setsum* (λ*x*. (*f x*)$*i*) *S*)
⟨*proof*⟩

lemma *setsum-clauses*:
shows *setsum f* {} = *0*
and *finite S* ⟹ *setsum f* (*insert x S*) =
(*if x* ∈ *S then setsum f S else f x + setsum f S*)
⟨*proof*⟩

lemma *setsum-cmul*:

fixes f:: $'c \Rightarrow ('a$::*semiring-1*$)\hat{}'n$
shows *setsum* $(\lambda x.\ c *s f\ x)\ S = c *s$ *setsum* $f\ S$
⟨*proof*⟩

lemma *setsum-norm*:
fixes f :: $'a \Rightarrow 'b$::*real-normed-vector*
assumes *fS*: *finite* S
shows *norm* (*setsum* $f\ S$) <= *setsum* $(\lambda x.$ *norm*$(f\ x))\ S$
⟨*proof*⟩

lemma *real-setsum-norm*:
fixes f :: $'a \Rightarrow$ *real* $\hat{}'n$::*finite*
assumes *fS*: *finite* S
shows *norm* (*setsum* $f\ S$) <= *setsum* $(\lambda x.$ *norm*$(f\ x))\ S$
⟨*proof*⟩

lemma *setsum-norm-le*:
fixes f :: $'a \Rightarrow 'b$::*real-normed-vector*
assumes *fS*: *finite* S
and *fg*: $\forall x \in S.$ *norm* $(f\ x) \leq g\ x$
shows *norm* $(setsum\ f\ S) \leq setsum\ g\ S$
⟨*proof*⟩

lemma *real-setsum-norm-le*:
fixes f :: $'a \Rightarrow$ *real* $\hat{}\ 'n$::*finite*
assumes *fS*: *finite* S
and *fg*: $\forall x \in S.$ *norm* $(f\ x) \leq g\ x$
shows *norm* $(setsum\ f\ S) \leq setsum\ g\ S$
⟨*proof*⟩

lemma *setsum-norm-bound*:
fixes f :: $'a \Rightarrow 'b$::*real-normed-vector*
assumes *fS*: *finite* S
and K: $\forall x \in S.$ *norm* $(f\ x) \leq K$
shows *norm* $(setsum\ f\ S) \leq$ *of-nat* $(card\ S) * K$
⟨*proof*⟩

lemma *real-setsum-norm-bound*:
fixes f :: $'a \Rightarrow$ *real* $\hat{}\ 'n$::*finite*
assumes *fS*: *finite* S
and K: $\forall x \in S.$ *norm* $(f\ x) \leq K$
shows *norm* $(setsum\ f\ S) \leq$ *of-nat* $(card\ S) * K$
⟨*proof*⟩

lemma *setsum-vmul*:
fixes f :: $'a \Rightarrow 'b$::{*real-normed-vector*,*semiring*, *mult-zero*}
assumes *fS*: *finite* S
shows *setsum* $f\ S *s\ v =$ *setsum* $(\lambda x.\ f\ x *s\ v)\ S$
⟨*proof*⟩

lemma *setsum-add-split*: **assumes** *mn*: $(m::nat) \leq n + 1$
shows $setsum\ f\ \{m..n + p\} = setsum\ f\ \{m..n\} + setsum\ f\ \{n + 1..n + p\}$
⟨*proof*⟩

lemma *setsum-natinterval-left*:
assumes *mn*: $(m::nat) <= n$
shows $setsum\ f\ \{m..n\} = f\ m + setsum\ f\ \{m + 1..n\}$
⟨*proof*⟩

lemma *setsum-natinterval-difff*:
fixes $f::\ nat \Rightarrow ('a::ab\text{-}group\text{-}add)$
shows $setsum\ (\lambda k.\ f\ k - f(k + 1))\ \{(m::nat)\ ..\ n\} =$
$(if\ m <= n\ then\ f\ m - f(n + 1)\ else\ 0)$
⟨*proof*⟩

lemmas *setsum-restrict-set'* = *setsum-restrict-set*[*unfolded Int-def*]

lemma *setsum-setsum-restrict*:
$finite\ S \Longrightarrow finite\ T \Longrightarrow setsum\ (\lambda x.\ setsum\ (\lambda y.\ f\ x\ y)\ \{y.\ y \in T \wedge R\ x\ y\})\ S$
$= setsum\ (\lambda y.\ setsum\ (\lambda x.\ f\ x\ y)\ \{x.\ x \in S \wedge R\ x\ y\})\ T$
⟨*proof*⟩

lemma *setsum-image-gen*: **assumes** *fS*: *finite S*
shows $setsum\ g\ S = setsum\ (\lambda y.\ setsum\ g\ \{x.\ x \in S \wedge f\ x = y\})\ (f\ `\ S)$
⟨*proof*⟩

lemma *setsum-group*:
assumes *fS*: *finite S* **and** *fT*: *finite T* **and** *fST*: $f\ `\ S \subseteq T$
shows $setsum\ (\lambda y.\ setsum\ g\ \{x.\ x \in S \wedge f\ x = y\})\ T = setsum\ g\ S$

⟨*proof*⟩

lemma *vsum-norm-allsubsets-bound*:
fixes $f::\ 'a \Rightarrow real\ \hat{}\ 'n::finite$
assumes *fP*: *finite P* **and** *fPs*: $\bigwedge Q.\ Q \subseteq P \Longrightarrow norm\ (setsum\ f\ Q) \leq e$
shows $setsum\ (\lambda x.\ norm\ (f\ x))\ P \leq 2 * real\ CARD('n) *\ e$
⟨*proof*⟩

lemma *dot-lsum*: $finite\ S \Longrightarrow setsum\ f\ S \cdot (y::'a::\{comm\text{-}ring\}\ \hat{}\ 'n) = setsum\ (\lambda x.$
$f\ x \cdot y)\ S$
⟨*proof*⟩

lemma *dot-rsum*: $finite\ S \Longrightarrow (y::'a::\{comm\text{-}ring\}\ \hat{}\ 'n) \cdot setsum\ f\ S = setsum\ (\lambda x.$
$y \cdot f\ x)\ S$
⟨*proof*⟩

30.11 Basis vectors in coordinate directions.

definition *basis k = (χ i. if i = k then 1 else 0)*

lemma *basis-component* [*simp*]: *basis k $ i = (if k=i then 1 else 0)*
⟨*proof*⟩

lemma *delta-mult-idempotent*:
*(if k=a then 1 else (0::'a::semiring-1)) * (if k=a then 1 else 0) = (if k=a then 1 else 0)* ⟨*proof*⟩

lemma *norm-basis*:
shows *norm (basis k :: real ^'n::finite) = 1*
⟨*proof*⟩

lemma *norm-basis-1*: *norm(basis 1 :: real ^'n::{finite,one}) = 1*
⟨*proof*⟩

lemma *vector-choose-size*: *0 <= c ==> ∃(x::real^'n::finite). norm x = c*
⟨*proof*⟩

lemma *vector-choose-dist*: **assumes** *e*: *0 <= e*
shows *∃(y::real^'n::finite). dist x y = e*
⟨*proof*⟩

lemma *basis-inj*: *inj (basis :: 'n ⇒ real ^'n::finite)*
⟨*proof*⟩

lemma *cond-value-iff*: *f (if b then x else y) = (if b then f x else f y)*
⟨*proof*⟩

lemma *basis-expansion*:
*setsum (λi. (x$i) *s basis i) UNIV = (x::('a::ring-1) ^'n::finite)* (**is** *?lhs = ?rhs* **is** *setsum ?f ?S = -*)
⟨*proof*⟩

lemma *basis-expansion-unique*:
*setsum (λi. f i *s basis i) UNIV = (x::('a::comm-ring-1) ^'n::finite) ⟷ (∀ i. f i = x$i)*
⟨*proof*⟩

lemma *cond-application-beta*: *(if b then f else g) x = (if b then f x else g x)*
⟨*proof*⟩

lemma *dot-basis*:
shows *basis i · x = x$i x · (basis i :: 'a^'n::finite) = (x$i :: 'a::semiring-1)*
⟨*proof*⟩

lemma *basis-eq-0*: *basis i = (0::'a::semiring-1^'n) ⟷ False*
⟨*proof*⟩

lemma *basis-nonzero*:
shows *basis k* ≠ (*0*:: *'a*::*semiring-1* ^*'n*)
⟨*proof*⟩

lemma *vector-eq-ldot*: (∀ *x*. *x* · *y* = *x* · *z*) ⟷ *y* = (*z*::*'a*::*semiring-1*^*'n*::*finite*)
⟨*proof*⟩

lemma *vector-eq-rdot*: (∀ *z*. *x* · *z* = *y* · *z*) ⟷ *x* = (*y*::*'a*::*semiring-1*^*'n*::*finite*)
⟨*proof*⟩

30.12 Orthogonality.

definition *orthogonal x y* ⟷ (*x* · *y* = *0*)

lemma *orthogonal-basis*:
shows *orthogonal* (*basis i* :: *'a*^*'n*::*finite*) *x* ⟷ *x*$*i* = (*0*::*'a*::*ring-1*)
⟨*proof*⟩

lemma *orthogonal-basis-basis*:
shows *orthogonal* (*basis i* :: *'a*::*ring-1*^*'n*::*finite*) (*basis j*) ⟷ *i* ≠ *j*
⟨*proof*⟩

lemma *orthogonal-clauses*:
orthogonal a (*0*::*'a*::*comm-ring* ^*'n*)
orthogonal a x ==> *orthogonal a* (*c* *s *x*)
orthogonal a x ==> *orthogonal a* (−*x*)
orthogonal a x ⟹ *orthogonal a y* ==> *orthogonal a* (*x* + *y*)
orthogonal a x ⟹ *orthogonal a y* ==> *orthogonal a* (*x* − *y*)
orthogonal 0 a
orthogonal x a ==> *orthogonal* (*c* *s *x*) *a*
orthogonal x a ==> *orthogonal* (−*x*) *a*
orthogonal x a ⟹ *orthogonal y a* ==> *orthogonal* (*x* + *y*) *a*
orthogonal x a ⟹ *orthogonal y a* ==> *orthogonal* (*x* − *y*) *a*
⟨*proof*⟩

lemma *orthogonal-commute*: *orthogonal* (*x*::*'a*::{*ab-semigroup-mult*,*comm-monoid-add*}
^*'n*)*y* ⟷ *orthogonal y x*
⟨*proof*⟩

30.13 Explicit vector construction from lists.

primrec *from-nat* :: *nat* ⇒ *'a*::{*monoid-add*,*one*}
where *from-nat 0* = *0* | *from-nat* (*Suc n*) = *1* + *from-nat n*

lemma *from-nat* [*simp*]: *from-nat* = *of-nat*
⟨*proof*⟩

primrec

list-fun :: *nat* ⇒ - *list* ⇒ - ⇒ -
where
list-fun n [] = (λ*x*. *0*)
| *list-fun n* (*x* # *xs*) = *fun-upd* (*list-fun* (*Suc n*) *xs*) (*from-nat n*) *x*

definition *vector l* = (χ *i*. *list-fun 1 l i*)

lemma *vector-1*: (*vector*[*x*]) $*1* = *x*
⟨*proof*⟩

lemma *vector-2*:
(*vector*[*x*,*y*]) $*1* = *x*
(*vector*[*x*,*y*] :: *'a^2*)$*2* = (*y*::*'a*::*zero*)
⟨*proof*⟩

lemma *vector-3*:
(*vector* [*x*,*y*,*z*] ::(*'a*::*zero*) *^3*)$*1* = *x*
(*vector* [*x*,*y*,*z*] ::(*'a*::*zero*) *^3*)$*2* = *y*
(*vector* [*x*,*y*,*z*] ::(*'a*::*zero*) *^3*)$*3* = *z*
⟨*proof*⟩

lemma *forall-vector-1*: (∀ *v*::*'a*::*zero^1*. *P v*) ⟷ (∀ *x*. *P*(*vector*[*x*]))
⟨*proof*⟩

lemma *forall-vector-2*: (∀ *v*::*'a*::*zero^2*. *P v*) ⟷ (∀ *x y*. *P*(*vector*[*x*, *y*]))
⟨*proof*⟩

lemma *forall-vector-3*: (∀ *v*::*'a*::*zero^3*. *P v*) ⟷ (∀ *x y z*. *P*(*vector*[*x*, *y*, *z*]))
⟨*proof*⟩

30.14 Linear functions.

definition *linear f* ⟷ (∀ *x y*. *f*(*x* + *y*) = *f x* + *f y*) ∧ (∀ *c x*. *f*(*c* ∗*s* *x*) = *c* ∗*s* *f x*)

lemma *linear-compose-cmul*: *linear f* ==> *linear* (λ*x*. (*c*::*'a*::*comm-semiring*) ∗*s* *f x*)
⟨*proof*⟩

lemma *linear-compose-neg*: *linear* (*f* :: *'a ^'n* ⇒ *'a*::*comm-ring ^'m*) ==> *linear* (λ*x*. −(*f*(*x*))) ⟨*proof*⟩

lemma *linear-compose-add*: *linear* (*f* :: *'a ^'n* ⇒ *'a*::*semiring-1 ^'m*) ⟹ *linear g* ==> *linear* (λ*x*. *f*(*x*) + *g*(*x*))
⟨*proof*⟩

lemma *linear-compose-sub*: *linear* (*f* :: *'a ^'n* ⇒ *'a*::*ring-1 ^'m*) ⟹ *linear g* ==> *linear* (λ*x*. *f x* − *g x*)

⟨*proof*⟩

lemma *linear-compose*: *linear f* ⟹ *linear g* ==> *linear* (*g o f*)
⟨*proof*⟩

lemma *linear-id*: *linear id* ⟨*proof*⟩

lemma *linear-zero*: *linear* (λ*x*. *0*::*'a*::*semiring-1* ^ *'n*) ⟨*proof*⟩

lemma *linear-compose-setsum*:
assumes *fS*: *finite S* **and** *lS*: ∀ *a* ∈ *S*. *linear* (*f a* :: *'a*::*semiring-1* ^ *'n* ⇒ *'a* ^ *'m*)
shows *linear*(λ*x*. *setsum* (λ*a*. *f a x* :: *'a*::*semiring-1* ^*'m*) *S*)
⟨*proof*⟩

lemma *linear-vmul-component*:
fixes *f*:: *'a*::*semiring-1*^*'m* ⇒ *'a*^*'n*
assumes *lf*: *linear f*
shows *linear* (λ*x*. *f x* $ *k* ∗*s v*)
⟨*proof*⟩

lemma *linear-0*: *linear f* ==> *f 0* = (*0*::*'a*::*semiring-1* ^*'n*)
⟨*proof*⟩

lemma *linear-cmul*: *linear f* ==> *f*(*c*∗*s x*) = *c* ∗*s f x* ⟨*proof*⟩

lemma *linear-neg*: *linear* (*f* :: *'a*::*ring-1* ^*'n* ⇒ -) ==> *f* (−*x*) = − *f x*
⟨*proof*⟩

lemma *linear-add*: *linear f* ==> *f*(*x* + *y*) = *f x* + *f y* ⟨*proof*⟩

lemma *linear-sub*: *linear* (*f*::*'a*::*ring-1* ^*'n* ⇒ -) ==> *f*(*x* − *y*) = *f x* − *f y*
⟨*proof*⟩

lemma *linear-setsum*:
fixes *f*:: *'a*::*semiring-1*^*'n* ⇒ -
assumes *lf*: *linear f* **and** *fS*: *finite S*
shows *f* (*setsum g S*) = *setsum* (*f o g*) *S*
⟨*proof*⟩

lemma *linear-setsum-mul*:
fixes *f*:: *'a* ^*'n* ⇒ *'a*::*semiring-1*^*'m*
assumes *lf*: *linear f* **and** *fS*: *finite S*
shows *f* (*setsum* (λ*i*. *c i* ∗*s v i*) *S*) = *setsum* (λ*i*. *c i* ∗*s f* (*v i*)) *S*
⟨*proof*⟩

lemma *linear-injective-0*:
assumes *lf*: *linear* (*f*:: *'a*::*ring-1* ^ *'n* ⇒ -)
shows *inj f* ⟷ (∀ *x*. *f x* = *0* ⟶ *x* = *0*)

⟨*proof*⟩

lemma *linear-bounded*:
fixes *f*:: *real ^'m::finite ⇒ real ^'n::finite*
assumes *lf*: *linear f*
shows ∃ *B*. ∀ *x*. *norm* (*f x*) ≤ *B* ∗ *norm x*
⟨*proof*⟩

lemma *linear-bounded-pos*:
fixes *f*:: *real ^'n::finite ⇒ real ^ 'm::finite*
assumes *lf*: *linear f*
shows ∃ *B* > *0*. ∀ *x*. *norm* (*f x*) ≤ *B* ∗ *norm x*
⟨*proof*⟩

30.15 Bilinear functions.

definition *bilinear f* ⟷ (∀ *x*. *linear*(λ*y*. *f x y*)) ∧ (∀ *y*. *linear*(λ*x*. *f x y*))

lemma *bilinear-ladd*: *bilinear h* ==> *h* (*x* + *y*) *z* = (*h x z*) + (*h y z*)
⟨*proof*⟩
lemma *bilinear-radd*: *bilinear h* ==> *h x* (*y* + *z*) = (*h x y*) + (*h x z*)
⟨*proof*⟩

lemma *bilinear-lmul*: *bilinear h* ==> *h* (*c* ∗*s x*) *y* = *c* ∗*s* (*h x y*)
⟨*proof*⟩

lemma *bilinear-rmul*: *bilinear h* ==> *h x* (*c* ∗*s y*) = *c* ∗*s* (*h x y*)
⟨*proof*⟩

lemma *bilinear-lneg*: *bilinear h* ==> *h* (− (*x*:: *'a::ring-1 ^ 'n*)) *y* = −(*h x y*)
⟨*proof*⟩

lemma *bilinear-rneg*: *bilinear h* ==> *h x* (− (*y*:: *'a::ring-1 ^ 'n*)) = − *h x y*
⟨*proof*⟩

lemma (**in** *ab-group-add*) *eq-add-iff*: *x* = *x* + *y* ⟷ *y* = *0*
⟨*proof*⟩

lemma *bilinear-lzero*:
fixes *h* :: *'a::ring^'n* ⇒ - **assumes** *bh*: *bilinear h* **shows** *h 0 x* = *0*
⟨*proof*⟩

lemma *bilinear-rzero*:
fixes *h* :: *'a::ring^'n* ⇒ - **assumes** *bh*: *bilinear h* **shows** *h x 0* = *0*
⟨*proof*⟩

lemma *bilinear-lsub*: *bilinear h* ==> *h* (*x* − (*y*:: *'a::ring-1 ^ 'n*)) *z* = *h x z* − *h y z*
⟨*proof*⟩

lemma *bilinear-rsub*: *bilinear h ==> h z (x − (y:: 'a::ring-1 ^ 'n)) = h z x − h z y*
⟨*proof*⟩

lemma *bilinear-setsum*:
fixes *h*:: *'a ^'n ⇒ 'a::semiring-1^'m ⇒ 'a ^ 'k*
assumes *bh*: *bilinear h* **and** *fS*: *finite S* **and** *fT*: *finite T*
shows *h (setsum f S) (setsum g T) = setsum (λ(i,j). h (f i) (g j)) (S × T)*
⟨*proof*⟩

lemma *bilinear-bounded*:
fixes *h*:: *real ^'m::finite ⇒ real^'n::finite ⇒ real ^ 'k::finite*
assumes *bh*: *bilinear h*
shows *∃ B. ∀ x y. norm (h x y) ≤ B * norm x * norm y*
⟨*proof*⟩

lemma *bilinear-bounded-pos*:
fixes *h*:: *real ^'m::finite ⇒ real^'n::finite ⇒ real ^ 'k::finite*
assumes *bh*: *bilinear h*
shows *∃ B > 0. ∀ x y. norm (h x y) ≤ B * norm x * norm y*
⟨*proof*⟩

30.16 Adjoints.

definition *adjoint f = (SOME f'. ∀ x y. f x · y = x · f' y)*

lemma *choice-iff*: *(∀ x. ∃ y. P x y) ⟷ (∃ f. ∀ x. P x (f x))* ⟨*proof*⟩

lemma *adjoint-works-lemma*:
fixes *f*:: *'a::ring-1 ^'n::finite ⇒ 'a ^ 'm::finite*
assumes *lf*: *linear f*
shows *∀ x y. f x · y = x · adjoint f y*
⟨*proof*⟩

lemma *adjoint-works*:
fixes *f*:: *'a::ring-1 ^'n::finite ⇒ 'a ^ 'm::finite*
assumes *lf*: *linear f*
shows *x · adjoint f y = f x · y*
⟨*proof*⟩

lemma *adjoint-linear*:
fixes *f* :: *'a::comm-ring-1 ^'n::finite ⇒ 'a ^ 'm::finite*
assumes *lf*: *linear f*
shows *linear (adjoint f)*
⟨*proof*⟩

lemma *adjoint-clauses*:

fixes f:: $'a$::*comm-ring-1* ^ $'n$::*finite* ⇒ $'a$ ^ $'m$::*finite*
assumes *lf*: *linear f*
shows $x \cdot$ *adjoint f y* = *f x* $\cdot$ *y*
and *adjoint f y* $\cdot$ *x* = *y* $\cdot$ *f x*
⟨*proof*⟩

lemma *adjoint-adjoint*:
fixes f:: $'a$::*comm-ring-1* ^ $'n$::*finite* ⇒ $'a$ ^ $'m$::*finite*
assumes *lf*: *linear f*
shows *adjoint* (*adjoint f*) = *f*
⟨*proof*⟩

lemma *adjoint-unique*:
fixes f:: $'a$::*comm-ring-1* ^ $'n$::*finite* ⇒ $'a$ ^ $'m$::*finite*
assumes *lf*: *linear f* **and** *u*: ∀ *x y*. *f'* *x* $\cdot$ *y* = *x* $\cdot$ *f y*
shows *f'* = *adjoint f*
⟨*proof*⟩

Matrix notation. NB: an MxN matrix is of type $'a$ ^ $'n$ ^ $'m$, not $'a$ ^ $'m$ ^ $'n$

consts *generic-mult* :: $'a$ ⇒ $'b$ ⇒ $'c$ (**infixr** ⋆ *75*)

defs (**overloaded**)
matrix-matrix-mult-def: (*m*:: ($'a$::*semiring-1*) ^$'n$^$'m$) ⋆ (*m'* :: $'a$ ^$'p$^$'n$) ≡ (χ *i j*. *setsum* (λ*k*. ((*m*$*i*)$*k*) ∗ ((*m'*$*k*)$*j*)) (*UNIV* :: $'n$ *set*)) ::$'a$ ^ $'p$ ^$'m$

abbreviation
matrix-matrix-mult' :: ($'a$::*semiring-1*) ^$'n$^$'m$ ⇒ $'a$ ^$'p$^$'n$ ⇒ $'a$ ^ $'p$ ^$'m$ (**infixl** ∗∗ *70*)
where *m* ∗∗ *m'* == *m*⋆ *m'*

defs (**overloaded**)
matrix-vector-mult-def: (*m*::($'a$::*semiring-1*) ^$'n$^$'m$) ⋆ (*x*::$'a$ ^$'n$) ≡ (χ *i*. *setsum* (λ*j*. ((*m*$*i*)$*j*) ∗ (*x*$*j*)) (*UNIV* ::$'n$ *set*)) :: $'a$^$'m$

abbreviation
matrix-vector-mult' :: ($'a$::*semiring-1*) ^$'n$^$'m$ ⇒ $'a$ ^$'n$ ⇒ $'a$ ^ $'m$ (**infixl** ∗*v* *70*)
where
m ∗*v v* == *m* ⋆ *v*

defs (**overloaded**)
vector-matrix-mult-def: (*x*::$'a$^$'m$) ⋆ (*m*::($'a$::*semiring-1*) ^$'n$^$'m$) ≡ (χ *j*. *setsum* (λ*i*. ((*m*$*i*)$*j*) ∗ (*x*$*i*)) (*UNIV* :: $'m$ *set*)) :: $'a$^$'n$

abbreviation
vactor-matrix-mult' :: $'a$ ^ $'m$ ⇒ ($'a$::*semiring-1*) ^$'n$^$'m$ ⇒ $'a$ ^$'n$ (**infixl** *v*∗ *70*)
where

$v\ v*\ m == v \star m$

definition $(mat::'a::zero => 'a\ \hat{}\ 'n\hat{}\ 'n)\ k = (\chi\ i\ j.\ if\ i = j\ then\ k\ else\ 0)$
definition $(transp::'a\hat{}\ 'n\hat{}\ 'm \Rightarrow 'a\hat{}\ 'm\hat{}\ 'n)\ A = (\chi\ i\ j.\ ((A\$j)\$i))$
definition $(row::'m => 'a\ \hat{}\ 'n\hat{}\ 'm \Rightarrow 'a\ \hat{}\ 'n)\ i\ A = (\chi\ j.\ ((A\$i)\$j))$
definition $(column::'n =>'a\hat{}\ 'n\hat{}\ 'm =>'a\hat{}\ 'm)\ j\ A = (\chi\ i.\ ((A\$i)\$j))$
definition $rows(A::'a\hat{}\ 'n\hat{}\ 'm) = \{\ row\ i\ A \mid i.\ i \in (UNIV :: 'm\ set)\}$
definition $columns(A::'a\hat{}\ 'n\hat{}\ 'm) = \{\ column\ i\ A \mid i.\ i \in (UNIV :: 'n\ set)\}$

lemma *mat-0*[*simp*]: $mat\ 0 = 0$ ⟨*proof*⟩
lemma *matrix-add-ldistrib*: $(A ** (B + C)) = (A \star B) + (A \star C)$
⟨*proof*⟩

lemma *setsum-delta'*:
assumes *fS*: *finite S* **shows**
$setsum\ (\lambda k.\ if\ a = k\ then\ b\ k\ else\ 0)\ S =$
$(if\ a \in S\ then\ b\ a\ else\ 0)$
⟨*proof*⟩

lemma *matrix-mul-lid*:
fixes $A :: 'a::semiring\text{-}1\ \hat{}\ 'm\ \hat{}\ 'n::finite$
shows $mat\ 1 ** A = A$
⟨*proof*⟩

lemma *matrix-mul-rid*:
fixes $A :: 'a::semiring\text{-}1\ \hat{}\ 'm::finite\ \hat{}\ 'n$
shows $A ** mat\ 1 = A$
⟨*proof*⟩

lemma *matrix-mul-assoc*: $A ** (B ** C) = (A ** B) ** C$
⟨*proof*⟩

lemma *matrix-vector-mul-assoc*: $A *v\ (B *v\ x) = (A ** B) *v\ x$
⟨*proof*⟩

lemma *matrix-vector-mul-lid*: $mat\ 1 *v\ x = (x::'a::semiring\text{-}1\ \hat{}\ 'n::finite)$
⟨*proof*⟩

lemma *matrix-transp-mul*: $transp(A ** B) = transp\ B ** transp\ (A::'a::comm\text{-}semiring\text{-}1\hat{}\ 'm\hat{}\ 'n)$
⟨*proof*⟩

lemma *matrix-eq*:
fixes $A\ B :: 'a::semiring\text{-}1\ \hat{}\ 'n::finite\ \hat{}\ 'm$
shows $A = B \longleftrightarrow (\forall x.\ A *v\ x = B *v\ x)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *matrix-vector-mul-component*:
shows $((A::'a::semiring\text{-}1\hat{}\ 'n'\hat{}\ 'm) *v\ x)\$k = (A\$k) \cdot x$

⟨*proof*⟩

lemma *dot-lmul-matrix*: $((x::'a::comm\text{-}semiring\text{-}1\ \hat{}\ 'n)\ v*\ A) \cdot y = x \cdot (A *v\ y)$
⟨*proof*⟩

lemma *transp-mat*: $transp\ (mat\ n) = mat\ n$
⟨*proof*⟩

lemma *transp-transp*: $transp(transp\ A) = A$
⟨*proof*⟩

lemma *row-transp*:
fixes $A::\ 'a::semiring\text{-}1\hat{}\ 'n\hat{}\ 'm$
shows $row\ i\ (transp\ A) = column\ i\ A$
⟨*proof*⟩

lemma *column-transp*:
fixes $A::\ 'a::semiring\text{-}1\hat{}\ 'n\hat{}\ 'm$
shows $column\ i\ (transp\ A) = row\ i\ A$
⟨*proof*⟩

lemma *rows-transp*: $rows(transp\ (A::'a::semiring\text{-}1\hat{}\ 'n\hat{}\ 'm)) = columns\ A$
⟨*proof*⟩

lemma *columns-transp*: $columns(transp\ (A::'a::semiring\text{-}1\hat{}\ 'n\hat{}\ 'm)) = rows\ A$ ⟨*proof*⟩

Two sometimes fruitful ways of looking at matrix-vector multiplication.

lemma *matrix-mult-dot*: $A *v\ x = (\chi\ i.\ A\$i \cdot x)$
⟨*proof*⟩

lemma *matrix-mult-vsum*: $(A::'a::comm\text{-}semiring\text{-}1\hat{}\ 'n\hat{}\ 'm) *v\ x = setsum\ (\lambda i.\ (x\$i) *s\ column\ i\ A)\ (UNIV::\ 'n\ set)$
⟨*proof*⟩

lemma *vector-componentwise*:
$(x::'a::ring\text{-}1\hat{}\ 'n::finite) = (\chi\ j.\ setsum\ (\lambda i.\ (x\$i) * (basis\ i::\ 'a\hat{}\ 'n)\$j)\ (UNIV::\ 'n\ set))$
⟨*proof*⟩

lemma *linear-componentwise*:
fixes $f::\ 'a::ring\text{-}1\ \hat{}\ 'm::finite \Rightarrow 'a\ \hat{}\ 'n$
assumes *lf*: *linear f*
shows $(f\ x)\$j = setsum\ (\lambda i.\ (x\$i) * (f\ (basis\ i)\$j))\ (UNIV::\ 'm\ set)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

Inverse matrices (not necessarily square)

definition $invertible(A::'a::semiring\text{-}1\hat{}\ 'n\hat{}\ 'm) \longleftrightarrow (\exists A'::'a\hat{}\ 'm\hat{}\ 'n.\ A **\ A' = mat\ 1 \wedge A' **\ A = mat\ 1)$

definition *matrix-inv*(*A*:: *'a*::*semiring-1*^*'n*^*'m*) =
(*SOME A'*::*'a*^*'m*^*'n*. *A* ** *A'* = *mat 1* ∧ *A'* ** *A* = *mat 1*)

Correspondence between matrices and linear operators.

definition *matrix*:: (*'a*::{*plus*,*times*, *one*, *zero*}^*'m* ⇒ *'a* ^ *'n*) ⇒ *'a*^*'m*^*'n*
where *matrix f* = (χ *i j*. (*f*(*basis j*))$*i*)

lemma *matrix-vector-mul-linear*: *linear*(λ*x*. *A* *v (*x*::*'a*::*comm-semiring-1* ^ *'n*))
⟨*proof*⟩

lemma *matrix-works*: **assumes** *lf*: *linear f* **shows** *matrix f* *v *x* = *f* (*x*::*'a*::*comm-ring-1*
^ *'n*::*finite*)
⟨*proof*⟩

lemma *matrix-vector-mul*: *linear f* ==> *f* = (λ*x*. *matrix f* *v (*x*::*'a*::*comm-ring-1*
^ *'n*::*finite*)) ⟨*proof*⟩

lemma *matrix-of-matrix-vector-mul*: *matrix*(λ*x*. *A* *v (*x* :: *'a*:: *comm-ring-1* ^
'n::*finite*)) = *A*
⟨*proof*⟩

lemma *matrix-compose*:
assumes *lf*: *linear* (*f*::*'a*::*comm-ring-1*^*'n*::*finite* ⇒ *'a*^*'m*::*finite*)
and *lg*: *linear* (*g*::*'a*::*comm-ring-1*^*'m*::*finite* ⇒ *'a*^*'k*)
shows *matrix* (*g o f*) = *matrix g* ** *matrix f*
⟨*proof*⟩

lemma *matrix-vector-column*:(*A*::*'a*::*comm-semiring-1*^*'n*^*'m*) *v *x* = *setsum* (λ*i*.
(*x*$*i*) *s ((*transp A*)$*i*)) (*UNIV*:: *'n set*)
⟨*proof*⟩

lemma *adjoint-matrix*: *adjoint*(λ*x*. (*A*::*'a*::*comm-ring-1*^*'n*::*finite*^*'m*::*finite*) *v
x) = (λ*x*. *transp A* *v *x*)
⟨*proof*⟩

lemma *matrix-adjoint*: **assumes** *lf*: *linear* (*f* :: *'a*::*comm-ring-1*^*'n*::*finite* ⇒ *'a* ^
'm::*finite*)
shows *matrix*(*adjoint f*) = *transp*(*matrix f*)
⟨*proof*⟩

30.17 Interlude: Some properties of real sets

lemma *seq-mono-lemma*: **assumes** ∀(*n*::*nat*) ≥ *m*. (*d n* :: *real*) < *e n* **and** ∀ *n* ≥
m. *e n* <= *e m*
shows ∀ *n* ≥ *m*. *d n* < *e m*
⟨*proof*⟩

lemma *real-convex-bound-lt*:

assumes *xa*: $(x::real) < a$ **and** *ya*: $y < a$ **and** *u*: $0 <= u$ **and** *v*: $0 <= v$
and *uv*: $u + v = 1$
shows $u * x + v * y < a$
⟨*proof*⟩

lemma *real-convex-bound-le*:
assumes *xa*: $(x::real) \leq a$ **and** *ya*: $y \leq a$ **and** *u*: $0 <= u$ **and** *v*: $0 <= v$
and *uv*: $u + v = 1$
shows $u * x + v * y \leq a$
⟨*proof*⟩

lemma *infinite-enumerate*: **assumes** *fS*: *infinite S*
shows $\exists\, r.\ subseq\ r \wedge (\forall\, n.\ r\ n \in S)$
⟨*proof*⟩

lemma *approachable-lt-le*: $(\exists\, (d::real)>0.\ \forall\, x.\ f\ x < d \longrightarrow P\ x) \longleftrightarrow (\exists\, d>0.\ \forall\, x.\ f\ x \leq d \longrightarrow P\ x)$
⟨*proof*⟩

lemma *triangle-lemma*:
assumes *x*: $0 <= (x::real)$ **and** *y*:$0 <= y$ **and** *z*: $0 <= z$ **and** *xy*: $x\hat{}2 <= y\hat{}2 + z\hat{}2$
shows $x <= y + z$
⟨*proof*⟩

lemma *lambda-skolem*: $(\forall\, i.\ \exists\, x.\ P\ i\ x) \longleftrightarrow$
$(\exists\, x::'a\ \hat{}\ 'n.\ \forall\, i.\ P\ i\ (x\$i))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

definition *rsup*:: *real set* ⇒ *real* **where**
rsup S = (*SOME a. isLub UNIV S a*)

lemma *rsup-alt*: *rsup S* = $(SOME\ a.\ (\forall\, x \in S.\ x \leq a) \wedge (\forall\, b.\ (\forall\, x \in S.\ x \leq b) \longrightarrow a \leq b))$ ⟨*proof*⟩

lemma *rsup*: **assumes** *Se*: $S \neq \{\}$ **and** *b*: $\exists\, b.\ S\ *<=\ b$
shows *isLub UNIV S* (*rsup S*)
⟨*proof*⟩

lemma *rsup-le*: **assumes** *Se*: $S \neq \{\}$ **and** *Sb*: $S\ *<=\ b$ **shows** *rsup S* $\leq b$
⟨*proof*⟩

lemma *rsup-finite-Max*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows *rsup S* = *Max S*

⟨*proof*⟩

lemma *rsup-finite-in*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows *rsup S* $\in$ *S*
⟨*proof*⟩

lemma *rsup-finite-Ub*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows *isUb S S* (*rsup S*)
⟨*proof*⟩

lemma *rsup-finite-ge-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a \leq$ *rsup S* $\longleftrightarrow$ $(\exists\ x \in S.\ a \leq x)$
⟨*proof*⟩

lemma *rsup-finite-le-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a \geq$ *rsup S* $\longleftrightarrow$ $(\forall\ x \in S.\ a \geq x)$
⟨*proof*⟩

lemma *rsup-finite-gt-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a <$ *rsup S* $\longleftrightarrow$ $(\exists\ x \in S.\ a < x)$
⟨*proof*⟩

lemma *rsup-finite-lt-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a >$ *rsup S* $\longleftrightarrow$ $(\forall\ x \in S.\ a > x)$
⟨*proof*⟩

lemma *rsup-unique*: **assumes** *b*: S *<=* b **and** *S*: $\forall\, b' < b.\ \exists x \in S.\ b' < x$
shows *rsup S* $= b$
⟨*proof*⟩

lemma *rsup-le-subset*: $S \neq \{\} \Longrightarrow S \subseteq T \Longrightarrow (\exists\, b.\ T$ *<=* $b) \Longrightarrow$ *rsup S* $\leq$ *rsup T*
⟨*proof*⟩

lemma *isUb-def'*: *isUb R S* $= (\lambda x.\ S$ *<=* $x \wedge x \in R)$
⟨*proof*⟩

lemma *UNIV-trivial*: *UNIV x* ⟨*proof*⟩
lemma *rsup-bounds*: **assumes** *Se*: $S \neq \{\}$ **and** *l*: a *<=** S **and** *u*: S *<=* b
shows $a \leq$ *rsup S* $\wedge$ *rsup S* $\leq b$
⟨*proof*⟩

lemma *rsup-abs-le*: $S \neq \{\} \Longrightarrow (\forall\, x{\in}S.\ |x| \leq a) \Longrightarrow |$*rsup S*$| \leq a$
⟨*proof*⟩

lemma *rsup-asclose*: **assumes** *S*:$S \neq \{\}$ **and** *b*: $\forall\, x{\in}S.\ |x - l| \leq e$ **shows** $|$*rsup S* $- l| \leq e$
⟨*proof*⟩

definition *rinf*:: *real set* ⇒ *real* **where**
rinf S = (*SOME a*. *isGlb UNIV S a*)

lemma *rinf-alt*: *rinf S* = (*SOME a*. (∀ *x* ∈ *S*. *x* ≥ *a*) ∧ (∀ *b*. (∀ *x* ∈ *S*. *x* ≥ *b*) ⟶ *a* ≥ *b*)) ⟨*proof*⟩

lemma *reals-complete-Glb*: **assumes** *Se*: ∃ *x*. *x* ∈ *S* **and** *lb*: ∃ *y*. *isLb UNIV S y*
shows ∃ (*t*::*real*). *isGlb UNIV S t*
⟨*proof*⟩

lemma *rinf*: **assumes** *Se*: *S* ≠ {} **and** *b*: ∃ *b*. *b* <=∗ *S*
shows *isGlb UNIV S* (*rinf S*)
⟨*proof*⟩

lemma *rinf-ge*: **assumes** *Se*: *S* ≠ {} **and** *Sb*: *b* <=∗ *S* **shows** *rinf S* ≥ *b*
⟨*proof*⟩

lemma *rinf-finite-Min*: **assumes** *fS*: *finite S* **and** *Se*: *S* ≠ {}
shows *rinf S* = *Min S*
⟨*proof*⟩

lemma *rinf-finite-in*: **assumes** *fS*: *finite S* **and** *Se*: *S* ≠ {}
shows *rinf S* ∈ *S*
⟨*proof*⟩

lemma *rinf-finite-Lb*: **assumes** *fS*: *finite S* **and** *Se*: *S* ≠ {}
shows *isLb S S* (*rinf S*)
⟨*proof*⟩

lemma *rinf-finite-ge-iff*: **assumes** *fS*: *finite S* **and** *Se*: *S* ≠ {}
shows *a* ≤ *rinf S* ⟷ (∀ *x* ∈ *S*. *a* ≤ *x*)
⟨*proof*⟩

lemma *rinf-finite-le-iff*: **assumes** *fS*: *finite S* **and** *Se*: *S* ≠ {}
shows *a* ≥ *rinf S* ⟷ (∃ *x* ∈ *S*. *a* ≥ *x*)
⟨*proof*⟩

lemma *rinf-finite-gt-iff*: **assumes** *fS*: *finite S* **and** *Se*: *S* ≠ {}
shows *a* < *rinf S* ⟷ (∀ *x* ∈ *S*. *a* < *x*)
⟨*proof*⟩

lemma *rinf-finite-lt-iff*: **assumes** *fS*: *finite S* **and** *Se*: *S* ≠ {}
shows *a* > *rinf S* ⟷ (∃ *x* ∈ *S*. *a* > *x*)
⟨*proof*⟩

lemma *rinf-unique*: **assumes** *b*: *b* <=∗ *S* **and** *S*: ∀ *b*′ > *b*. ∃ *x* ∈ *S*. *b*′ > *x*
shows *rinf S* = *b*
⟨*proof*⟩

lemma *rinf-ge-subset*: $S \neq \{\} \Longrightarrow S \subseteq T \Longrightarrow (\exists\, b.\ b <=* T) \Longrightarrow rinf\ S >= rinf\ T$
⟨*proof*⟩

lemma *isLb-def'*: $isLb\ R\ S = (\lambda x.\ x <=* S \wedge x \in R)$
⟨*proof*⟩

lemma *rinf-bounds*: **assumes** *Se*: $S \neq \{\}$ **and** *l*: $a <=* S$ **and** *u*: $S *<= b$
shows $a \leq rinf\ S \wedge rinf\ S \leq b$
⟨*proof*⟩

lemma *rinf-abs-ge*: $S \neq \{\} \Longrightarrow (\forall\, x{\in}S.\ |x| \leq a) \Longrightarrow |rinf\ S| \leq a$
⟨*proof*⟩

lemma *rinf-asclose*: **assumes** *S*:$S \neq \{\}$ **and** *b*: $\forall\, x{\in}S.\ |x - l| \leq e$ **shows** $|rinf\ S - l| \leq e$
⟨*proof*⟩

30.18 Operator norm.

definition $onorm\ f = rsup\ \{norm\ (f\ x)|\ x.\ norm\ x = 1\}$

lemma *norm-bound-generalize*:
fixes *f*:: *real ^'n::finite ⇒ real^'m::finite*
assumes *lf*: *linear f*
shows $(\forall\, x.\ norm\ x = 1 \longrightarrow norm\ (f\ x) \leq b) \longleftrightarrow (\forall\, x.\ norm\ (f\ x) \leq b * norm\ x)$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *onorm*:
fixes *f*:: *real ^'n::finite ⇒ real ^'m::finite*
assumes *lf*: *linear f*
shows $norm\ (f\ x) <= onorm\ f * norm\ x$
and $\forall\, x.\ norm\ (f\ x) <= b * norm\ x \Longrightarrow onorm\ f <= b$
⟨*proof*⟩

lemma *onorm-pos-le*: **assumes** *lf*: *linear* (*f*::*real ^'n::finite ⇒ real ^'m::finite*) **shows** $0 <= onorm\ f$
⟨*proof*⟩

lemma *onorm-eq-0*: **assumes** *lf*: *linear* (*f*::*real ^'n::finite ⇒ real ^'m::finite*)
shows $onorm\ f = 0 \longleftrightarrow (\forall\, x.\ f\ x = 0)$
⟨*proof*⟩

lemma *onorm-const*: $onorm(\lambda x{::}real\hat{}'n{::}finite.\ (y{::}real\ \hat{}\ 'm{::}finite)) = norm\ y$
⟨*proof*⟩

lemma *onorm-pos-lt*: **assumes** *lf*: *linear* (*f*::*real ^ 'n::finite ⇒ real ^'m::finite*)
shows $0 < onorm\ f \longleftrightarrow \sim(\forall\, x.\ f\ x = 0)$

⟨*proof*⟩

lemma *onorm-compose*:
assumes *lf*: *linear* (*f*::*real* ^'*n*::*finite* ⇒ *real* ^'*m*::*finite*)
and *lg*: *linear* (*g*::*real*^'*k*::*finite* ⇒ *real*^'*n*::*finite*)
shows *onorm* (*f o g*) <= *onorm f* ∗ *onorm g*
⟨*proof*⟩

lemma *onorm-neg-lemma*: **assumes** *lf*: *linear* (*f*::*real* ^'*n*::*finite* ⇒ *real*^'*m*::*finite*)
shows *onorm* (λ*x*. − *f x*) ≤ *onorm f*
⟨*proof*⟩

lemma *onorm-neg*: **assumes** *lf*: *linear* (*f*::*real* ^'*n*::*finite* ⇒ *real*^'*m*::*finite*)
shows *onorm* (λ*x*. − *f x*) = *onorm f*
⟨*proof*⟩

lemma *onorm-triangle*:
assumes *lf*: *linear* (*f*::*real* ^'*n*::*finite* ⇒ *real* ^'*m*::*finite*) **and** *lg*: *linear g*
shows *onorm* (λ*x*. *f x* + *g x*) <= *onorm f* + *onorm g*
⟨*proof*⟩

lemma *onorm-triangle-le*: *linear* (*f*::*real* ^'*n*::*finite* ⇒ *real* ^'*m*::*finite*) ⟹ *linear* *g* ⟹ *onorm*(*f*) + *onorm*(*g*) <= *e*
⟹ *onorm*(λ*x*. *f x* + *g x*) <= *e*
⟨*proof*⟩

lemma *onorm-triangle-lt*: *linear* (*f*::*real* ^'*n*::*finite* ⇒ *real* ^'*m*::*finite*) ⟹ *linear* *g* ⟹ *onorm*(*f*) + *onorm*(*g*) < *e*
==> *onorm*(λ*x*. *f x* + *g x*) < *e*
⟨*proof*⟩

definition *vec1*:: '*a* ⇒ '*a* ^ *1* **where** *vec1 x* = (χ *i*. *x*)

definition *dest-vec1*:: '*a* ^*1* ⇒ '*a*
where *dest-vec1 x* = (*x*$*1*)

lemma *vec1-component*[*simp*]: (*vec1 x*)$*1* = *x*
⟨*proof*⟩

lemma *vec1-dest-vec1*[*simp*]: *vec1*(*dest-vec1 x*) = *x dest-vec1*(*vec1 y*) = *y*
⟨*proof*⟩

lemma *forall-vec1*: (∀ *x*. *P x*) ⟷ (∀ *x*. *P* (*vec1 x*)) ⟨*proof*⟩

lemma *exists-vec1*: (∃ *x*. *P x*) ⟷ (∃ *x*. *P*(*vec1 x*)) ⟨*proof*⟩

lemma *forall-dest-vec1*: (∀ *x*. *P x*) ⟷ (∀ *x*. *P*(*dest-vec1 x*)) ⟨*proof*⟩

lemma *exists-dest-vec1*: $(\exists\, x.\ P\ x) \longleftrightarrow (\exists\, x.\ P(\textit{dest-vec1}\ x))\langle proof\rangle$

lemma *vec1-eq*[*simp*]: $\textit{vec1}\ x = \textit{vec1}\ y \longleftrightarrow x = y\ \langle proof\rangle$

lemma *dest-vec1-eq*[*simp*]: $\textit{dest-vec1}\ x = \textit{dest-vec1}\ y \longleftrightarrow x = y\ \langle proof\rangle$

lemma *vec1-in-image-vec1*: $\textit{vec1}\ x \in (\textit{vec1}\ `\ S) \longleftrightarrow x \in S\ \langle proof\rangle$

lemma *vec1-vec*: $\textit{vec1}\ x = \textit{vec}\ x\ \langle proof\rangle$

lemma *vec1-add*: $\textit{vec1}(x + y) = \textit{vec1}\ x + \textit{vec1}\ y\ \langle proof\rangle$
lemma *vec1-sub*: $\textit{vec1}(x - y) = \textit{vec1}\ x - \textit{vec1}\ y\ \langle proof\rangle$
lemma *vec1-cmul*: $\textit{vec1}(c* x) = c *s \textit{vec1}\ x\ \langle proof\rangle$
lemma *vec1-neg*: $\textit{vec1}(- x) = - \textit{vec1}\ x\ \langle proof\rangle$

lemma *vec1-setsum*: **assumes** *fS*: *finite S*
shows $\textit{vec1}(\textit{setsum}\ f\ S) = \textit{setsum}\ (\textit{vec1}\ o\ f)\ S$
$\langle proof\rangle$

lemma *dest-vec1-lambda*: $\textit{dest-vec1}(\chi\ i.\ x\ i) = x\ 1$
$\langle proof\rangle$

lemma *dest-vec1-vec*: $\textit{dest-vec1}(\textit{vec}\ x) = x$
$\langle proof\rangle$

lemma *dest-vec1-add*: $\textit{dest-vec1}(x + y) = \textit{dest-vec1}\ x + \textit{dest-vec1}\ y$
$\langle proof\rangle$

lemma *dest-vec1-sub*: $\textit{dest-vec1}(x - y) = \textit{dest-vec1}\ x - \textit{dest-vec1}\ y$
$\langle proof\rangle$

lemma *dest-vec1-cmul*: $\textit{dest-vec1}(c*sx) = c * \textit{dest-vec1}\ x$
$\langle proof\rangle$

lemma *dest-vec1-neg*: $\textit{dest-vec1}(- x) = - \textit{dest-vec1}\ x$
$\langle proof\rangle$

lemma *dest-vec1-0*[*simp*]: $\textit{dest-vec1}\ 0 = 0\ \langle proof\rangle$

lemma *dest-vec1-sum*: **assumes** *fS*: *finite S*
shows $\textit{dest-vec1}(\textit{setsum}\ f\ S) = \textit{setsum}\ (\textit{dest-vec1}\ o\ f)\ S$
$\langle proof\rangle$

lemma *norm-vec1*: $\textit{norm}(\textit{vec1}\ x) = \textit{abs}(x)$
$\langle proof\rangle$

lemma *dist-vec1*: $\textit{dist}(\textit{vec1}\ x)\ (\textit{vec1}\ y) = \textit{abs}(x - y)$
$\langle proof\rangle$

lemma *abs-dest-vec1*: *norm x* = |*dest-vec1 x*|
⟨*proof*⟩

lemma *linear-vmul-dest-vec1*:
fixes *f*:: *'a*::*semiring-1*^*'n* ⇒ *'a*^*1*
shows *linear f* ⟹ *linear* (λ*x*. *dest-vec1*(*f x*) ∗*s v*)
⟨*proof*⟩

lemma *linear-from-scalars*:
assumes *lf*: *linear* (*f*::*'a*::*comm-ring-1* ^*1* ⇒ *'a*^*'n*)
shows *f* = (λ*x*. *dest-vec1 x* ∗*s column 1* (*matrix f*))
⟨*proof*⟩

lemma *linear-to-scalars*: **assumes** *lf*: *linear* (*f*::*'a*::*comm-ring-1* ^*'n*::*finite* ⇒ *'a*^*1*)
shows *f* = (λ*x*. *vec1*(*row 1* (*matrix f*) · *x*))
⟨*proof*⟩

lemma *dest-vec1-eq-0*: *dest-vec1 x* = *0* ⟷ *x* = *0*
⟨*proof*⟩

lemma *setsum-scalars*: **assumes** *fS*: *finite S*
shows *setsum f S* = *vec1* (*setsum* (*dest-vec1 o f*) *S*)
⟨*proof*⟩

lemma *dest-vec1-wlog-le*: (⋀(*x*::*'a*::*linorder* ^ *1*) *y*. *P x y* ⟷ *P y x*) ⟹ (⋀*x y*. *dest-vec1 x* <= *dest-vec1 y* ==> *P x y*) ⟹ *P x y*
⟨*proof*⟩

Pasting vectors.

lemma *linear-fstcart*: *linear fstcart*
⟨*proof*⟩

lemma *linear-sndcart*: *linear sndcart*
⟨*proof*⟩

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

lemma *fstcart-add*[*simp*]:*fstcart*(*x* + *y*) = *fstcart* (*x*::*'a*::{*plus*,*times*}^(*'b* + *'c*)) + *fstcart y*
⟨*proof*⟩

lemma *fstcart-cmul*[*simp*]:*fstcart*(*c*∗*s x*) = *c*∗*s fstcart* (*x*::*'a*::{*plus*,*times*}^(*'b* + *'c*))
⟨*proof*⟩

lemma *fstcart-neg*[*simp*]:*fstcart*(− *x*) = − *fstcart* (*x*::*'a*::*ring-1*^(*'b* + *'c*))
⟨*proof*⟩

lemma *fstcart-sub*[*simp*]:*fstcart*($x - y$) = *fstcart* (x::'*a*::*ring-1*^('*b* + '*c*)) − *fstcart* *y*
⟨*proof*⟩

lemma *fstcart-setsum*:
fixes *f*:: '*d* ⇒ '*a*::*semiring-1*^-
assumes *fS*: *finite S*
shows *fstcart* (*setsum f S*) = *setsum* (λ*i*. *fstcart* (*f i*)) *S*
⟨*proof*⟩

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

lemma *sndcart-add*[*simp*]:*sndcart*($x + y$) = *sndcart* (x::'*a*::{*plus*,*times*}^('*b* + '*c*)) + *sndcart* *y*
⟨*proof*⟩

lemma *sndcart-cmul*[*simp*]:*sndcart*(*c*∗*s x*) = *c*∗*s sndcart* (x::'*a*::{*plus*,*times*}^('*b* + '*c*))
⟨*proof*⟩

lemma *sndcart-neg*[*simp*]:*sndcart*(− *x*) = − *sndcart* (x::'*a*::*ring-1*^('*b* + '*c*))
⟨*proof*⟩

lemma *sndcart-sub*[*simp*]:*sndcart*($x - y$) = *sndcart* (x::'*a*::*ring-1*^('*b* + '*c*)) − *sndcart* *y*
⟨*proof*⟩

lemma *sndcart-setsum*:
fixes *f*:: '*d* ⇒ '*a*::*semiring-1*^-
assumes *fS*: *finite S*
shows *sndcart* (*setsum f S*) = *setsum* (λ*i*. *sndcart* (*f i*)) *S*
⟨*proof*⟩

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

lemma *pastecart-add*[*simp*]:*pastecart* (*x1*::'*a*::{*plus*,*times*}^-) *y1* + *pastecart x2 y2* = *pastecart* (*x1* + *x2*) (*y1* + *y2*)
⟨*proof*⟩

lemma *pastecart-cmul*[*simp*]: *pastecart* (*c* ∗*s* (*x1*::'*a*::{*plus*,*times*}^-)) (*c* ∗*s y1*) = *c* ∗*s pastecart x1 y1*
⟨*proof*⟩

lemma *pastecart-neg*[*simp*]: *pastecart* (− (*x*::'*a*::*ring-1*^-)) (− *y*) = − *pastecart x y*
⟨*proof*⟩

lemma *pastecart-sub*: *pastecart* (*x1*::'*a*::*ring-1*^-) *y1* − *pastecart x2 y2* = *pastecart* (*x1* − *x2*) (*y1* − *y2*)
⟨*proof*⟩

lemma *pastecart-setsum*:
fixes *f*:: '*d* ⇒ '*a*::*semiring-1*^-
assumes *fS*: *finite S*
shows *pastecart* (*setsum f S*) (*setsum g S*) = *setsum* (λ*i*. *pastecart* (*f i*) (*g i*)) *S*
⟨*proof*⟩

lemma *setsum-Plus*:
⟦*finite A*; *finite B*⟧ ⟹
(∑ *x*∈*A* <+> *B*. *g x*) = (∑ *x*∈*A*. *g* (*Inl x*)) + (∑ *x*∈*B*. *g* (*Inr x*))
⟨*proof*⟩

lemma *setsum-UNIV-sum*:
fixes *g* :: '*a*::*finite* + '*b*::*finite* ⇒ -
shows (∑ *x*∈*UNIV*. *g x*) = (∑ *x*∈*UNIV*. *g* (*Inl x*)) + (∑ *x*∈*UNIV*. *g* (*Inr x*))
⟨*proof*⟩

lemma *norm-fstcart*: *norm*(*fstcart x*) <= *norm* (*x*::*real* ^('*n*::*finite* + '*m*::*finite*))
⟨*proof*⟩

lemma *dist-fstcart*: *dist*(*fstcart* (*x*::*real*^-)) (*fstcart y*) <= *dist x y*
⟨*proof*⟩

lemma *norm-sndcart*: *norm*(*sndcart x*) <= *norm* (*x*::*real* ^('*n*::*finite* + '*m*::*finite*))
⟨*proof*⟩

lemma *dist-sndcart*: *dist*(*sndcart* (*x*::*real*^-)) (*sndcart y*) <= *dist x y*
⟨*proof*⟩

lemma *dot-pastecart*: (*pastecart* (*x1*::'*a*::{*times*,*comm-monoid-add*} ^'*n*::*finite*) (*x2*::'*a*::{*times*,*comm-monoid-*
· (*pastecart y1 y2*) = *x1* · *y1* + *x2* · *y2*
⟨*proof*⟩

lemma *norm-pastecart*: *norm*(*pastecart x y*) <= *norm*(*x* :: *real* ^ '*m*::*finite*) + *norm*(*y*::*real*^'*n*::*finite*)
⟨*proof*⟩

30.19 A generic notion of "hull" (convex, affine, conic hull and closure).

definition *hull* :: '*a set set* ⇒ '*a set* ⇒ '*a set* (**infixl** *hull* 75) **where**
S hull s = *Inter* {*t*. *t* ∈ *S* ∧ *s* ⊆ *t*}

lemma *hull-same*: *s* ∈ *S* ⟹ *S hull s* = *s*
⟨*proof*⟩

lemma *hull-in*: $(\bigwedge T.\ T \subseteq S \Longrightarrow Inter\ T \in S) \Longrightarrow (S\ hull\ s) \in S$
⟨*proof*⟩

lemma *hull-eq*: $(\bigwedge T.\ T \subseteq S \Longrightarrow Inter\ T \in S) \Longrightarrow (S\ hull\ s) = s \longleftrightarrow s \in S$
⟨*proof*⟩

lemma *hull-hull*: $S\ hull\ (S\ hull\ s) = S\ hull\ s$
⟨*proof*⟩

lemma *hull-subset*: $s \subseteq (S\ hull\ s)$
⟨*proof*⟩

lemma *hull-mono*: $s \subseteq t \Longrightarrow (S\ hull\ s) \subseteq (S\ hull\ t)$
⟨*proof*⟩

lemma *hull-antimono*: $S \subseteq T \Longrightarrow (T\ hull\ s) \subseteq (S\ hull\ s)$
⟨*proof*⟩

lemma *hull-minimal*: $s \subseteq t \Longrightarrow t \in S \Longrightarrow (S\ hull\ s) \subseteq t$
⟨*proof*⟩

lemma *subset-hull*: $t \in S \Longrightarrow S\ hull\ s \subseteq t \longleftrightarrow s \subseteq t$
⟨*proof*⟩

lemma *hull-unique*: $s \subseteq t \Longrightarrow t \in S \Longrightarrow (\bigwedge t'.\ s \subseteq t' \Longrightarrow t' \in S \Longrightarrow t \subseteq t') \Longrightarrow (S\ hull\ s = t)$
⟨*proof*⟩

lemma *hull-induct*: $(\bigwedge x.\ x \in S \Longrightarrow P\ x) \Longrightarrow Q\ \{x.\ P\ x\} \Longrightarrow \forall x \in Q\ hull\ S.\ P\ x$
⟨*proof*⟩

lemma *hull-inc*: $x \in S \Longrightarrow x \in P\ hull\ S$ ⟨*proof*⟩

lemma *hull-union-subset*: $(S\ hull\ s) \cup (S\ hull\ t) \subseteq (S\ hull\ (s \cup t))$
⟨*proof*⟩

lemma *hull-union*: **assumes** T: $\bigwedge T.\ T \subseteq S \Longrightarrow Inter\ T \in S$
shows $S\ hull\ (s \cup t) = S\ hull\ (S\ hull\ s \cup S\ hull\ t)$
⟨*proof*⟩

lemma *hull-redundant-eq*: $a \in (S\ hull\ s) \longleftrightarrow (S\ hull\ (insert\ a\ s) = S\ hull\ s)$
⟨*proof*⟩

lemma *hull-redundant*: $a \in (S\ hull\ s) \Longrightarrow (S\ hull\ (insert\ a\ s) = S\ hull\ s)$
⟨*proof*⟩

Archimedian properties and useful consequences.

lemma *real-arch-simple*: $\exists n.\ x <= real\ (n::nat)$

⟨*proof*⟩
lemmas *real-arch-lt* = *reals-Archimedean2*

lemmas *real-arch* = *reals-Archimedean3*

lemma *real-arch-inv*: $0 < e \longleftrightarrow (\exists\, n{::}nat.\ n \neq 0 \wedge 0 < inverse\ (real\ n) \wedge inverse\ (real\ n) < e)$
⟨*proof*⟩

lemma *real-pow-lbound*: $0 <= x ==> 1 + real\ n * x <= (1 + x)\ \hat{}\ n$
⟨*proof*⟩

lemma *real-arch-pow*: **assumes** x: $1 < (x{::}real)$ **shows** $\exists\, n.\ y < x\hat{}n$
⟨*proof*⟩

lemma *real-arch-pow2*: $\exists\, n.\ (x{::}real) < 2\hat{}\ n$
⟨*proof*⟩

lemma *real-arch-pow-inv*: **assumes** y: $(y{::}real) > 0$ **and** *x1*: $x < 1$
shows $\exists\, n.\ x\hat{}n < y$
⟨*proof*⟩

lemma *forall-pos-mono*: $(\bigwedge d\ e{::}real.\ d < e \Longrightarrow P\ d ==> P\ e) \Longrightarrow (\bigwedge n{::}nat.\ n \neq 0 ==> P(inverse(real\ n))) \Longrightarrow (\bigwedge e.\ 0 < e ==> P\ e)$
⟨*proof*⟩

lemma *forall-pos-mono-1*: $(\bigwedge d\ e{::}real.\ d < e \Longrightarrow P\ d ==> P\ e) \Longrightarrow (\bigwedge n.\ P(inverse(real\ (Suc\ n)))) ==> 0 < e ==> P\ e$
⟨*proof*⟩

lemma *real-archimedian-rdiv-eq-0*: **assumes** *x0*: $x \geq 0$ **and** c: $c \geq 0$ **and** xc: $\forall\, (m{::}nat){>}0.\ real\ m * x \leq c$
shows $x = 0$
⟨*proof*⟩

lemma *real-max-rsup*: $max\ x\ y = rsup\ \{x,y\}$
⟨*proof*⟩

lemma *real-min-rinf*: $min\ x\ y = rinf\ \{x,y\}$
⟨*proof*⟩

lemma *sum-gp-basic*: $((1::'a::\{field, recpower\}) - x) * setsum\ (\lambda i.\ x\hat{}i)\ \{0\ ..\ n\} = (1 - x\hat{}(Suc\ n))$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *sum-gp-multiplied*: **assumes** *mn*: $m <= n$
shows $((1::'a::\{field, recpower\}) - x) * setsum\ (op\ \hat{}\ x)\ \{m..n\} = x\hat{}m - x\hat{}Suc\ n$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *sum-gp*: $setsum\ (op\ \hat{}\ (x::'a::\{field, recpower\}))\ \{m\ ..\ n\} =$
$(if\ n < m\ then\ 0\ else\ if\ x = 1\ then\ of\text{-}nat\ ((n + 1) - m)$
$else\ (x\hat{}\ m - x\hat{}\ (Suc\ n))\ /\ (1 - x))$
⟨*proof*⟩

lemma *sum-gp-offset*: $setsum\ (op\ \hat{}\ (x::'a::\{field,recpower\}))\ \{m\ ..\ m+n\} =$
$(if\ x = 1\ then\ of\text{-}nat\ n + 1\ else\ x\hat{}m * (1 - x\hat{}Suc\ n)\ /\ (1 - x))$
⟨*proof*⟩

30.20 A bit of linear algebra.

definition $subspace\ S \longleftrightarrow 0 \in S \wedge (\forall x{\in}\ S.\ \forall y \in S.\ x + y \in S) \wedge (\forall c.\ \forall x \in S.\ c *s x \in S\ )$
definition $span\ S = (subspace\ hull\ S)$
definition $dependent\ S \longleftrightarrow (\exists a \in S.\ a \in span(S - \{a\}))$
abbreviation $independent\ s == \sim(dependent\ s)$

lemma *subspace-UNIV*[*simp*]: $subspace(UNIV)$ ⟨*proof*⟩

lemma *subspace-0*: $subspace\ S ==> 0 \in S$ ⟨*proof*⟩

lemma *subspace-add*: $subspace\ S \Longrightarrow x \in S \Longrightarrow y \in S ==> x + y \in S$
⟨*proof*⟩

lemma *subspace-mul*: $subspace\ S \Longrightarrow x \in S \Longrightarrow c *s x \in S$
⟨*proof*⟩

lemma *subspace-neg*: $subspace\ S \Longrightarrow (x::'a::ring\text{-}1\hat{}'n) \in S \Longrightarrow - x \in S$
⟨*proof*⟩

lemma *subspace-sub*: $subspace\ S \Longrightarrow (x::'a::ring\text{-}1\hat{}'n) \in S \Longrightarrow y \in S \Longrightarrow x - y \in S$
⟨*proof*⟩

lemma *subspace-setsum*:
assumes *sA*: $subspace\ A$ **and** *fB*: $finite\ B$

and f: $\forall x{\in}\ B.\ f\ x \in A$
shows *setsum* $f\ B \in A$
⟨*proof*⟩

lemma *subspace-linear-image*:
assumes *lf*: *linear* (f::'*a*::*semiring-1*^'*n* ⇒ -) **and** *sS*: *subspace* S
shows *subspace*(f ' S)
⟨*proof*⟩

lemma *subspace-linear-preimage*: *linear* (f::'*a*::*semiring-1*^'*n* ⇒ -) ==> *subspace* S ==> *subspace* $\{x.\ f\ x \in S\}$
⟨*proof*⟩

lemma *subspace-trivial*: *subspace* {*0*::'*a*::*semiring-1* ^-}
⟨*proof*⟩

lemma *subspace-inter*: *subspace* $A \Longrightarrow$ *subspace* B ==> *subspace* $(A \cap B)$
⟨*proof*⟩

lemma *span-mono*: $A \subseteq B$ ==> *span* $A \subseteq$ *span* B
⟨*proof*⟩

lemma *subspace-span*: *subspace*(*span* S)
⟨*proof*⟩

lemma *span-clauses*:
$a \in S$ ==> $a \in$ *span* S
0 ∈ *span* S
$x{\in}$ *span* $S \Longrightarrow y \in$ *span* S ==> $x + y \in$ *span* S
$x \in$ *span* $S \Longrightarrow c *s\ x \in$ *span* S
⟨*proof*⟩

lemma *span-induct*: **assumes** *SP*: $\bigwedge x.\ x \in S$ ==> $P\ x$
and P: *subspace* P **and** x: $x \in$ *span* S **shows** $P\ x$
⟨*proof*⟩

lemma *span-empty*: *span* {} = {(*0*::'*a*::*semiring-0* ^ '*n*)}
⟨*proof*⟩

lemma *independent-empty*: *independent* {}
⟨*proof*⟩

lemma *independent-mono*: *independent* $A \Longrightarrow B \subseteq A$ ==> *independent* B
⟨*proof*⟩

lemma *span-subspace*: $A \subseteq B \Longrightarrow B \leq$ *span* $A \Longrightarrow$ *subspace* $B \Longrightarrow$ *span* $A = B$
⟨*proof*⟩

lemma *span-induct'*: **assumes** *SP*: $\forall x \in S.\ P\ x$
and *P*: *subspace P* **shows** $\forall x \in span\ S.\ P\ x$
⟨*proof*⟩

inductive *span-induct-alt-help* **for** *S*:: *'a*::*semiring-1^'n* ⇒ *bool*
where
span-induct-alt-help-0: *span-induct-alt-help S 0*
| *span-induct-alt-help-S*: $x \in S \Longrightarrow$ *span-induct-alt-help S z* ⟹ *span-induct-alt-help S* $(c *s\ x + z)$

lemma *span-induct-alt'*:
assumes *h0*: *h* $(0::'a::semiring\text{-}1\hat{}'n)$ **and** *hS*: $\bigwedge c\ x\ y.\ x \in S \Longrightarrow h\ y \Longrightarrow h\ (c*s\ x + y)$ **shows** $\forall x \in span\ S.\ h\ x$
⟨*proof*⟩

lemma *span-induct-alt*:
assumes *h0*: *h* $(0::'a::semiring\text{-}1\hat{}'n)$ **and** *hS*: $\bigwedge c\ x\ y.\ x \in S \Longrightarrow h\ y \Longrightarrow h\ (c*s\ x + y)$ **and** *x*: $x \in span\ S$
shows *h x*
⟨*proof*⟩

lemma *span-superset*: $x \in S$ ==> $x \in span\ S$ ⟨*proof*⟩

lemma *span-0*: $0 \in span\ S$ ⟨*proof*⟩

lemma *span-add*: $x \in span\ S \Longrightarrow y \in span\ S$ ==> $x + y \in span\ S$
⟨*proof*⟩

lemma *span-mul*: $x \in span\ S$ ==> $(c *s\ x) \in span\ S$
⟨*proof*⟩

lemma *span-neg*: $x \in span\ S$ ==> $-\ (x::'a::ring\text{-}1\hat{}'n) \in span\ S$
⟨*proof*⟩

lemma *span-sub*: $(x::'a::ring\text{-}1\hat{}'n) \in span\ S \Longrightarrow y \in span\ S$ ==> $x - y \in span\ S$
⟨*proof*⟩

lemma *span-setsum*: *finite A* ⟹ $\forall x \in A.\ f\ x \in span\ S$ ==> *setsum f A* $\in span\ S$
⟨*proof*⟩

lemma *span-add-eq*: $(x::'a::ring\text{-}1\hat{}'n) \in span\ S \Longrightarrow x + y \in span\ S \longleftrightarrow y \in span\ S$
⟨*proof*⟩

lemma *span-linear-image*: **assumes** *lf*: *linear* (*f*::*'a*::*semiring-1* ^ *'n* => -)
shows *span* (*f* ' *S*) = *f* ' (*span S*)
⟨*proof*⟩

lemma *span-breakdown*:
assumes *bS*: (*b*::*'a*::*ring-1* ^ *'n*) ∈ *S* **and** *aS*: *a* ∈ *span S*
shows ∃ *k*. *a* − *k*∗*s b* ∈ *span* (*S* − {*b*}) (**is** *?P a*)
⟨*proof*⟩

lemma *span-breakdown-eq*:
(*x*::*'a*::*ring-1*^*'n*) ∈ *span* (*insert a S*) ⟷ (∃ *k*. (*x* − *k* ∗*s a*) ∈ *span S*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *in-span-insert*:
assumes *a*: (*a*::*'a*::*field*^*'n*) ∈ *span* (*insert b S*) **and** *na*: *a* ∉ *span S*
shows *b* ∈ *span* (*insert a S*)
⟨*proof*⟩

lemma *in-span-delete*:
assumes *a*: (*a*::*'a*::*field*^*'n*) ∈ *span S*
and *na*: *a* ∉ *span* (*S*−{*b*})
shows *b* ∈ *span* (*insert a* (*S* − {*b*}))
⟨*proof*⟩

lemma *span-trans*:
assumes *x*: (*x*::*'a*::*ring-1*^*'n*) ∈ *span S* **and** *y*: *y* ∈ *span* (*insert x S*)
shows *y* ∈ *span S*
⟨*proof*⟩

lemma *span-explicit*:
span P = {*y*::*'a*::*semiring-1*^*'n*. ∃ *S u*. *finite S* ∧ *S* ⊆ *P* ∧ *setsum* (λ*v*. *u v* ∗*s v*) *S* = *y*}
(**is** - = *?E* **is** - = {*y*. *?h y*} **is** - = {*y*. ∃ *S u*. *?Q S u y*})
⟨*proof*⟩

lemma *dependent-explicit*:
dependent P ⟷ (∃ *S u*. *finite S* ∧ *S* ⊆ *P* ∧ (∃ (*v*::*'a*::{*idom*,*field*}^*'n*) ∈*S*. *u v*

$\neq 0 \wedge setsum\ (\lambda v.\ u\ v *s v)\ S = 0))$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *span-finite*:
assumes *fS*: *finite S*
shows *span S* = $\{(y::'a::semiring\text{-}1\hat{}'n).\ \exists u.\ setsum\ (\lambda v.\ u\ v *s v)\ S = y\}$
(**is** - = *?rhs*)
⟨*proof*⟩

lemma *span-stdbasis*:*span* $\{basis\ i :: 'a::ring\text{-}1\hat{}'n::finite \mid i.\ i \in (UNIV :: 'n\ set)\}$
= *UNIV*
⟨*proof*⟩

lemma *has-size-stdbasis*: $\{basis\ i ::real\ \hat{}'n::finite \mid i.\ i \in (UNIV :: 'n\ set)\}$ *hassize*
CARD('*n*) (**is** *?S hassize ?n*)
⟨*proof*⟩

lemma *finite-stdbasis*: *finite* $\{basis\ i ::real\hat{}'n::finite \mid i.\ i\in (UNIV:: 'n\ set)\}$
⟨*proof*⟩

lemma *card-stdbasis*: *card* $\{basis\ i ::real\hat{}'n::finite \mid i.\ i\in (UNIV :: 'n\ set)\}$ =
CARD('*n*)
⟨*proof*⟩

lemma *independent-stdbasis-lemma*:
assumes *x*: $(x::'a::semiring\text{-}1\ \hat{}\ 'n) \in span\ (basis\ `\ S)$
and *iS*: $i \notin S$
shows $(x\$i) = 0$
⟨*proof*⟩

lemma *independent-stdbasis*: *independent* $\{basis\ i ::real\hat{}'n::finite \mid i.\ i\in (UNIV ::$
$'n\ set)\}$
⟨*proof*⟩

lemma *independent-insert*:
independent(*insert* $(a::'a::field\ \hat{}'n)\ S) \longleftrightarrow$
(*if* $a \in S$ *then independent S*
else independent $S \wedge a \notin span\ S$) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *mem-delete*: $x \in (A - \{a\}) \longleftrightarrow x \neq a \wedge x \in A$

⟨*proof*⟩

lemma *span-span*: *span* (*span A*) = *span A*
⟨*proof*⟩

lemma *span-inc*: *S* ⊆ *span S*
⟨*proof*⟩

lemma *spanning-subset-independent*:
assumes *BA*: *B* ⊆ *A* **and** *iA*: *independent* (*A*::('*a*::*field* ^ '*n*) *set*)
and *AsB*: *A* ⊆ *span B*
shows *A* = *B*
⟨*proof*⟩

lemma *exchange-lemma*:
assumes *f*:*finite* (*t*:: ('*a*::*field*^'*n*) *set*) **and** *i*: *independent s*
and *sp*:*s* ⊆ *span t*
shows ∃ *t'*. (*t' hassize card t*) ∧ *s* ⊆ *t'* ∧ *t'* ⊆ *s* ∪ *t* ∧ *s* ⊆ *span t'*
⟨*proof*⟩

lemma *independent-span-bound*:
assumes *f*: *finite t* **and** *i*: *independent* (*s*::('*a*::*field*^'*n*) *set*) **and** *sp*:*s* ⊆ *span t*
shows *finite s* ∧ *card s* ≤ *card t*
⟨*proof*⟩

lemma *finite-Atleast-Atmost*[*simp*]: *finite* {*f x* |*x*. *x*∈ {(*i*::'*a*::*finite-intvl-succ*) .. *j*}}
⟨*proof*⟩

lemma *finite-Atleast-Atmost-nat*[*simp*]: *finite* {*f x* |*x*. *x*∈ (*UNIV*::'*a*::*finite set*)}
⟨*proof*⟩

lemma *independent-bound*:
fixes *S*:: (*real*^'*n*::*finite*) *set*
shows *independent S* ⟹ *finite S* ∧ *card S* <= *CARD*('*n*)
⟨*proof*⟩

lemma *dependent-biggerset*: (*finite* (*S*::(*real* ^'*n*::*finite*) *set*) ==> *card S* > *CARD*('*n*))
==> *dependent S*
⟨*proof*⟩

lemma *maximal-independent-subset-extend*:

assumes *sv*: $(S::(real\hat{}'n::finite)\ set) \subseteq V$ **and** *iS*: *independent S*
shows $\exists B.\ S \subseteq B \wedge B \subseteq V \wedge independent\ B \wedge V \subseteq span\ B$
⟨*proof*⟩

lemma *maximal-independent-subset*:
$\exists (B::(real\ \hat{}'n::finite)\ set).\ B \subseteq V \wedge independent\ B \wedge V \subseteq span\ B$
⟨*proof*⟩

definition $dim\ V = (SOME\ n.\ \exists B.\ B \subseteq V \wedge independent\ B \wedge V \subseteq span\ B \wedge (B\ hassize\ n))$

lemma *basis-exists*: $\exists B.\ (B :: (real\ \hat{}'n::finite)\ set) \subseteq V \wedge independent\ B \wedge V \subseteq span\ B \wedge (B\ hassize\ dim\ V)$
⟨*proof*⟩

lemma *independent-card-le-dim*: $(B::(real\ \hat{}'n::finite)\ set) \subseteq V \Longrightarrow independent\ B \Longrightarrow finite\ B \wedge card\ B \leq dim\ V$
⟨*proof*⟩

lemma *span-card-ge-dim*: $(B::(real\ \hat{}'n::finite)\ set) \subseteq V \Longrightarrow V \subseteq span\ B \Longrightarrow finite\ B \Longrightarrow dim\ V \leq card\ B$
⟨*proof*⟩

lemma *basis-card-eq-dim*:
$B \subseteq (V::(real\ \hat{}'n::finite)\ set) \Longrightarrow V \subseteq span\ B \Longrightarrow independent\ B \Longrightarrow finite\ B \wedge card\ B = dim\ V$
⟨*proof*⟩

lemma *dim-unique*: $(B::(real\ \hat{}'n::finite)\ set) \subseteq V \Longrightarrow V \subseteq span\ B \Longrightarrow independent\ B \Longrightarrow B\ hassize\ n \Longrightarrow dim\ V = n$
⟨*proof*⟩

lemma *dim-univ*: $dim\ (UNIV :: (real\hat{}'n::finite)\ set) = CARD('n)$
⟨*proof*⟩

lemma *dim-subset*:
$(S::(real\ \hat{}'n::finite)\ set) \subseteq T \Longrightarrow dim\ S \leq dim\ T$
⟨*proof*⟩

lemma *dim-subset-univ*: $dim\ (S::(real\hat{}'n::finite)\ set) \leq CARD('n)$
⟨*proof*⟩

lemma *card-ge-dim-independent*:
assumes *BV*:(*B*::(*real* ^'*n*::*finite*) *set*) ⊆ *V* **and** *iB*:*independent B* **and** *dVB*:*dim V* ≤ *card B*
shows *V* ⊆ *span B*
⟨*proof*⟩

lemma *card-le-dim-spanning*:
assumes *BV*: (*B*:: (*real* ^'*n*::*finite*) *set*) ⊆ *V* **and** *VB*: *V* ⊆ *span B*
and *fB*: *finite B* **and** *dVB*: *dim V* ≥ *card B*
shows *independent B*
⟨*proof*⟩

lemma *card-eq-dim*: (*B*:: (*real* ^'*n*::*finite*) *set*) ⊆ *V* ⟹ *B hassize dim V* ⟹ *independent B* ⟷ *V* ⊆ *span B*
⟨*proof*⟩

lemma *independent-bound-general*:
independent (*S*:: (*real*^'*n*::*finite*) *set*) ⟹ *finite S* ∧ *card S* ≤ *dim S*
⟨*proof*⟩

lemma *dependent-biggerset-general*: (*finite* (*S*:: (*real*^'*n*::*finite*) *set*) ⟹ *card S* > *dim S*) ⟹ *dependent S*
⟨*proof*⟩

lemma *dim-span*: *dim* (*span* (*S*:: (*real* ^'*n*::*finite*) *set*)) = *dim S*
⟨*proof*⟩

lemma *subset-le-dim*: (*S*:: (*real* ^'*n*::*finite*) *set*) ⊆ *span T* ⟹ *dim S* ≤ *dim T*
⟨*proof*⟩

lemma *span-eq-dim*: *span* (*S*:: (*real* ^'*n*::*finite*) *set*) = *span T* ==> *dim S* = *dim T*
⟨*proof*⟩

lemma *spans-image*:
assumes *lf*: *linear* (*f*::'*a*::*semiring-1*^'*n* ⇒ -) **and** *VB*: *V* ⊆ *span B*
shows *f* ‘ *V* ⊆ *span* (*f* ‘ *B*)
⟨*proof*⟩

lemma *dim-image-le*:
fixes *f* :: *real*^'*n*::*finite* ⇒ *real*^'*m*::*finite*
assumes *lf*: *linear f* **shows** *dim* (*f* ‘ *S*) ≤ *dim* (*S*:: (*real* ^'*n*::*finite*) *set*)
⟨*proof*⟩

lemma *spanning-surjective-image*:
assumes *us*: *UNIV* ⊆ *span* (*S*:: (*'a*::*semiring-1* ^ *'n*) *set*)
and *lf*: *linear f* **and** *sf*: *surj f*
shows *UNIV* ⊆ *span* (*f* ' *S*)
⟨*proof*⟩

lemma *independent-injective-image*:
assumes *iS*: *independent* (*S*::(*'a*::*semiring-1*^*'n*) *set*) **and** *lf*: *linear f* **and** *fi*: *inj f*
shows *independent* (*f* ' *S*)
⟨*proof*⟩

definition *pairwise R S* ⟷ (∀ *x* ∈ *S*. ∀ *y*∈ *S*. *x*≠*y* ⟶ *R x y*)

lemma *vector-sub-project-orthogonal*: (*b*::*'a*::*ordered-field*^*'n*::*finite*) · (*x* − ((*b* · *x*) / (*b*·*b*)) ∗*s* *b*) = *0*
⟨*proof*⟩

lemma *basis-orthogonal*:
fixes *B* :: (*real* ^ *'n*::*finite*) *set*
assumes *fB*: *finite B*
shows ∃ *C*. *finite C* ∧ *card C* ≤ *card B* ∧ *span C* = *span B* ∧ *pairwise orthogonal C*
(**is** ∃ *C*. *?P B C*)
⟨*proof*⟩
thm *pairwise-def*
⟨*proof*⟩
thm *dot-ladd*
⟨*proof*⟩

lemma *orthogonal-basis-exists*:
fixes *V* :: (*real* ^ *'n*::*finite*) *set*
shows ∃ *B*. *independent B* ∧ *B* ⊆ *span V* ∧ *V* ⊆ *span B* ∧ (*B hassize dim V*) ∧ *pairwise orthogonal B*
⟨*proof*⟩

lemma *span-eq*: *span S* = *span T* ⟷ *S* ⊆ *span T* ∧ *T* ⊆ *span S*
⟨*proof*⟩

lemma *span-not-univ-orthogonal*:
assumes *sU*: *span S* $\neq$ *UNIV*
shows $\exists (a::$ *real* ^'*n*::*finite*$).\ a \neq 0 \wedge (\forall x \in$ *span S*$.\ a \cdot x = 0)$
⟨*proof*⟩

lemma *span-not-univ-subset-hyperplane*:
assumes *SU*: *span S* $\neq$ (*UNIV* ::(*real*^'*n*::*finite*) *set*)
shows $\exists\ a.\ a \neq 0 \wedge$ *span S* $\subseteq \{x.\ a \cdot x = 0\}$
⟨*proof*⟩

lemma *lowdim-subset-hyperplane*:
assumes *d*: *dim S* $<$ *CARD*('*n*::*finite*)
shows $\exists (a::$*real* ^'*n*::*finite*$).\ a\ \neq 0 \wedge$ *span S* $\subseteq \{x.\ a \cdot x = 0\}$
⟨*proof*⟩

lemma *linear-indep-image-lemma*:
assumes *lf*: *linear f* **and** *fB*: *finite B*
and *ifB*: *independent* (*f* ' *B*)
and *fi*: *inj-on f B* **and** *xsB*: *x* $\in$ *span B*
and *fx*: *f* (*x*::'*a*::*field*^'*n*) = *0*
shows *x* = *0*
⟨*proof*⟩

lemma *linear-independent-extend-lemma*:
assumes *fi*: *finite B* **and** *ib*: *independent B*
shows $\exists g.\ (\forall x \in$ *span B*$.\ \forall y \in$ *span B*$.\ g\ ((x::'a::field\hat{}'n) + y) = g\ x + g\ y)$
$\wedge\ (\forall x \in$ *span B*$.\ \forall c.\ g\ (c *s\ x) = c *s\ g\ x)$
$\wedge\ (\forall x \in B.\ g\ x = f\ x)$
⟨*proof*⟩

lemma *linear-independent-extend*:
assumes *iB*: *independent* (*B*:: (*real* ^'*n*::*finite*) *set*)
shows $\exists g.$ *linear g* $\wedge\ (\forall x \in B.\ g\ x = f\ x)$
⟨*proof*⟩

lemma *card-le-inj*: **assumes** *fA*: *finite A* **and** *fB*: *finite B*
and *c*: *card A* $\leq$ *card B* **shows** $(\exists f.\ f$ ' *A* $\subseteq B \wedge$ *inj-on f A*)
⟨*proof*⟩

lemma *card-subset-eq*: **assumes** *fB*: *finite B* **and** *AB*: *A* $\subseteq$ *B* **and**
c: *card A* = *card B*
shows *A* = *B*
⟨*proof*⟩

lemma *subspace-isomorphism*:
assumes *s*: *subspace* (*S*:: (*real* ^'*n*::*finite*) *set*)
and *t*: *subspace* (*T* :: (*real* ^ '*m*::*finite*) *set*)
and *d*: *dim S* = *dim T*
shows ∃*f*. *linear f* ∧ *f* ' *S* = *T* ∧ *inj-on f S*
⟨*proof*⟩

lemma *subspace-kernel*:
assumes *lf*: *linear* (*f*::'*a*::*semiring-1* ^'*n* ⇒ -)
shows *subspace* {*x*. *f x* = *0*}
⟨*proof*⟩

lemma *linear-eq-0-span*:
assumes *lf*: *linear f* **and** *f0*: ∀ *x*∈*B*. *f x* = *0*
shows ∀ *x* ∈ *span B*. *f x* = (*0*::'*a*::*semiring-1* ^'*n*)
⟨*proof*⟩

lemma *linear-eq-0*:
assumes *lf*: *linear f* **and** *SB*: *S* ⊆ *span B* **and** *f0*: ∀ *x*∈*B*. *f x* = *0*
shows ∀ *x* ∈ *S*. *f x* = (*0*::'*a*::*semiring-1*^'*n*)
⟨*proof*⟩

lemma *linear-eq*:
assumes *lf*: *linear* (*f*::'*a*::*ring-1*^'*n* ⇒ -) **and** *lg*: *linear g* **and** *S*: *S* ⊆ *span B*
and *fg*: ∀ *x*∈ *B*. *f x* = *g x*
shows ∀ *x*∈ *S*. *f x* = *g x*
⟨*proof*⟩

lemma *linear-eq-stdbasis*:
assumes *lf*: *linear* (*f*::'*a*::*ring-1*^'*m*::*finite* ⇒ '*a*^'*n*::*finite*) **and** *lg*: *linear g*
and *fg*: ∀ *i*. *f* (*basis i*) = *g*(*basis i*)
shows *f* = *g*
⟨*proof*⟩

lemma *bilinear-eq*:
assumes *bf*: *bilinear* (*f*:: '*a*::*ring*^'*m* ⇒ '*a*^'*n* ⇒ '*a*^'*p*)
and *bg*: *bilinear g*
and *SB*: *S* ⊆ *span B* **and** *TC*: *T* ⊆ *span C*
and *fg*: ∀ *x*∈ *B*. ∀ *y*∈ *C*. *f x y* = *g x y*
shows ∀ *x*∈*S*. ∀ *y*∈*T*. *f x y* = *g x y*
⟨*proof*⟩

lemma *bilinear-eq-stdbasis*:
assumes *bf*: *bilinear* (*f*:: '*a*::*ring-1*^'*m*::*finite* ⇒ '*a*^'*n*::*finite* ⇒ '*a*^'*p*)

and *bg*: *bilinear g*
and *fg*: $\forall\, i\ j.\ f\ (basis\ i)\ (basis\ j) = g\ (basis\ i)\ (basis\ j)$
shows $f = g$
⟨*proof*⟩

lemma *left-invertible-transp*:
$(\exists\,(B::'a\hat{}'n\hat{}'m).\ B\ **\ transp\ (A::'a\hat{}'n\hat{}'m) = mat\ (1::'a::comm\text{-}semiring\text{-}1)) \longleftrightarrow (\exists\,(B::'a\hat{}'m\hat{}'n).\ A\ **\ B = mat\ 1)$
⟨*proof*⟩

lemma *right-invertible-transp*:
$(\exists\,(B::'a\hat{}'n\hat{}'m).\ transp\ (A::'a\hat{}'n\hat{}'m)\ **\ B = mat\ (1::'a::comm\text{-}semiring\text{-}1)) \longleftrightarrow (\exists\,(B::'a\hat{}'m\hat{}'n).\ B\ **\ A = mat\ 1)$
⟨*proof*⟩

lemma *linear-injective-left-inverse*:
assumes *lf*: *linear* $(f::real\ \hat{}'n::finite \Rightarrow real\ \hat{}'m::finite)$ **and** *fi*: *inj f*
shows $\exists\, g.\ linear\ g \wedge g\ o\ f = id$
⟨*proof*⟩

lemma *linear-surjective-right-inverse*:
assumes *lf*: *linear* $(f::real\ \hat{}'m::finite \Rightarrow real\ \hat{}'n::finite)$ **and** *sf*: *surj f*
shows $\exists\, g.\ linear\ g \wedge f\ o\ g = id$
⟨*proof*⟩

lemma *matrix-left-invertible-injective*:
$(\exists\, B.\ (B::real\hat{}'m\hat{}'n)\ **\ (A::real\hat{}'n::finite\hat{}'m::finite) = mat\ 1) \longleftrightarrow (\forall\, x\ y.\ A\ *v\ x = A\ *v\ y \longrightarrow x = y)$
⟨*proof*⟩

lemma *matrix-left-invertible-ker*:
$(\exists\, B.\ (B::real\ \hat{}'m::finite\hat{}'n::finite)\ **\ (A::real\hat{}'n\hat{}'m) = mat\ 1) \longleftrightarrow (\forall\, x.\ A\ *v\ x = 0 \longrightarrow x = 0)$
⟨*proof*⟩

lemma *matrix-right-invertible-surjective*:
$(\exists\, B.\ (A::real\hat{}'n::finite\hat{}'m::finite)\ **\ (B::real\hat{}'m\hat{}'n) = mat\ 1) \longleftrightarrow surj\ (\lambda x.\ A\ *v\ x)$
⟨*proof*⟩

lemma *matrix-left-invertible-independent-columns*:
fixes $A :: real\hat{}'n::finite\hat{}'m::finite$
shows $(\exists\,(B::real\ \hat{}'m\hat{}'n).\ B\ **\ A = mat\ 1) \longleftrightarrow (\forall\, c.\ setsum\ (\lambda i.\ c\ i\ *s\ column\ i\ A)\ (UNIV :: 'n\ set) = 0 \longrightarrow (\forall\, i.\ c\ i = 0))$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *matrix-right-invertible-independent-rows*:
fixes $A :: real\hat{}'n::finite\hat{}'m::finite$
shows $(\exists (B::real\hat{}'m\hat{}'n).\ A **\ B = mat\ 1) \longleftrightarrow (\forall c.\ setsum\ (\lambda i.\ c\ i *s row\ i\ A)\ (UNIV :: 'm\ set) = 0 \longrightarrow (\forall i.\ c\ i = 0))$
⟨*proof*⟩

lemma *matrix-right-invertible-span-columns*:
$(\exists (B::real\ \hat{}'n::finite\hat{}'m::finite).\ (A::real\ \hat{}'m\hat{}'n)\ **\ B = mat\ 1) \longleftrightarrow span\ (columns\ A) = UNIV$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *matrix-left-invertible-span-rows*:
$(\exists (B::real\hat{}'m::finite\hat{}'n::finite).\ B **\ (A::real\hat{}'n\hat{}'m) = mat\ 1) \longleftrightarrow span\ (rows\ A) = UNIV$
⟨*proof*⟩

lemma *linear-injective-imp-surjective*:
assumes *lf*: *linear* $(f::real\ \hat{}'n::finite \Rightarrow real\ \hat{}'n)$ **and** *fi*: *inj f*
shows *surj f*
⟨*proof*⟩

lemma *surjective-iff-injective-gen*:
assumes *fS*: *finite S* **and** *fT*: *finite T* **and** *c*: *card S = card T*
and *ST*: $f\ `\ S \subseteq T$
shows $(\forall y \in T.\ \exists x \in S.\ f\ x = y) \longleftrightarrow$ *inj-on f S* (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *linear-surjective-imp-injective*:
assumes *lf*: *linear* $(f::real\ \hat{}'n::finite => real\ \hat{}'n)$ **and** *sf*: *surj f*
shows *inj f*
⟨*proof*⟩

lemma *left-right-inverse-eq*:
assumes *fg*: $f\ o\ g = id$ **and** *gh*: $g\ o\ h = id$
shows $f = h$
⟨*proof*⟩

lemma *isomorphism-expand*:
$f\ o\ g = id \wedge g\ o\ f = id \longleftrightarrow (\forall x.\ f(g\ x) = x) \wedge (\forall x.\ g(f\ x) = x)$
⟨*proof*⟩

lemma *linear-injective-isomorphism*:
assumes *lf*: *linear* $(f :: real\hat{}'n::finite \Rightarrow real\ \hat{}'n)$ **and** *fi*: *inj f*

shows $\exists f'.\ \mathit{linear}\ f' \wedge (\forall x.\ f'\ (f\ x) = x) \wedge (\forall x.\ f\ (f'\ x) = x)$
⟨*proof*⟩

lemma *linear-surjective-isomorphism*:
assumes *lf*: *linear* (*f*::*real* ^'*n*::*finite* ⇒ *real* ^'*n*) **and** *sf*: *surj f*
shows $\exists f'.\ \mathit{linear}\ f' \wedge (\forall x.\ f'\ (f\ x) = x) \wedge (\forall x.\ f\ (f'\ x) = x)$
⟨*proof*⟩

lemma *linear-inverse-left*:
assumes *lf*: *linear* (*f*::*real* ^'*n*::*finite* ⇒ *real* ^'*n*) **and** *lf'*: *linear f'*
shows $f\ o\ f' = id \longleftrightarrow f'\ o\ f = id$
⟨*proof*⟩

lemma *left-inverse-linear*:
assumes *lf*: *linear* (*f*::*real* ^'*n*::*finite* ⇒ *real* ^'*n*) **and** *gf*: $g\ o\ f = id$
shows *linear g*
⟨*proof*⟩

lemma *right-inverse-linear*:
assumes *lf*: *linear* (*f*:: *real* ^'*n*::*finite* ⇒ *real* ^'*n*) **and** *gf*: $f\ o\ g = id$
shows *linear g*
⟨*proof*⟩

lemma *matrix-left-right-inverse*:
fixes *A A'* :: *real* ^'*n*::*finite*^'*n*
shows $A ** A' = mat\ 1 \longleftrightarrow A' ** A = mat\ 1$
⟨*proof*⟩

definition *rowvector v* = (χ *i j*. (*v*$*j*))

definition *columnvector v* = (χ *i j*. (*v*$*i*))

lemma *transp-columnvector*:
transp(*columnvector v*) = *rowvector v*
⟨*proof*⟩

lemma *transp-rowvector*: *transp*(*rowvector v*) = *columnvector v*
⟨*proof*⟩

lemma *dot-rowvector-columnvector*:
columnvector (*A* ∗*v v*) = *A* ∗∗ *columnvector v*

⟨*proof*⟩

lemma *dot-matrix-product*: (*x*::*'a*::*semiring-1^'n*::*finite*) ⋅ *y* = (((*rowvector x* ::*'a^'n^1*) ** (*columnvector y* :: *'a^1^'n*))$*1*)$*1*
⟨*proof*⟩

lemma *dot-matrix-vector-mul*:
fixes *A B* :: *real ^'n*::*finite ^'n* **and** *x y* :: *real ^'n*
shows (*A* *v *x*) ⋅ (*B* *v *y*) =
(((*rowvector x* :: *real^'n^1*) ** ((*transp A* ** *B*) ** (*columnvector y* :: *real ^1^'n*)))$*1*)$*1*
⟨*proof*⟩

definition *infnorm* (*x*::*real^'n*::*finite*) = *rsup* {*abs*(*x*$*i*) |*i*. *i*∈ (*UNIV* :: *'n set*)}

lemma *numseg-dimindex-nonempty*: ∃ *i*. *i* ∈ (*UNIV* :: *'n set*)
⟨*proof*⟩

lemma *infnorm-set-image*:
{*abs*(*x*$*i*) |*i*. *i*∈ (*UNIV* :: *'n set*)} =
(λ*i*. *abs*(*x*$*i*)) ‘ (*UNIV* :: *'n set*) ⟨*proof*⟩

lemma *infnorm-set-lemma*:
shows *finite* {*abs*((*x*::*'a*::*abs ^'n*::*finite*)$*i*) |*i*. *i*∈ (*UNIV* :: *'n set*)}
and {*abs*(*x*$*i*) |*i*. *i*∈ (*UNIV* :: *'n*::*finite set*)} ≠ {}
⟨*proof*⟩

lemma *infnorm-pos-le*: *0* ≤ *infnorm* (*x*::*real^'n*::*finite*)
⟨*proof*⟩

lemma *infnorm-triangle*: *infnorm* ((*x*::*real^'n*::*finite*) + *y*) ≤ *infnorm x* + *infnorm y*
⟨*proof*⟩

lemma *infnorm-eq-0*: *infnorm x* = *0* ⟷ (*x*::*real ^'n*::*finite*) = *0*
⟨*proof*⟩

lemma *infnorm-0*: *infnorm 0* = *0*
⟨*proof*⟩

lemma *infnorm-neg*: *infnorm* (− *x*) = *infnorm x*
⟨*proof*⟩

lemma *infnorm-sub*: *infnorm* (*x* − *y*) = *infnorm* (*y* − *x*)
⟨*proof*⟩

lemma *real-abs-sub-infnorm*: | *infnorm x* − *infnorm y*| ≤ *infnorm* (*x* − *y*)

⟨*proof*⟩

lemma *real-abs-infnorm*: $|infnorm\ x| = infnorm\ x$
⟨*proof*⟩

lemma *component-le-infnorm*:
shows $|x\$i| \leq infnorm\ (x::real\hat{}'n::finite)$
⟨*proof*⟩

lemma *infnorm-mul-lemma*: $infnorm(a *s x) <= |a| * infnorm\ x$
⟨*proof*⟩

lemma *infnorm-mul*: $infnorm(a *s x) = abs\ a * infnorm\ x$
⟨*proof*⟩

lemma *infnorm-pos-lt*: $infnorm\ x > 0 \longleftrightarrow x \neq 0$
⟨*proof*⟩

lemma *infnorm-le-norm*: $infnorm\ x \leq norm\ x$
⟨*proof*⟩
lemma *card-enum*: $card\ \{1\ ..\ n\} = n$ ⟨*proof*⟩
lemma *norm-le-infnorm*: $norm(x) <= sqrt(real\ CARD('n)) * infnorm(x::real\ \hat{}'n::finite)$
⟨*proof*⟩

lemma *norm-cauchy-schwarz-eq*: $(x::real\ \hat{}'n::finite) \cdot y = norm\ x * norm\ y \longleftrightarrow$
$norm\ x *s y = norm\ y *s x$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *norm-cauchy-schwarz-abs-eq*:
fixes $x\ y :: real\ \hat{}\ 'n::finite$
shows $abs(x \cdot y) = norm\ x * norm\ y \longleftrightarrow$
$norm\ x *s y = norm\ y *s x \vee norm(x) *s y = - norm\ y *s x$ (**is**
?lhs $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *norm-triangle-eq*:
fixes $x\ y :: real\ \hat{}\ 'n::finite$
shows $norm(x + y) = norm\ x + norm\ y \longleftrightarrow norm\ x *s y = norm\ y *s x$
⟨*proof*⟩

definition *collinear* $S \longleftrightarrow (\exists u.\ \forall x \in S.\ \forall\ y \in S.\ \exists c.\ x - y = c *s u)$

lemma *collinear-empty*: *collinear* $\{\}$ ⟨*proof*⟩

lemma *collinear-sing*: *collinear* {(*x*::'*a*::*ring-1*^'*n*)}
⟨*proof*⟩

lemma *collinear-2*: *collinear* {(*x*::'*a*::*ring-1*^'*n*),*y*}
⟨*proof*⟩

lemma *collinear-lemma*: *collinear* {(*0*::*real*^'*n*),*x*,*y*} ⟷ *x* = *0* ∨ *y* = *0* ∨ (∃ *c*. *y* = *c* ∗*s* *x*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *norm-cauchy-schwarz-equal*:
fixes *x y* :: *real* ^ '*n*::*finite*
shows *abs*(*x* ∙ *y*) = *norm x* ∗ *norm y* ⟷ *collinear* {(*0*::*real*^'*n*),*x*,*y*}
⟨*proof*⟩

end

31 Permutations: Permutations, both general and specifically on finite sets.

theory *Permutations*
imports *Finite-Cartesian-Product Parity Fact Main*
begin

definition *permutes* (**infixr** *permutes 41*) **where**
(*p permutes S*) ⟷ (∀ *x*. *x* ∉ *S* ⟶ *p x* = *x*) ∧ (∀ *y*. ∃!*x*. *p x* = *y*)

declare *swap-self*[*simp*]
lemma *swapid-sym*: *Fun.swap a b id* = *Fun.swap b a id*
⟨*proof*⟩
lemma *swap-id-refl*: *Fun.swap a a id* = *id* ⟨*proof*⟩
lemma *swap-id-sym*: *Fun.swap a b id* = *Fun.swap b a id*
⟨*proof*⟩
lemma *swap-id-idempotent*[*simp*]: *Fun.swap a b id o Fun.swap a b id* = *id*
⟨*proof*⟩

lemma *inv-unique-comp*: **assumes** *fg*: *f o g* = *id* **and** *gf*: *g o f* = *id*
shows *inv f* = *g*
⟨*proof*⟩

lemma *inverse-swap-id*: *inv* (*Fun.swap a b id*) = *Fun.swap a b id*
⟨*proof*⟩

lemma *swap-id-eq*: *Fun.swap a b id x* = (*if x* = *a then b else if x* = *b then a else x*)
⟨*proof*⟩

lemma *permutes-in-image*: *p permutes S* ⟹ *p x* ∈ *S* ⟷ *x* ∈ *S*
⟨*proof*⟩

lemma *permutes-image*: **assumes** *pS*: *p permutes S* **shows** *p* ' *S* = *S*
⟨*proof*⟩

lemma *permutes-inj*: *p permutes S* ==> *inj p*
⟨*proof*⟩

lemma *permutes-surj*: *p permutes s* ==> *surj p*
⟨*proof*⟩

lemma *permutes-inv-o*: **assumes** *pS*: *p permutes S*
shows *p o inv p* = *id*
and *inv p o p* = *id*
⟨*proof*⟩

lemma *permutes-inverses*:
fixes *p* :: *'a* ⇒ *'a*
assumes *pS*: *p permutes S*
shows *p* (*inv p x*) = *x*
and *inv p* (*p x*) = *x*
⟨*proof*⟩

lemma *permutes-subset*: *p permutes S* ⟹ *S* ⊆ *T* ==> *p permutes T*
⟨*proof*⟩

lemma *permutes-empty*[*simp*]: *p permutes* {} ⟷ *p* = *id*
⟨*proof*⟩

lemma *permutes-sing*[*simp*]: *p permutes* {*a*} ⟷ *p* = *id*
⟨*proof*⟩

lemma *permutes-univ*: *p permutes UNIV* ⟷ (∀ *y*. ∃!*x*. *p x* = *y*)
⟨*proof*⟩

lemma *permutes-inv-eq*: *p permutes S* ==> *inv p y* = *x* ⟷ *p x* = *y*

⟨*proof*⟩

lemma *permutes-swap-id*: $a \in S \Longrightarrow b \in S \Longrightarrow$ *Fun.swap a b id permutes S*
⟨*proof*⟩

lemma *permutes-superset*: *p permutes S* $\Longrightarrow$ $(\forall x \in S - T.\ p\ x = x) \Longrightarrow$ *p permutes T*
⟨*proof*⟩

lemma *permutes-id*: *id permutes S* ⟨*proof*⟩

lemma *permutes-compose*: *p permutes S* $\Longrightarrow$ *q permutes S* $\Longrightarrow$ *q o p permutes S*
⟨*proof*⟩

lemma *permutes-inv*: **assumes** *pS*: *p permutes S* **shows** *inv p permutes S*
⟨*proof*⟩

lemma *permutes-inv-inv*: **assumes** *pS*: *p permutes S* **shows** *inv (inv p) = p*
⟨*proof*⟩

lemma *permutes-insert-lemma*:
assumes *pS*: *p permutes* (*insert a S*)
shows *Fun.swap a* (*p a*) *id o p permutes S*
⟨*proof*⟩

lemma *permutes-insert*: {*p*. *p permutes* (*insert a S*)} =
(λ(*b*,*p*). *Fun.swap a b id o p*) ' {(*b*,*p*). *b* ∈ *insert a S* ∧ *p* ∈ {*p*. *p permutes S*}}
⟨*proof*⟩

lemma *hassize-insert*: $a \notin F \Longrightarrow$ *insert a F hassize n* $\Longrightarrow$ *F hassize* $(n - 1)$
⟨*proof*⟩

lemma *hassize-permutations*: **assumes** *Sn*: *S hassize n*
shows {*p*. *p permutes S*} *hassize* (*fact n*)
⟨*proof*⟩

lemma *finite-permutations*: *finite S* ⟹ *finite* {*p*. *p permutes S*}
⟨*proof*⟩

lemma (**in** *ab-semigroup-mult*) *fold-image-permute*: **assumes** *fS*: *finite S* **and** *pS*: *p permutes S*
shows *fold-image times f z S = fold-image times (f o p) z S*
⟨*proof*⟩
lemma (**in** *ab-semigroup-add*) *fold-image-permute*: **assumes** *fS*: *finite S* **and** *pS*: *p permutes S*
shows *fold-image plus f z S = fold-image plus (f o p) z S*
⟨*proof*⟩

lemma *setsum-permute*: **assumes** *pS*: *p permutes S*
shows *setsum f S = setsum (f o p) S*
⟨*proof*⟩

lemma *setsum-permute-natseg*:**assumes** *pS*: *p permutes* {*m .. n*}
shows *setsum f* {*m .. n*} = *setsum* (*f o p*) {*m .. n*}
⟨*proof*⟩

lemma *setprod-permute*: **assumes** *pS*: *p permutes S*
shows *setprod f S = setprod (f o p) S*
⟨*proof*⟩

lemma *setprod-permute-natseg*:**assumes** *pS*: *p permutes* {*m .. n*}
shows *setprod f* {*m .. n*} = *setprod* (*f o p*) {*m .. n*}
⟨*proof*⟩

lemma *swap-id-common*: *a* ≠ *c* ⟹ *b* ≠ *c* ⟹ *Fun.swap a b id o Fun.swap a c id = Fun.swap b c id o Fun.swap a b id* ⟨*proof*⟩

lemma *swap-id-common'*: ~(*a = b*) ⟹ ~(*a = c*) ⟹ *Fun.swap a c id o Fun.swap b c id = Fun.swap b c id o Fun.swap a b id* ⟨*proof*⟩

lemma *swap-id-independent*: ~(*a = c*) ⟹ ~(*a = d*) ⟹ ~(*b = c*) ⟹ ~(*b = d*) ==> *Fun.swap a b id o Fun.swap c d id = Fun.swap c d id o Fun.swap a b id*
⟨*proof*⟩

inductive *swapidseq* :: *nat* ⇒ (*'a* ⇒ *'a*) ⇒ *bool* **where**

id[*simp*]: *swapidseq 0 id*
| *comp-Suc*: *swapidseq n p* $\Longrightarrow$ $a \neq b$ $\Longrightarrow$ *swapidseq* (*Suc n*) (*Fun.swap a b id o p*)

declare *id*[*unfolded id-def*, *simp*]
definition *permutation p* $\longleftrightarrow$ ($\exists$ *n*. *swapidseq n p*)

lemma *permutation-id*[*simp*]: *permutation id*⟨*proof*⟩
declare *permutation-id*[*unfolded id-def*, *simp*]

lemma *swapidseq-swap*: *swapidseq* (*if a = b then 0 else 1*) (*Fun.swap a b id*)
⟨*proof*⟩

lemma *permutation-swap-id*: *permutation* (*Fun.swap a b id*)
⟨*proof*⟩

lemma *swapidseq-comp-add*: *swapidseq n p* $\Longrightarrow$ *swapidseq m q* ==> *swapidseq* (*n + m*) (*p o q*)
⟨*proof*⟩

lemma *permutation-compose*: *permutation p* $\Longrightarrow$ *permutation q* ==> *permutation*(*p o q*)
⟨*proof*⟩

lemma *swapidseq-endswap*: *swapidseq n p* $\Longrightarrow$ $a \neq b$ ==> *swapidseq* (*Suc n*) (*p o Fun.swap a b id*)
⟨*proof*⟩

lemma *swapidseq-inverse-exists*: *swapidseq n p* ==> $\exists$ *q*. *swapidseq n q* $\wedge$ *p o q = id* $\wedge$ *q o p = id*
⟨*proof*⟩

lemma *swapidseq-inverse*: **assumes** *H*: *swapidseq n p* **shows** *swapidseq n* (*inv p*)
⟨*proof*⟩

lemma *permutation-inverse*: *permutation p* ==> *permutation* (*inv p*)
⟨*proof*⟩

lemma *symmetry-lemma*:($\bigwedge$*a b c d*. *P a b c d* ==> *P a b d c*) $\Longrightarrow$
($\bigwedge$*a b c d*. $a \neq b \Longrightarrow c \neq d \Longrightarrow$ ($a = c \wedge b = d \vee$ $a = c \wedge b \neq d \vee a \neq c \wedge$

$b = d \vee a \neq c \wedge a \neq d \wedge b \neq c \wedge b \neq d) ==> P\ a\ b\ c\ d)$
$==> (\bigwedge a\ b\ c\ d.\ a \neq b\ --> c \neq d \longrightarrow\ P\ a\ b\ c\ d)$ ⟨*proof*⟩

lemma *swap-general*: $a \neq b \Longrightarrow c \neq d \Longrightarrow$ *Fun.swap a b id o Fun.swap c d id* = *id* ∨
(∃ *x y z*. $x \neq a \wedge y \neq a \wedge z \neq a \wedge x \neq y \wedge$ *Fun.swap a b id o Fun.swap c d id* = *Fun.swap x y id o Fun.swap a z id*)
⟨*proof*⟩

lemma *swapidseq-id-iff*[*simp*]: *swapidseq 0 p* ⟷ *p* = *id*
⟨*proof*⟩

lemma *swapidseq-cases*: *swapidseq n p* ⟷ (*n*=*0* ∧ *p* = *id* ∨ (∃ *a b q m*. *n* = *Suc m* ∧ *p* = *Fun.swap a b id o q* ∧ *swapidseq m q* ∧ *a*≠ *b*))
⟨*proof*⟩
lemma *fixing-swapidseq-decrease*:
assumes *spn*: *swapidseq n p* **and** *ab*: *a*≠*b* **and** *pa*: (*Fun.swap a b id o p*) *a* = *a*
shows *n* ≠ *0* ∧ *swapidseq* (*n* − *1*) (*Fun.swap a b id o p*)
⟨*proof*⟩

lemma *swapidseq-identity-even*:
assumes *swapidseq n* (*id* :: *'a* ⇒ *'a*) **shows** *even n*
⟨*proof*⟩

definition *evenperm p* = *even* (*SOME n*. *swapidseq n p*)

lemma *swapidseq-even-even*: **assumes**
m: *swapidseq m p* **and** *n*: *swapidseq n p*
shows *even m* ⟷ *even n*
⟨*proof*⟩

lemma *evenperm-unique*: **assumes** *p*: *swapidseq n p* **and** *n*:*even n* = *b*
shows *evenperm p* = *b*
⟨*proof*⟩

lemma *evenperm-id*[*simp*]: *evenperm id* = *True*
⟨*proof*⟩

lemma *evenperm-swap*: *evenperm* (*Fun.swap a b id*) = (*a* = *b*)
⟨*proof*⟩

lemma *evenperm-comp*:
  assumes *p*: *permutation p* **and** *q*:*permutation q*
  shows *evenperm* (*p o q*) = (*evenperm p* = *evenperm q*)
⟨*proof*⟩

lemma *evenperm-inv*: **assumes** *p*: *permutation p*
  shows *evenperm* (*inv p*) = *evenperm p*
⟨*proof*⟩

lemma *bij-iff*: *bij f* ⟷ (∀ *x*. ∃!*y*. *f y* = *x*)
  ⟨*proof*⟩

lemma *permutation-bijective*:
  assumes *p*: *permutation p*
  shows *bij p*
⟨*proof*⟩

lemma *permutation-finite-support*: **assumes** *p*: *permutation p*
  shows *finite* {*x*. *p x* ≠ *x*}
⟨*proof*⟩

lemma *bij-inv-eq-iff*: *bij p* ==> *x* = *inv p y* ⟷ *p x* = *y*
  ⟨*proof*⟩

lemma *bij-swap-comp*:
  assumes *bp*: *bij p* **shows** *Fun.swap a b id o p* = *Fun.swap* (*inv p a*) (*inv p b*) *p*
  ⟨*proof*⟩

lemma *bij-swap-ompose-bij*: *bij p* ⟹ *bij* (*Fun.swap a b id o p*)
⟨*proof*⟩

lemma *permutation-lemma*:
  assumes *fS*: *finite S* **and** *p*: *bij p* **and** *pS*: ∀ *x*. *x*∉ *S* ⟶ *p x* = *x*
  shows *permutation p*
⟨*proof*⟩

lemma *permutation*: *permutation p* ⟷ *bij p* ∧ *finite* {*x*. *p x* ≠ *x*}
  (**is** *?lhs* ⟷ *?b* ∧ *?f*)
⟨*proof*⟩

lemma *permutation-inverse-works*: **assumes** *p*: *permutation p*
  shows *inv p o p* = *id p o inv p* = *id*
⟨*proof*⟩

lemma *permutation-inverse-compose*:
assumes *p*: *permutation p* **and** *q*: *permutation q*
shows *inv* (*p o q*) = *inv q o inv p*
⟨*proof*⟩

lemma *permutation-permutes*: *permutation p* ⟷ (∃ *S*. *finite S* ∧ *p permutes S*)
⟨*proof*⟩

lemma *permutes-induct*: *finite S* ⟹ *P id* ⟹ (⋀ *a b p*. *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *P p* ⟹ *P p* ⟹ *permutation p* ==> *P* (*Fun.swap a b id o p*))
==> (⋀*p*. *p permutes S* ==> *P p*)
⟨*proof*⟩

definition *sign p* = (*if evenperm p then* (*1*::*int*) *else* −*1*)

lemma *sign-nz*: *sign p* ≠ *0* ⟨*proof*⟩
lemma *sign-id*: *sign id* = *1* ⟨*proof*⟩
lemma *sign-inverse*: *permutation p* ==> *sign* (*inv p*) = *sign p*
⟨*proof*⟩
lemma *sign-compose*: *permutation p* ⟹ *permutation q* ==> *sign* (*p o q*) = *sign*(*p*) ∗ *sign*(*q*) ⟨*proof*⟩
lemma *sign-swap-id*: *sign* (*Fun.swap a b id*) = (*if a* = *b then 1 else* −*1*)
⟨*proof*⟩
lemma *sign-idempotent*: *sign p* ∗ *sign p* = *1* ⟨*proof*⟩

lemma *permutes-natset-le*:
assumes *p*: *p permutes* (*S*::′*a*::*wellorder set*) **and** *le*: ∀ *i* ∈ *S*. *p i* <= *i* **shows** *p* = *id*
⟨*proof*⟩

lemma *permutes-natset-ge*:
assumes *p*: *p permutes* (*S*::′*a*::*wellorder set*) **and** *le*: ∀ *i* ∈ *S*. *p i* ≥ *i* **shows** *p* = *id*

⟨*proof*⟩

lemma *image-inverse-permutations*: {*inv p* |*p*. *p permutes S*} = {*p*. *p permutes S*}
⟨*proof*⟩

lemma *image-compose-permutations-left*:
assumes *q*: *q permutes S* **shows** {*q o p* | *p*. *p permutes S*} = {*p* . *p permutes S*}
⟨*proof*⟩

lemma *image-compose-permutations-right*:
assumes *q*: *q permutes S*
shows {*p o q* | *p*. *p permutes S*} = {*p* . *p permutes S*}
⟨*proof*⟩

lemma *permutes-in-seg*: *p permutes* {*1* ..*n*} ⟹ *i* ∈ {*1*..*n*} ==> *1* <= *p i* ∧ *p i* <= *n*

⟨*proof*⟩

term *setsum*
lemma *setsum-permutations-inverse*: *setsum f* {*p*. *p permutes S*} = *setsum* (λ*p*. *f*(*inv p*)) {*p*. *p permutes S*} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *setum-permutations-compose-left*:
assumes *q*: *q permutes S*
shows *setsum f* {*p*. *p permutes S*} =
setsum (λ*p*. *f*(*q o p*)) {*p*. *p permutes S*} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *sum-permutations-compose-right*:
assumes *q*: *q permutes S*
shows *setsum f* {*p*. *p permutes S*} =
setsum (λ*p*. *f*(*p o q*)) {*p*. *p permutes S*} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *setsum-over-permutations-insert*:
assumes *fS*: *finite S* **and** *aS*: *a* ∉ *S*
shows *setsum f* {*p*. *p permutes* (*insert a S*)} = *setsum* (λ*b*. *setsum* (λ*q*. *f* (*Fun.swap a b id o q*)) {*p*. *p permutes S*}) (*insert a S*)
⟨*proof*⟩

end

32 Determinants: Traces, Determinant of square matrices and some properties

theory *Determinants*
imports *Euclidean-Space Permutations*
begin

32.1 First some facts about products

lemma *setprod-insert-eq*: *finite A* $\Longrightarrow$ *setprod f* (*insert a A*) = (*if a* $\in$ *A then setprod f A else f a* $*$ *setprod f A*)
⟨*proof*⟩

lemma *setprod-add-split*:
assumes *mn*: (*m*::*nat*) <= *n* + *1*
shows *setprod f* {*m*.. *n*+*p*} = *setprod f* {*m* .. *n*} $*$ *setprod f* {*n*+*1*..*n*+*p*}
⟨*proof*⟩

lemma *setprod-offset*: *setprod f* {(*m*::*nat*) + *p* .. *n* + *p*} = *setprod* (λ*i*. *f* (*i* + *p*)) {*m*..*n*}
⟨*proof*⟩

lemma *setprod-singleton*: *setprod f* {*x*} = *f x* ⟨*proof*⟩

lemma *setprod-singleton-nat-seg*: *setprod f* {*n*..*n*} = *f* (*n*::'*a*::*order*) ⟨*proof*⟩

lemma *setprod-numseg*: *setprod f* {*m*..*0*} = (*if m*=*0 then f 0 else 1*)
setprod f {*m* .. *Suc n*} = (*if m* $\leq$ *Suc n then f* (*Suc n*) $*$ *setprod f* {*m*..*n*}
else setprod f {*m*..*n*})
⟨*proof*⟩

lemma *setprod-le*: **assumes** *fS*: *finite S* **and** *fg*: $\forall$ *x*$\in$*S*. *f x* $\geq$ *0* $\wedge$ *f x* $\leq$ (*g x* :: '*a*::*ordered-idom*)
shows *setprod f S* $\leq$ *setprod g S*
⟨*proof*⟩

lemma *setprod-inversef*: *finite A* ==> *setprod* (*inverse* $\circ$ *f*) *A* = (*inverse* (*setprod f A*) :: '*a*:: {*division-by-zero*, *field*})
⟨*proof*⟩

lemma *setprod-le-1*: **assumes** *fS*: *finite S* **and** *f*: $\forall$ *x*$\in$*S*. *f x* $\geq$ *0* $\wedge$ *f x* $\leq$ (*1*::'*a*::*ordered-idom*)
shows *setprod f S* $\leq$ *1*
⟨*proof*⟩

32.2 Trace

definition *trace* :: *'a::semiring-1^'n^'n ⇒ 'a* **where**
trace A = setsum (λi. ((A$i)$i)) (UNIV::'n set)

lemma *trace-0*: *trace(mat 0) = 0*
⟨*proof*⟩

lemma *trace-I*: *trace(mat 1 :: 'a::semiring-1^'n^'n) = of-nat(CARD('n))*
⟨*proof*⟩

lemma *trace-add*: *trace ((A::'a::comm-semiring-1^'n^'n) + B) = trace A + trace B*
⟨*proof*⟩

lemma *trace-sub*: *trace ((A::'a::comm-ring-1^'n^'n) − B) = trace A − trace B*
⟨*proof*⟩

lemma *trace-mul-sym*:*trace ((A::'a::comm-semiring-1^'n^'n) ** B) = trace (B**A)*
⟨*proof*⟩

definition *det*:: *'a::comm-ring-1^'n^'n ⇒ 'a* **where**
*det A = setsum (λp. of-int (sign p) * setprod (λi. Aip i) (UNIV :: 'n set)) {p. p permutes (UNIV :: 'n set)}*

lemma *setprod-permute*:
assumes *p*: *p permutes S*
shows *setprod f S = setprod (f o p) S*
⟨*proof*⟩

lemma *setproduct-permute-nat-interval*: *p permutes {m::nat .. n} ==> setprod f {m..n} = setprod (f o p) {m..n}*
⟨*proof*⟩

lemma *det-transp*: *det (transp A) = det (A::'a::comm-ring-1 ^'n^'n::finite)*
⟨*proof*⟩

lemma *det-lowerdiagonal*:

fixes $A :: 'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n::\{finite,wellorder\}$
assumes ld: $\bigwedge i\ j.\ i < j \Longrightarrow A\$i\$j = 0$
shows $det\ A = setprod\ (\lambda i.\ A\$i\$i)\ (UNIV :: 'n\ set)$
⟨*proof*⟩

lemma *det-upperdiagonal*:
fixes $A :: 'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n::\{finite,wellorder\}$
assumes ld: $\bigwedge i\ j.\ i > j \Longrightarrow A\$i\$j = 0$
shows $det\ A = setprod\ (\lambda i.\ A\$i\$i)\ (UNIV :: 'n\ set)$
⟨*proof*⟩

lemma *det-diagonal*:
fixes $A :: 'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n::finite$
assumes ld: $\bigwedge i\ j.\ i \neq j \Longrightarrow A\$i\$j = 0$
shows $det\ A = setprod\ (\lambda i.\ A\$i\$i)\ (UNIV::'n\ set)$
⟨*proof*⟩

lemma *det-I*: $det\ (mat\ 1 :: 'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n::finite) = 1$
⟨*proof*⟩

lemma *det-0*: $det\ (mat\ 0 :: 'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n::finite) = 0$
⟨*proof*⟩

lemma *det-permute-rows*:
fixes $A :: 'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n::finite$
assumes p: $p\ permutes\ (UNIV :: 'n::finite\ set)$
shows $det(\chi\ i.\ A\$p\ i :: 'a\hat{}'n\hat{}'n) = of\text{-}int\ (sign\ p) * det\ A$
⟨*proof*⟩

lemma *det-permute-columns*:
fixes $A :: 'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n::finite$
assumes p: $p\ permutes\ (UNIV :: 'n\ set)$
shows $det(\chi\ i\ j.\ A\$i\$\ p\ j :: 'a\hat{}'n\hat{}'n) = of\text{-}int\ (sign\ p) * det\ A$
⟨*proof*⟩

lemma *det-identical-rows*:
fixes $A :: 'a::ordered\text{-}idom\hat{}'n\hat{}'n::finite$
assumes ij: $i \neq j$
and r: $row\ i\ A = row\ j\ A$
shows $det\ A = 0$
⟨*proof*⟩

lemma *det-identical-columns*:
fixes $A :: 'a::ordered\text{-}idom\hat{}'n\hat{}'n::finite$
assumes ij: $i \neq j$
and r: $column\ i\ A = column\ j\ A$
shows $det\ A = 0$
⟨*proof*⟩

lemma *det-zero-row*:
fixes $A :: 'a::\{idom, ring\text{-}char\text{-}0\}\hat{}'n\hat{}'n::finite$
assumes r: $row\ i\ A = 0$
shows $det\ A = 0$
⟨*proof*⟩

lemma *det-zero-column*:
fixes $A :: 'a::\{idom,ring\text{-}char\text{-}0\}\hat{}'n\hat{}'n::finite$
assumes r: $column\ i\ A = 0$
shows $det\ A = 0$
⟨*proof*⟩

lemma *det-row-add*:
fixes $a\ b\ c :: 'n::finite \Rightarrow - \hat{}\ 'n$
shows $det((\chi\ i.\ if\ i = k\ then\ a\ i + b\ i\ else\ c\ i)::'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n) =$
$det((\chi\ i.\ if\ i = k\ then\ a\ i\ else\ c\ i)::'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n) +$
$det((\chi\ i.\ if\ i = k\ then\ b\ i\ else\ c\ i)::'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n)$
⟨*proof*⟩

lemma *det-row-mul*:
fixes $a\ b :: 'n::finite \Rightarrow - \hat{}\ 'n$
shows $det((\chi\ i.\ if\ i = k\ then\ c *s a\ i\ else\ b\ i)::'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n) =$
$c* det((\chi\ i.\ if\ i = k\ then\ a\ i\ else\ b\ i)::'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n)$

⟨*proof*⟩

lemma *det-row-0*:
fixes $b :: 'n::finite \Rightarrow - \hat{}\ 'n$
shows $det((\chi\ i.\ if\ i = k\ then\ 0\ else\ b\ i)::'a::comm\text{-}ring\text{-}1\hat{}'n\hat{}'n) = 0$
⟨*proof*⟩

lemma *det-row-operation*:
fixes $A :: 'a::ordered\text{-}idom\hat{}'n\hat{}'n::finite$
assumes ij: $i \neq j$
shows $det\ (\chi\ k.\ if\ k = i\ then\ row\ i\ A + c *s row\ j\ A\ else\ row\ k\ A) = det\ A$
⟨*proof*⟩

lemma *det-row-span*:
fixes $A :: 'a::\ ordered\text{-}idom\hat{}'n\hat{}'n::finite$
assumes x: $x \in span\ \{row\ j\ A\ |j.\ j \neq i\}$
shows $det\ (\chi\ k.\ if\ k = i\ then\ row\ i\ A + x\ else\ row\ k\ A) = det\ A$
⟨*proof*⟩

lemma *det-dependent-rows*:

fixes *A*:: *'a*::*ordered-idom^'n^'n*::*finite*
assumes *d*: *dependent* (*rows A*)
shows *det A = 0*
⟨*proof*⟩

lemma *det-dependent-columns*: **assumes** *d*: *dependent*(*columns* (*A*::*'a*::*ordered-idom^'n^'n*::*finite*))
shows *det A = 0*
⟨*proof*⟩

lemma *Cart-lambda-cong*: (⋀*x*. *f x = g x*) ⟹ (*Cart-lambda f*::*'a^'n*) = (*Cart-lambda g* :: *'a^'n*)
⟨*proof*⟩

lemma *det-linear-row-setsum*:
assumes *fS*: *finite S*
shows *det* ((χ *i*. *if i = k then setsum* (*a i*) *S else c i*)::*'a*::*comm-ring-1^'n^'n*::*finite*)
= *setsum* (λ*j*. *det* ((χ *i*. *if i = k then a i j else c i*)::*'a^'n^'n*)) *S*
⟨*proof*⟩

lemma *finite-bounded-functions*:
assumes *fS*: *finite S*
shows *finite* {*f*. (∀ *i* ∈ {*1*.. (*k*::*nat*)}. *f i* ∈ *S*) ∧ (∀ *i*. *i* ∉ {*1* .. *k*} ⟶ *f i = i*)}
⟨*proof*⟩

lemma *eq-id-iff*[*simp*]: (∀ *x*. *f x = x*) = (*f = id*) ⟨*proof*⟩

lemma *det-linear-rows-setsum-lemma*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
shows *det*((χ *i*. *if i* ∈ *T then setsum* (*a i*) *S else c i*):: *'a*::*comm-ring-1^'n^'n*::*finite*)
=
setsum (λ*f*. *det*((χ *i*. *if i* ∈ *T then a i* (*f i*) *else c i*)::*'a^'n^'n*))
{*f*. (∀ *i* ∈ *T*. *f i* ∈ *S*) ∧ (∀ *i*. *i* ∉ *T* ⟶ *f i = i*)}
⟨*proof*⟩

lemma *det-linear-rows-setsum*:
assumes *fS*: *finite* (*S*::*'n*::*finite set*)
shows *det* (χ *i*. *setsum* (*a i*) *S*) = *setsum* (λ*f*. *det* (χ *i*. *a i* (*f i*) :: *'a*::*comm-ring-1*
^ *'n^'n*::*finite*)) {*f*. ∀ *i*. *f i* ∈ *S*}
⟨*proof*⟩

lemma *matrix-mul-setsum-alt*:
fixes *A B* :: *'a*::*comm-ring-1^'n^'n*::*finite*
shows *A* ** *B* = (χ *i*. *setsum* (λ*k*. *Aik* *s *B* $ *k*) (*UNIV* :: *'n set*))
⟨*proof*⟩

lemma *det-rows-mul*:
det((χ *i*. *c i* ∗*s a i*)::*'a*::*comm-ring-1^'n^'n*::*finite*) =
setprod (λ*i*. *c i*) (*UNIV*:: *'n set*) ∗ *det*((χ *i*. *a i*)::*'a^'n^'n*)
⟨*proof*⟩

lemma *det-mul*:
fixes *A B* :: *'a*::*ordered-idom^'n^'n*::*finite*
shows *det* (*A* ∗∗ *B*) = *det A* ∗ *det B*
⟨*proof*⟩

lemma *invertible-left-inverse*:
fixes *A* :: *real^'n^'n*::*finite*
shows *invertible A* ⟷ (∃(*B*::*real^'n^'n*). *B*∗∗ *A* = *mat 1*)
⟨*proof*⟩

lemma *invertible-righ-inverse*:
fixes *A* :: *real^'n^'n*::*finite*
shows *invertible A* ⟷ (∃(*B*::*real^'n^'n*). *A*∗∗ *B* = *mat 1*)
⟨*proof*⟩

lemma *invertible-det-nz*:
fixes *A*::*real ^'n^'n*::*finite*
shows *invertible A* ⟷ *det A* ≠ *0*
⟨*proof*⟩

lemma *cramer-lemma-transp*:
fixes *A*:: *'a*::*ordered-idom^'n^'n*::*finite* **and** *x* :: *'a ^'n*::*finite*
shows *det* ((χ *i*. *if i* = *k then setsum* (λ*i*. *x*$*i* ∗*s row i A*) (*UNIV*::*'n set*)
else row i A)::*'a^'n^'n*) = *x*$*k* ∗ *det A*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *cramer-lemma*:
fixes *A* :: *'a*::*ordered-idom ^'n^'n*::*finite*
shows *det*((χ *i j*. *if j* = *k then* (*A* ∗*v x*)$*i else A*$*i*$*j*):: *'a^'n^'n*) = *x*$*k* ∗ *det A*
⟨*proof*⟩

lemma *cramer*:
fixes *A* ::*real^'n^'n*::*finite*
assumes *d0*: *det A* ≠ *0*

shows $A *v x = b \longleftrightarrow x = (\chi\ k.\ det(\chi\ i\ j.\ if\ j{=}k\ then\ b\$i\ else\ A\$i\$j :: real\hat{}'n\hat{}'n)\ /\ det\ A)$
⟨*proof*⟩

definition *orthogonal-transformation f* $\longleftrightarrow$ *linear f* $\wedge$ $(\forall v\ w.\ f\ v \cdot f\ w = v \cdot w)$

lemma *orthogonal-transformation*: *orthogonal-transformation f* $\longleftrightarrow$ *linear f* $\wedge$ $(\forall (v{::}real\ \hat{}\text{-}).\ norm\ (f\ v) = norm\ v)$
⟨*proof*⟩

definition *orthogonal-matrix* $(Q{::}'a{::}semiring\text{-}1\hat{}'n\hat{}'n) \longleftrightarrow transp\ Q ** Q = mat\ 1 \wedge Q ** transp\ Q = mat\ 1$

lemma *orthogonal-matrix*: *orthogonal-matrix* $(Q{::}\ real\ \hat{}'n\hat{}'n{::}finite) \longleftrightarrow transp\ Q ** Q = mat\ 1$
⟨*proof*⟩

lemma *orthogonal-matrix-id*: *orthogonal-matrix* $(mat\ 1 :: -\hat{}'n\hat{}'n{::}finite)$
⟨*proof*⟩

lemma *orthogonal-matrix-mul*:
fixes $A :: real\ \hat{}'n\hat{}'n{::}finite$
assumes *oA* : *orthogonal-matrix A*
and *oB*: *orthogonal-matrix B*
shows *orthogonal-matrix*$(A ** B)$
⟨*proof*⟩

lemma *orthogonal-transformation-matrix*:
fixes $f{::}\ real\hat{}'n \Rightarrow real\hat{}'n{::}finite$
shows *orthogonal-transformation f* $\longleftrightarrow$ *linear f* $\wedge$ *orthogonal-matrix*$(matrix\ f)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *det-orthogonal-matrix*:
fixes $Q{::}\ 'a{::}ordered\text{-}idom\hat{}'n\hat{}'n{::}finite$
assumes *oQ*: *orthogonal-matrix Q*
shows *det Q* $= 1 \vee$ *det Q* $= -1$
⟨*proof*⟩

lemma *scaling-linear*:
fixes $f :: real\ \hat{}'n \Rightarrow real\ \hat{}'n{::}finite$
assumes *f0*: *f 0* $= 0$ **and** *fd*: $\forall x\ y.\ dist\ (f\ x)\ (f\ y) = c * dist\ x\ y$

shows *linear f*
⟨*proof*⟩

lemma *isometry-linear*:
f (*0*:: *real^'n*) = (*0*:: *real^'n*::*finite*) ⟹ ∀ *x y*. *dist*(*f x*) (*f y*) = *dist x y*
⟹ *linear f*
⟨*proof*⟩

lemma *orthogonal-transformation-isometry*:
orthogonal-transformation f ⟷ *f*(*0*::*real^'n*) = (*0*::*real^'n*::*finite*) ∧ (∀ *x y*. *dist*(*f x*) (*f y*) = *dist x y*)
⟨*proof*⟩

lemma *isometry-sphere-extend*:
fixes *f*:: *real ^'n* ⇒ *real ^'n*::*finite*
assumes *f1*: ∀ *x*. *norm x* = *1* ⟶ *norm* (*f x*) = *1*
and *fd1*: ∀ *x y*. *norm x* = *1* ⟶ *norm y* = *1* ⟶ *dist* (*f x*) (*f y*) = *dist x y*
shows ∃ *g*. *orthogonal-transformation g* ∧ (∀ *x*. *norm x* = *1* ⟶ *g x* = *f x*)
⟨*proof*⟩

definition *rotation-matrix Q* ⟷ *orthogonal-matrix Q* ∧ *det Q* = *1*
definition *rotoinversion-matrix Q* ⟷ *orthogonal-matrix Q* ∧ *det Q* = − *1*

lemma *orthogonal-rotation-or-rotoinversion*:
fixes *Q* :: *'a*::*ordered-idom^'n^'n*::*finite*
shows *orthogonal-matrix Q* ⟷ *rotation-matrix Q* ∨ *rotoinversion-matrix Q*
⟨*proof*⟩

lemma *setprod-1*: *setprod f* {(*1*::*nat*)..*1*} = *f 1* ⟨*proof*⟩

lemma *setprod-2*: *setprod f* {(*1*::*nat*)..*2*} = *f 1* ∗ *f 2*
⟨*proof*⟩
lemma *setprod-3*: *setprod f* {(*1*::*nat*)..*3*} = *f 1* ∗ *f 2* ∗ *f 3*
⟨*proof*⟩

lemma *det-1*: *det* (*A*::'*a*::*comm-ring-1*^*1*^*1*) = *A*$*1*$*1*
⟨*proof*⟩

lemma *det-2*: *det* (*A*::'*a*::*comm-ring-1*^*2*^*2*) = *A*$*1*$*1* * *A*$*2*$*2* − *A*$*1*$*2* * *A*$*2*$*1*
⟨*proof*⟩

lemma *det-3*: *det* (*A*::'*a*::*comm-ring-1*^*3*^*3*) =
A$*1*$*1* * *A*$*2*$*2* * *A*$*3*$*3* +
A$*1*$*2* * *A*$*2*$*3* * *A*$*3*$*1* +
A$*1*$*3* * *A*$*2*$*1* * *A*$*3*$*2* −
A$*1*$*1* * *A*$*2*$*3* * *A*$*3*$*2* −
A$*1*$*2* * *A*$*2*$*1* * *A*$*3*$*3* −
A$*1*$*3* * *A*$*2*$*2* * *A*$*3*$*1*
⟨*proof*⟩

end

33 Diagonalize: A constructive version of Cantor's first diagonalization argument.

theory *Diagonalize*
imports *Main*
begin

33.1 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*:

$n \leq sum\ n$
⟨*proof*⟩

lemma *sum-rest-aux*:
assumes $q \leq n$
assumes $sum\ m \leq sum\ n + q$
shows $m \leq n$
⟨*proof*⟩

lemma *sum-rest*:
assumes $q \leq n$
shows $sum\ m \leq sum\ n + q \longleftrightarrow m \leq n$
⟨*proof*⟩

33.2 Diagonalization: an injective embedding of two *nat*s to one *nat*

definition *diagonalize* :: $nat \Rightarrow nat \Rightarrow nat$ **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*⟩

33.3 The reverse diagonalization: reconstruction a pair of *nat*s from one *nat*

The inverse of the *sum* function

definition *tupelize* :: $nat \Rightarrow nat \times nat$ **where**
$tupelize\ q = (let\ d = Max\ \{d.\ sum\ d \leq q\};\ m = q - sum\ d$
$in\ (m,\ d - m))$

lemma *tupelize-diagonalize*:
$tupelize\ (diagonalize\ m\ n) = (m,\ n)$
⟨*proof*⟩

lemma *snd-tupelize*:
$snd\ (tupelize\ n) \leq n$
⟨*proof*⟩

end

34 Efficient-Nat: Implementation of natural numbers by target-language integers

theory *Efficient-Nat*

imports *Code-Index 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.

34.1 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 inline*]:
0 = (*Numeral0* :: *nat*)
⟨*proof*⟩
lemmas [*code post*] = *zero-nat-code* [*symmetric*]

lemma *one-nat-code* [*code*, *code inline*]:
1 = (*Numeral1* :: *nat*)
⟨*proof*⟩
lemmas [*code post*] = *one-nat-code* [*symmetric*]

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* = *Divides.divmod*

lemma [*code*]:
Divides.divmod 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*⟩

34.2 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*⟩

34.3 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 = (*if n* = *0 then g else h* (*n* − *1*))
⟨*proof*⟩

lemma *Suc-if-eq*: (⋀*n*. *f* (*Suc n*) ≡ *h n*) ⟹ *f 0* ≡ *g* ⟹
f n ≡ *if n* = *0 then g else h* (*n* − *1*)
⟨*proof*⟩

lemma *Suc-clause*: (⋀*n*. *P n* (*Suc n*)) ⟹ *n* ≠ *0* ⟹ *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*⟩

34.4 Target language setup

For ML, we map *nat* to target language integers, where we assert that values are always non-negative.

```
code-type nat
  (SML 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 we define our own *nat* type. The reason is that we have to distinguish type class instances for *nat* and *int*.

```
code-include Haskell Nat ⟪
newtype Nat = Nat Integer deriving (Show, Eq);

instance Num Nat where {
  fromInteger k = Nat (if k >= 0 then k else 0);
  Nat n + Nat m = Nat (n + m);
  Nat n - Nat m = fromInteger (n - m);
  Nat n * Nat m = Nat (n * m);
  abs n = n;
  signum - = 1;
  negate n = error negate Nat;
};

instance Ord Nat where {
  Nat n <= Nat m = n <= m;
  Nat n < Nat m = n < m;
};

instance Real Nat where {
  toRational (Nat n) = toRational n;
};
```

```
instance Enum Nat where {
  toEnum k = fromInteger (toEnum k);
  fromEnum (Nat n) = fromEnum n;
};

instance Integral Nat where {
  toInteger (Nat n) = n;
  divMod n m = quotRem n m;
  quotRem (Nat n) (Nat m) = (Nat k, Nat l) where (k, l) = quotRem n m;
};
⟫
```

code-reserved *Haskell Nat*

code-type *nat*
(*Haskell Nat.Nat*)

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

Natural numerals.

lemma [*code inline*, *symmetric*, *code post*]:
nat (*number-of i*) = *number-nat-inst.number-of-nat i*
— this interacts as desired with *number-of ?v* = *nat* (*number-of ?v*)
⟨*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*]:
of-nat = *int* ⟨*proof*⟩
declare *of-nat-int* [*symmetric*, *code post*]

code-const *int*
(*SML* -)
(*OCaml* -)

consts-code
int ((-))
nat (⟨**module**⟩*nat*)
attach ⟪
fun nat i = if i < 0 then 0 else i;
⟫

code-const *nat*
(*SML IntInf.max*/ (/*0*,/ -))
(*OCaml Big'-int.max'-big'-int*/ *Big'-int.zero'-big'-int*)

For Haskell, things are slightly different again.

code-const *int* **and** *nat*
(*Haskell toInteger* **and** *fromInteger*)

Conversion from and to indices.

code-const *Code-Index.of-nat*
(*SML IntInf.toInt*)
(*OCaml Big'-int.int'-of'-big'-int*)
(*Haskell fromEnum*)

code-const *Code-Index.nat-of*
(*SML IntInf.fromInt*)
(*OCaml Big'-int.big'-int'-of'-int*)
(*Haskell toEnum*)

Using target language arithmetic operations whenever appropriate

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

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

code-const *divmod-aux*
(*SML IntInf.divMod*/ ((-),/ (-)))
(*OCaml Big'-int.quomod'-big'-int*)
(*Haskell divMod*)

code-const *eq-class.eq* :: *nat* ⇒ *nat* ⇒ *bool*
(*SML* !((- : *IntInf.int*) = -))

(*OCaml Big'-int.eq'-big'-int*)
(*Haskell* **infixl** *4* ==)

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

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

consts-code
0::*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-Eval.term-of* :: *nat* ⇒ *term*) = *Code-Eval.term-of* ⟨*proof*⟩

code-const *Code-Eval.term-of* :: *nat* ⇒ *term*
(*SML HOLogic.mk'-number*/ *HOLogic.natT*)

Module names

code-modulename *SML*
Nat Integer
Divides Integer
Ring-and-Field Integer
Efficient-Nat Integer

code-modulename *OCaml*
Nat Integer
Divides Integer
Ring-and-Field Integer
Efficient-Nat Integer

code-modulename *Haskell*
Nat Integer
Divides Integer
Ring-and-Field Integer
Efficient-Nat Integer

hide *const int*

end

35 Enum: Finite types as explicit enumerations

theory *Enum*
imports *Map Main*
begin

35.1 Class *enum*

class *enum* =
  fixes *enum* :: *'a list*
  assumes *UNIV-enum* [*code*]: *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

35.2 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*⟩

35.3 Quantifiers

lemma *all-code* [*code*]: $(\forall x.\ P\ x) \longleftrightarrow$ *list-all P enum*
⟨*proof*⟩

lemma *exists-code* [*code*]: $(\exists x.\ P\ x) \longleftrightarrow \neg$ *list-all* (*Not o P*) *enum*
⟨*proof*⟩

35.4 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-map*:
fixes *f* :: *'a*::*enum* ⇒ *'b*::*enum*
shows *map-of* (*zip xs* (*map f xs*)) = (λ*x*. *if x* ∈ *set xs then Some* (*f x*) *else None*)
⟨*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-char* [*code*]:
enum = [*Char Nibble0 Nibble0*, *Char Nibble0 Nibble1*, *Char Nibble0 Nibble2*,
Char Nibble0 Nibble3, *Char Nibble0 Nibble4*, *Char Nibble0 Nibble5*,
Char Nibble0 Nibble6, *Char Nibble0 Nibble7*, *Char Nibble0 Nibble8*,
Char Nibble0 Nibble9, *Char Nibble0 NibbleA*, *Char Nibble0 NibbleB*,
Char Nibble0 NibbleC, *Char Nibble0 NibbleD*, *Char Nibble0 NibbleE*,
Char Nibble0 NibbleF, *Char Nibble1 Nibble0*, *Char Nibble1 Nibble1*,
Char Nibble1 Nibble2, *Char Nibble1 Nibble3*, *Char Nibble1 Nibble4*,
Char Nibble1 Nibble5, *Char Nibble1 Nibble6*, *Char Nibble1 Nibble7*,
Char Nibble1 Nibble8, *Char Nibble1 Nibble9*, *Char Nibble1 NibbleA*,
Char Nibble1 NibbleB, *Char Nibble1 NibbleC*, *Char Nibble1 NibbleD*,
Char Nibble1 NibbleE, *Char Nibble1 NibbleF*, *CHR '' ''*, *CHR ''!''*,
Char Nibble2 Nibble2, *CHR ''#''*, *CHR ''$''*, *CHR ''%''*, *CHR ''&''*,
Char Nibble2 Nibble7, *CHR ''('*, *CHR '')''*, *CHR ''*''*, *CHR ''+''*, *CHR '',''*,
CHR ''−'', *CHR ''.''*, *CHR ''/''*, *CHR ''0''*, *CHR ''1''*, *CHR ''2''*, *CHR ''3''*,
CHR ''4'', *CHR ''5''*, *CHR ''6''*, *CHR ''7''*, *CHR ''8''*, *CHR ''9''*, *CHR '':''*,
CHR '';'', *CHR ''<''*, *CHR ''=''*, *CHR ''>''*, *CHR ''?''*, *CHR ''@''*, *CHR ''A''*,
CHR ''B'', *CHR ''C''*, *CHR ''D''*, *CHR ''E''*, *CHR ''F''*, *CHR ''G''*, *CHR ''H''*,
CHR ''I'', *CHR ''J''*, *CHR ''K''*, *CHR ''L''*, *CHR ''M''*, *CHR ''N''*, *CHR ''O''*,
CHR ''P'', *CHR ''Q''*, *CHR ''R''*, *CHR ''S''*, *CHR ''T''*, *CHR ''U''*, *CHR ''V''*,
CHR ''W'', *CHR ''X''*, *CHR ''Y''*, *CHR ''Z''*, *CHR ''[''*, *Char Nibble5 NibbleC*,
CHR '']'', *CHR ''^''*, *CHR ''-''*, *Char Nibble6 Nibble0*, *CHR ''a''*, *CHR ''b''*,
CHR ''c'', *CHR ''d''*, *CHR ''e''*, *CHR ''f''*, *CHR ''g''*, *CHR ''h''*, *CHR ''i''*,
CHR ''j'', *CHR ''k''*, *CHR ''l''*, *CHR ''m''*, *CHR ''n''*, *CHR ''o''*, *CHR ''p''*,
CHR ''q'', *CHR ''r''*, *CHR ''s''*, *CHR ''t''*, *CHR ''u''*, *CHR ''v''*, *CHR ''w''*,
CHR ''x'', *CHR ''y''*, *CHR ''z''*, *CHR ''{''*, *CHR ''|''*, *CHR ''}''*, *CHR ''~''*,
Char Nibble7 NibbleF, *Char Nibble8 Nibble0*, *Char Nibble8 Nibble1*,
Char Nibble8 Nibble2, *Char Nibble8 Nibble3*, *Char Nibble8 Nibble4*,
Char Nibble8 Nibble5, *Char Nibble8 Nibble6*, *Char Nibble8 Nibble7*,
Char Nibble8 Nibble8, *Char Nibble8 Nibble9*, *Char Nibble8 NibbleA*,
Char Nibble8 NibbleB, *Char Nibble8 NibbleC*, *Char Nibble8 NibbleD*,
Char Nibble8 NibbleE, *Char Nibble8 NibbleF*, *Char Nibble9 Nibble0*,
Char Nibble9 Nibble1, *Char Nibble9 Nibble2*, *Char Nibble9 Nibble3*,
Char Nibble9 Nibble4, *Char Nibble9 Nibble5*, *Char Nibble9 Nibble6*,
Char Nibble9 Nibble7, *Char Nibble9 Nibble8*, *Char Nibble9 Nibble9*,
Char Nibble9 NibbleA, *Char Nibble9 NibbleB*, *Char Nibble9 NibbleC*,
Char Nibble9 NibbleD, *Char Nibble9 NibbleE*, *Char Nibble9 NibbleF*,
Char NibbleA Nibble0, *Char NibbleA Nibble1*, *Char NibbleA Nibble2*,
Char NibbleA Nibble3, *Char NibbleA Nibble4*, *Char NibbleA Nibble5*,
Char NibbleA Nibble6, *Char NibbleA Nibble7*, *Char NibbleA Nibble8*,
Char NibbleA Nibble9, *Char NibbleA NibbleA*, *Char NibbleA NibbleB*,
Char NibbleA NibbleC, *Char NibbleA NibbleD*, *Char NibbleA NibbleE*,
Char NibbleA NibbleF, *Char NibbleB Nibble0*, *Char NibbleB Nibble1*,
Char NibbleB Nibble2, *Char NibbleB Nibble3*, *Char NibbleB Nibble4*,

Char NibbleB Nibble5, *Char NibbleB Nibble6*, *Char NibbleB Nibble7*,
Char NibbleB Nibble8, *Char NibbleB Nibble9*, *Char NibbleB NibbleA*,
Char NibbleB NibbleB, *Char NibbleB NibbleC*, *Char NibbleB NibbleD*,
Char NibbleB NibbleE, *Char NibbleB NibbleF*, *Char NibbleC Nibble0*,
Char NibbleC Nibble1, *Char NibbleC Nibble2*, *Char NibbleC Nibble3*,
Char NibbleC Nibble4, *Char NibbleC Nibble5*, *Char NibbleC Nibble6*,
Char NibbleC Nibble7, *Char NibbleC Nibble8*, *Char NibbleC Nibble9*,
Char NibbleC NibbleA, *Char NibbleC NibbleB*, *Char NibbleC NibbleC*,
Char NibbleC NibbleD, *Char NibbleC NibbleE*, *Char NibbleC NibbleF*,
Char NibbleD Nibble0, *Char NibbleD Nibble1*, *Char NibbleD Nibble2*,
Char NibbleD Nibble3, *Char NibbleD Nibble4*, *Char NibbleD Nibble5*,
Char NibbleD Nibble6, *Char NibbleD Nibble7*, *Char NibbleD Nibble8*,
Char NibbleD Nibble9, *Char NibbleD NibbleA*, *Char NibbleD NibbleB*,
Char NibbleD NibbleC, *Char NibbleD NibbleD*, *Char NibbleD NibbleE*,
Char NibbleD NibbleF, *Char NibbleE Nibble0*, *Char NibbleE Nibble1*,
Char NibbleE Nibble2, *Char NibbleE Nibble3*, *Char NibbleE Nibble4*,
Char NibbleE Nibble5, *Char NibbleE Nibble6*, *Char NibbleE Nibble7*,
Char NibbleE Nibble8, *Char NibbleE Nibble9*, *Char NibbleE NibbleA*,
Char NibbleE NibbleB, *Char NibbleE NibbleC*, *Char NibbleE NibbleD*,
Char NibbleE NibbleE, *Char NibbleE NibbleF*, *Char NibbleF Nibble0*,
Char NibbleF Nibble1, *Char NibbleF Nibble2*, *Char NibbleF Nibble3*,
Char NibbleF Nibble4, *Char NibbleF Nibble5*, *Char NibbleF Nibble6*,
Char NibbleF Nibble7, *Char NibbleF Nibble8*, *Char NibbleF Nibble9*,
Char NibbleF NibbleA, *Char NibbleF NibbleB*, *Char NibbleF NibbleC*,
Char NibbleF NibbleD, *Char NibbleF NibbleE*, *Char NibbleF NibbleF*]
⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *option* :: (*enum*) *enum*
begin

definition
enum = *None* # *map Some enum*

instance ⟨*proof*⟩

end

end

36 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*⟩

36.1 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*⟩

36.2 Discussion

36.2.1 Conflicts

This theory conflicts with EfficientNat, since the *ml-equiv* instance for natural numbers is not valid when they are mapped to ML integers. With that

theory loaded, we could use our oracle to prove $\exists\, n.\ n < (0::'a)$ by providing $^{\sim}1$ as a witness.

This shows that *ml-equiv* declarations have to be used with care, taking the configuration of the code generator into account.

36.2.2 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

37 Executable-Set: Implementation of finite sets by lists

theory *Executable-Set*
imports *Main*
begin

37.1 Definitional rewrites

definition *subset* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
subset = *op* ≤

declare *subset-def* [*symmetric*, *code unfold*]

lemma [*code*]: *subset A B* ⟷ (∀ *x*∈*A*. *x* ∈ *B*)
⟨*proof*⟩

definition *is-empty* :: *'a set* ⇒ *bool* **where**
is-empty A ⟷ *A* = {}

definition *eq-set* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
[*code del*]: *eq-set* = *op* =

lemma [*code*]: *eq-set A B* ⟷ *A* ⊆ *B* ∧ *B* ⊆ *A*
⟨*proof*⟩

lemma [*code*]:
a ∈ *A* ⟷ (∃ *x*∈*A*. *x* = *a*)

⟨*proof*⟩

definition *filter-set* :: (′*a* ⇒ *bool*) ⇒ ′*a set* ⇒ ′*a set* **where**
filter-set P xs = {*x*∈*xs*. *P x*}

declare *filter-set-def*[*symmetric*, *code unfold*]

37.2 Operations on lists

37.2.1 Basic definitions

definition
flip :: (′*a* ⇒ ′*b* ⇒ ′*c*) ⇒ ′*b* ⇒ ′*a* ⇒ ′*c* **where**
flip f a b = *f b a*

definition
member :: ′*a list* ⇒ ′*a* ⇒ *bool* **where**
member xs x ⟷ *x* ∈ *set xs*

definition
insertl :: ′*a* ⇒ ′*a list* ⇒ ′*a list* **where**
insertl x xs = (*if member xs x then xs else x*#*xs*)

lemma [*code target*: *List*]: *member* [] *y* ⟷ *False*
and [*code target*: *List*]: *member* (*x*#*xs*) *y* ⟷ *y* = *x* ∨ *member xs y*
⟨*proof*⟩

fun
drop-first :: (′*a* ⇒ *bool*) ⇒ ′*a list* ⇒ ′*a list* **where**
drop-first f [] = []
| *drop-first f* (*x*#*xs*) = (*if f x then xs else x* # *drop-first f xs*)
declare *drop-first.simps* [*code del*]
declare *drop-first.simps* [*code target*: *List*]

declare *remove1.simps* [*code del*]
lemma [*code target*: *List*]:
remove1 x xs = (*if member xs x then drop-first* (λ*y*. *y* = *x*) *xs else xs*)
⟨*proof*⟩

lemma *member-nil* [*simp*]:
member [] = (λ*x*. *False*)
⟨*proof*⟩

lemma *member-insertl* [*simp*]:
x ∈ *set* (*insertl x xs*)
⟨*proof*⟩

lemma *insertl-member* [*simp*]:
fixes *xs x*
assumes *member*: *member xs x*

shows *insertl x xs = xs*
⟨*proof*⟩

lemma *insertl-not-member* [*simp*]:
fixes *xs x*
assumes *member*: ¬ (*member xs x*)
shows *insertl x xs = x # xs*
⟨*proof*⟩

lemma *foldr-remove1-empty* [*simp*]:
foldr remove1 xs [] = []
⟨*proof*⟩

37.2.2 Derived definitions

function *unionl* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
unionl [] *ys* = *ys*
| *unionl xs ys* = *foldr insertl xs ys*
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *unionl-eq* = *unionl.simps*(*2*)

function *intersect* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
intersect [] *ys* = []
| *intersect xs* [] = []
| *intersect xs ys* = *filter* (*member xs*) *ys*
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *intersect-eq* = *intersect.simps*(*3*)

function *subtract* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
subtract [] *ys* = *ys*
| *subtract xs* [] = []
| *subtract xs ys* = *foldr remove1 xs ys*
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *subtract-eq* = *subtract.simps*(*3*)

function *map-distinct* :: (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list*
where
map-distinct f [] = []
| *map-distinct f xs* = *foldr* (*insertl o f*) *xs* []
⟨*proof*⟩

termination ⟨*proof*⟩

lemmas *map-distinct-eq* = *map-distinct.simps*(*2*)

function *unions* :: *'a list list* ⇒ *'a list*
where
  unions [] = []
| *unions xs* = *foldr unionl xs* []
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *unions-eq* = *unions.simps*(*2*)

consts *intersects* :: *'a list list* ⇒ *'a list*
primrec
  intersects (*x*#*xs*) = *foldr intersect xs x*

definition
  map-union :: *'a list* ⇒ (*'a* ⇒ *'b list*) ⇒ *'b list* **where**
  map-union xs f = *unions* (*map f xs*)

definition
  map-inter :: *'a list* ⇒ (*'a* ⇒ *'b list*) ⇒ *'b list* **where**
  map-inter xs f = *intersects* (*map f xs*)

37.3 Isomorphism proofs

lemma *iso-member*:
  member xs x ⟷ *x* ∈ *set xs*
  ⟨*proof*⟩

lemma *iso-insert*:
  set (*insertl x xs*) = *insert x* (*set xs*)
  ⟨*proof*⟩

lemma *iso-remove1*:
  assumes *distnct*: *distinct xs*
  shows *set* (*remove1 x xs*) = *set xs* − {*x*}
  ⟨*proof*⟩

lemma *iso-union*:
  set (*unionl xs ys*) = *set xs* ∪ *set ys*
  ⟨*proof*⟩

lemma *iso-intersect*:
  set (*intersect xs ys*) = *set xs* ∩ *set ys*
  ⟨*proof*⟩

definition

subtract' :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
subtract' = *flip subtract*

lemma *iso-subtract*:
fixes *ys*
assumes *distnct*: *distinct ys*
shows *set* (*subtract' ys xs*) = *set ys* − *set xs*
and *distinct* (*subtract' ys xs*)
⟨*proof*⟩

lemma *iso-map-distinct*:
set (*map-distinct f xs*) = *image f* (*set xs*)
⟨*proof*⟩

lemma *iso-unions*:
set (*unions xss*) = ⋃ *set* (*map set xss*)
⟨*proof*⟩

lemma *iso-intersects*:
set (*intersects* (*xs*#*xss*)) = ⋂ *set* (*map set* (*xs*#*xss*))
⟨*proof*⟩

lemma *iso-UNION*:
set (*map-union xs f*) = *UNION* (*set xs*) (*set o f*)
⟨*proof*⟩

lemma *iso-INTER*:
set (*map-inter* (*x*#*xs*) *f*) = *INTER* (*set* (*x*#*xs*)) (*set o f*)
⟨*proof*⟩

definition
Blall :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Blall = *flip list-all*
definition
Blex :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Blex = *flip list-ex*

lemma *iso-Ball*:
Blall xs f = *Ball* (*set xs*) *f*
⟨*proof*⟩

lemma *iso-Bex*:
Blex xs f = *Bex* (*set xs*) *f*
⟨*proof*⟩

lemma *iso-filter*:
set (*filter P xs*) = *filter-set P* (*set xs*)
⟨*proof*⟩

37.4 code generator setup

⟨*ML*⟩

37.4.1 const serializations

consts-code
Set.empty ({∗[]∗})
insert ({∗*insertl*∗})
is-empty ({∗*null*∗})
op ∪ ({∗*unionl*∗})
op ∩ ({∗*intersect*∗})
op − :: *'a set* ⇒ *'a set* ⇒ *'a set* ({∗ *flip subtract* ∗})
image ({∗*map-distinct*∗})
Union ({∗*unions*∗})
Inter ({∗*intersects*∗})
UNION ({∗*map-union*∗})
INTER ({∗*map-inter*∗})
Ball ({∗*Blall*∗})
Bex ({∗*Blex*∗})
filter-set ({∗*filter*∗})
fold ({∗ *foldl o flip* ∗})

end

38 Float: Floating-Point Numbers

theory *Float*
imports *Complex-Main*
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*

fun *Ifloat* :: *float* ⇒ *real* **where**
Ifloat (*Float a b*) = *real a* ∗ *pow2 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

fun *mantissa* :: *float* ⇒ *int* **where**
mantissa (Float a b) = a

fun *scale* :: *float* ⇒ *int* **where**
scale (Float a b) = b

lemma *Ifloat-neg-exp*: *e < 0 ⟹ Ifloat (Float m e) = real m * inverse (2^nat (−e))* ⟨*proof*⟩
lemma *Ifloat-nge0-exp*: *¬ 0 ≤ e ⟹ Ifloat (Float m e) = real m * inverse (2^nat (−e))* ⟨*proof*⟩
lemma *Ifloat-ge0-exp*: *0 ≤ e ⟹ Ifloat (Float m e) = real m * (2^nat e)* ⟨*proof*⟩

lemma *Float-num*[*simp*]: **shows**
Ifloat (Float 1 0) = 1 **and** *Ifloat (Float 1 1) = 2* **and** *Ifloat (Float 1 2) = 4* **and**
Ifloat (Float 1 −1) = 1/2 **and** *Ifloat (Float 1 −2) = 1/4* **and** *Ifloat (Float 1 −3) = 1/8* **and**
Ifloat (Float −1 0) = −1 **and** *Ifloat (Float (number-of n) 0) = number-of 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*]: *Ifloat (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*]: *Ifloat f* = *Ifloat* (*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*: *Ifloat* (*Float a b*) = *Ifloat* (*Float a*′ *b*′)
shows *b* ≤ *b*′
⟨*proof*⟩

lemma *float-eq-odd*:
assumes *odd1*: *odd a*
and *odd2*: *odd a*′
and *floateq*: *Ifloat* (*Float a b*) = *Ifloat* (*Float a*′ *b*′)
shows *a* = *a*′ ∧ *b* = *b*′
⟨*proof*⟩

theorem *normfloat-unique*:
assumes *Ifloat-eq*: *Ifloat f* = *Ifloat 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**
fun *uminus-float* **where** [*simp del*]: *uminus-float* (*Float m e*) = *Float* (−*m*) *e*
instance ⟨*proof*⟩
end

instantiation *float* :: *minus* **begin**
fun *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

fun *float-pprt* :: *float* ⇒ *float* **where**
float-pprt (*Float a e*) = (*if 0* <= *a then* (*Float a e*) *else 0*)

fun *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* ≤ *w* ≡ *Ifloat z* ≤ *Ifloat w*
definition *less-float-def*: *z* < *w* ≡ *Ifloat z* < *Ifloat w*
instance ⟨*proof*⟩
end

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

lemma *Ifloat-minus*[*simp*]: *Ifloat* (− *a*) = − *Ifloat a*
⟨*proof*⟩

lemma *Ifloat-sub*[*simp*]: *Ifloat* (*a* − *b*) = *Ifloat a* − *Ifloat b*
⟨*proof*⟩

lemma *Ifloat-mult*[*simp*]: *Ifloat* (*a*∗*b*) = *Ifloat a* ∗ *Ifloat b*
⟨*proof*⟩

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

lemma *Ifloat-1*[*simp*]: *Ifloat 1* = *1*
⟨*proof*⟩

lemma *zero-le-float*:

$(0 <= Ifloat\ (Float\ a\ b)) = (0 <= a)$
⟨*proof*⟩

lemma *float-le-zero*:
$(Ifloat\ (Float\ a\ b) <= 0) = (a <= 0)$
⟨*proof*⟩

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

lemma *Ifloat-pprt*[*simp*]: $Ifloat\ (float\text{-}pprt\ a) = pprt\ (Ifloat\ a)$
⟨*proof*⟩

lemma *Ifloat-nprt*[*simp*]: $Ifloat\ (float\text{-}nprt\ a) = nprt\ (Ifloat\ a)$
⟨*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 *Ifloat-min*: $Ifloat\ (min\ x\ y) = min\ (Ifloat\ x)\ (Ifloat\ y)$ ⟨*proof*⟩
lemma *Ifloat-max*: $Ifloat\ (max\ a\ b) = max\ (Ifloat\ a)\ (Ifloat\ b)$ ⟨*proof*⟩

instantiation *float* :: *power* **begin**
fun *power-float* **where** [*simp del*]: $(Float\ m\ e)\ \hat{}\ n = Float\ (m\ \hat{}\ n)\ (e * int\ n)$
instance ⟨*proof*⟩
end

instance *float* :: *recpower*
⟨*proof*⟩

lemma *float-power*: *Ifloat* $(x \hat{} n) = (Ifloat\ x) \hat{} n$
⟨*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** *Ifloat* (*Float m* (− (*bitlen m* − *1*))) = *real m* / *2^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-of w*) = *number-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 *Ifloat* (*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* $(x\ div\ y) \leq$ *Ifloat* $($*lapprox-posrat* $n\ x\ y)$
⟨*proof*⟩

lemma *lapprox-posrat-nonneg*: **assumes** $0 \leq x$ **and** $0 < y$
shows $0 \leq$ *Ifloat* $($*lapprox-posrat* $n\ x\ y)$
⟨*proof*⟩

definition *rapprox-posrat* :: $nat \Rightarrow int \Rightarrow int \Rightarrow$ *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*: $0 \leq x$
and *y*: $0 < y$
shows *real x* / *real y* ≤ *Ifloat* (*rapprox-posrat prec x y*)
⟨*proof*⟩

lemma *rapprox-posrat-le1*: **assumes** $0 \leq x$ **and** $0 < y$ **and** $x \leq y$
shows *Ifloat* (*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 *Ifloat* (*rapprox-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-rat prec x y* = 0
| $0 \leq x \Longrightarrow 0 < y \Longrightarrow$ *lapprox-rat prec x y* = *lapprox-posrat prec x y*

| $x < 0 \Longrightarrow 0 < y \Longrightarrow$ *lapprox-rat prec x y* $= -$ (*rapprox-posrat prec* $(-x)$ *y*)
| $x < 0 \Longrightarrow y < 0 \Longrightarrow$ *lapprox-rat prec x y* $=$ *lapprox-posrat prec* $(-x)$ $(-y)$
| $0 \leq x \Longrightarrow y < 0 \Longrightarrow$ *lapprox-rat prec x y* $= -$ (*rapprox-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*: *Ifloat* (*lapprox-rat prec x y*) ≤ *real x* / *real y*
⟨*proof*⟩

lemma *lapprox-rat-bottom*: **assumes** *0* ≤ *x* **and** *0* < *y*
shows *real* (*x div y*) ≤ *Ifloat* (*lapprox-rat n x y*)
⟨*proof*⟩

function *rapprox-rat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
y = *0* ⟹ *rapprox-rat prec x y* = *0*
| *0* ≤ *x* ⟹ *0* < *y* ⟹ *rapprox-rat prec x y* = *rapprox-posrat prec x y*
| *x* < *0* ⟹ *0* < *y* ⟹ *rapprox-rat prec x y* = − (*lapprox-posrat prec* (−*x*) *y*)
| *x* < *0* ⟹ *y* < *0* ⟹ *rapprox-rat prec x y* = *rapprox-posrat prec* (−*x*) (−*y*)
| *0* ≤ *x* ⟹ *y* < *0* ⟹ *rapprox-rat prec x y* = − (*lapprox-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* ≤ *Ifloat* (*rapprox-rat prec x y*)
⟨*proof*⟩

lemma *rapprox-rat-le1*: **assumes** *0* ≤ *x* **and** *0* < *y* **and** *x* ≤ *y*
shows *Ifloat* (*rapprox-rat n x y*) ≤ *1*
⟨*proof*⟩

lemma *rapprox-rat-neg*: **assumes** *x* < *0* **and** *0* < *y*
shows *Ifloat* (*rapprox-rat n x y*) ≤ *0*
⟨*proof*⟩

lemma *rapprox-rat-nonneg-neg*: **assumes** *0* ≤ *x* **and** *y* < *0*

shows *Ifloat* (*rapprox-rat n x y*) ≤ *0*
⟨*proof*⟩

lemma *rapprox-rat-nonpos-pos*: **assumes** *x* ≤ *0* **and** *0* < *y*
shows *Ifloat* (*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*: *Ifloat* (*float-divl prec x y*) ≤ *Ifloat x* / *Ifloat 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*: *Ifloat x* / *Ifloat y* ≤ *Ifloat* (*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*⟩

fun *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)

fun *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*: *Ifloat x* ≤ *Ifloat* (*round-up prec x*)
⟨*proof*⟩

lemma *round-down*: *Ifloat* (*round-down prec x*) ≤ *Ifloat 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^*nat l*) + 1) (*e* + *l*)
else Float m e)

lemma *lb-mult*: *Ifloat* (*lb-mult prec x y*) ≤ *Ifloat* (*x* ∗ *y*)
⟨*proof*⟩

lemma *ub-mult*: *Ifloat* (*x* ∗ *y*) ≤ *Ifloat* (*ub-mult prec x y*)
⟨*proof*⟩

fun *float-abs* :: *float* ⇒ *float* **where**
float-abs (*Float m e*) = *Float* |*m*| *e*

instantiation *float* :: *abs* **begin**
definition *abs-float-def*: |*x*| = *float-abs x*
instance ⟨*proof*⟩
end

lemma *Ifloat-abs*: *Ifloat* |*x*| = |*Ifloat x*|
⟨*proof*⟩

fun *floor-fl* :: *float* ⇒ *float* **where**

floor-fl (*Float m e*) = (*if* 0 ≤ *e then Float m e*
else Float (*m div* (2 ^ (*nat* (−*e*)))) 0)

lemma *floor-fl*: *Ifloat* (*floor-fl x*) ≤ *Ifloat x*
⟨*proof*⟩

lemma *floor-pos-exp*: **assumes** *floor*: *Float m e* = *floor-fl x* **shows** 0 ≤ *e*
⟨*proof*⟩

declare *floor-fl.simps*[*simp del*]

fun *ceiling-fl* :: *float* ⇒ *float* **where**
ceiling-fl (*Float m e*) = (*if* 0 ≤ *e then Float m e*
else Float (*m div* (2 ^ (*nat* (−*e*))) + 1) 0)

lemma *ceiling-fl*: *Ifloat x* ≤ *Ifloat* (*ceiling-fl x*)
⟨*proof*⟩

declare *ceiling-fl.simps*[*simp del*]

definition *lb-mod* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* ⇒ *float* **where**
lb-mod prec x ub lb = *x* − *ceiling-fl* (*float-divr prec x lb*) ∗ *ub*

definition *ub-mod* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* ⇒ *float* **where**
ub-mod prec x ub lb = *x* − *floor-fl* (*float-divl prec x ub*) ∗ *lb*

lemma *lb-mod*: **fixes** *k* :: *int* **assumes** 0 ≤ *Ifloat x* **and** *real k* ∗ *y* ≤ *Ifloat x* (**is** *?k* ∗ *y* ≤ *?x*)
assumes 0 < *Ifloat lb Ifloat lb* ≤ *y* (**is** *?lb* ≤ *y*) *y* ≤ *Ifloat ub* (**is** *y* ≤ *?ub*)
shows *Ifloat* (*lb-mod prec x ub lb*) ≤ *?x* − *?k* ∗ *y*
⟨*proof*⟩

lemma *ub-mod*: **fixes** *k* :: *int* **assumes** 0 ≤ *Ifloat x* **and** *Ifloat x* ≤ *real k* ∗ *y* (**is** *?x* ≤ *?k* ∗ *y*)
assumes 0 < *Ifloat lb Ifloat lb* ≤ *y* (**is** *?lb* ≤ *y*) *y* ≤ *Ifloat ub* (**is** *y* ≤ *?ub*)
shows *?x* − *?k* ∗ *y* ≤ *Ifloat* (*ub-mod prec x ub lb*)
⟨*proof*⟩

lemma *le-float-def'*: *f* ≤ *g* = (*case f* − *g of Float a b* ⇒ *a* ≤ 0)
⟨*proof*⟩

lemma *float-less-zero*:
(*Ifloat* (*Float a b*) < 0) = (*a* < 0)
⟨*proof*⟩

lemma *less-float-def'*: *f* < *g* = (*case f* − *g of Float a b* ⇒ *a* < 0)
⟨*proof*⟩

end

39 Formal-Power-Series: A formalization of formal power series

theory *Formal-Power-Series*
imports *Main Fact Parity*
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-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* (λ*n*. *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* (λ*n*. *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 = (λf. Abs-fps (λn. − (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 * = (λf g. Abs-fps (λn. ∑ i=0..n. f $ i * g $ (n − i)))*

instance ⟨*proof*⟩
end

lemma *fps-mult-nth*: $(f * g)\ \$\ n = (\sum i{=}0..n.\ f\$i * g\$(n - i))$
⟨*proof*⟩

declare *atLeastAtMost-iff*[*presburger*]
declare *Bex-def*[*presburger*]
declare *Ball-def*[*presburger*]

lemma *mult-delta-left*:
fixes $x\ y :: 'a{::}mult\text{-}zero$
shows $(if\ b\ then\ x\ else\ 0) * y = (if\ b\ then\ x * y\ else\ 0)$
⟨*proof*⟩

lemma *mult-delta-right*:
fixes $x\ y :: 'a{::}mult\text{-}zero$
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** $f :: nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a{::}comm\text{-}monoid\text{-}add$
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** $f :: nat \Rightarrow nat \Rightarrow 'a{::}comm\text{-}monoid\text{-}add$
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-const c = Abs-fps (λn. if n = 0 then c else 0)

lemma *fps-nth-fps-const* [*simp*]: *fps-const c $ n = (if n = 0 then c else 0)*
⟨*proof*⟩

lemma *fps-const-0-eq-0* [*simp*]: *fps-const 0 = 0*
⟨*proof*⟩

lemma *fps-const-1-eq-1* [*simp*]: *fps-const 1 = 1*
⟨*proof*⟩

lemma *fps-const-neg* [*simp*]: *− (fps-const (c::'a::ring)) = fps-const (− c)*
⟨*proof*⟩

lemma *fps-const-add* [*simp*]: *fps-const (c::'a::monoid-add) + fps-const d = fps-const (c + d)*
⟨*proof*⟩

lemma *fps-const-mult*[*simp*]: *fps-const (c::'a::ring) ∗ fps-const d = fps-const (c ∗ d)*
⟨*proof*⟩

lemma *fps-const-add-left*: *fps-const (c::'a::monoid-add) + f = Abs-fps (λn. 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 (λn. 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 (λn. c ∗ f$n)*
⟨*proof*⟩

lemma *fps-const-mult-right*: *f ∗ fps-const (c::'a::semiring-0) = Abs-fps (λn. 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* :: (*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

39.6 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* ⇒ *nat* ⇒ '*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*⟩

instance *fps* :: ({*comm-monoid-add*,*inverse*, *times*, *uminus*}) *division-by-zero*
⟨*proof*⟩

lemma *inverse-mult-eq-1*[*intro*]: **assumes** *f0*: $f\$0 \neq (0::'a::field)$
shows $inverse\ f * f = 1$
⟨*proof*⟩

lemma *fps-inverse-0-iff*[*simp*]: $(inverse\ f)\$0 = (0::'a::division\text{-}ring) \longleftrightarrow f\$0 = 0$
⟨*proof*⟩

lemma *fps-inverse-eq-0-iff*[*simp*]: $inverse\ f = (0::('a::field)\ fps) \longleftrightarrow f\ \$0 = 0$
⟨*proof*⟩

lemma *fps-inverse-idempotent*[*intro*]: **assumes** *f0*: $f\$0 \neq (0::'a::field)$
shows $inverse\ (inverse\ f) = f$
⟨*proof*⟩

lemma *fps-inverse-unique*: **assumes** *f0*: $f\$0 \neq (0::'a::field)$ **and** *fg*: $f*g = 1$
shows $inverse\ f = g$
⟨*proof*⟩

lemma *fps-inverse-gp*: $inverse\ (Abs\text{-}fps(\lambda n.\ (1::'a::field)))$
$= Abs\text{-}fps\ (\lambda n.\ if\ n= 0\ then\ 1\ else\ if\ n=1\ then\ -\ 1\ else\ 0)$
⟨*proof*⟩

39.7 Formal Derivatives, and the MacLaurin theorem around 0

definition $fps\text{-}deriv\ f = Abs\text{-}fps\ (\lambda n.\ of\text{-}nat\ (n + 1) * f\ \$\ (n + 1))$

lemma *fps-deriv-nth*[*simp*]: $fps\text{-}deriv\ f\ \$\ n = of\text{-}nat\ (n +1) * f\ \$\ (n+1)$ ⟨*proof*⟩

lemma *fps-deriv-linear*[*simp*]: $fps\text{-}deriv\ (fps\text{-}const\ (a::'a::comm\text{-}semiring\text{-}1) * f + fps\text{-}const\ b * g) = fps\text{-}const\ a * fps\text{-}deriv\ f + fps\text{-}const\ b * fps\text{-}deriv\ g$
⟨*proof*⟩

lemma *fps-deriv-mult*[*simp*]:
fixes $f :: ('a :: comm\text{-}ring\text{-}1)\ fps$
shows $fps\text{-}deriv\ (f * g) = f * fps\text{-}deriv\ g + fps\text{-}deriv\ f * g$
⟨*proof*⟩

lemma *fps-deriv-neg*[*simp*]: $fps\text{-}deriv\ (-\ (f::('a::comm\text{-}ring\text{-}1)\ fps)) = -\ (fps\text{-}deriv\ f)$
⟨*proof*⟩
lemma *fps-deriv-add*[*simp*]: $fps\text{-}deriv\ ((f::('a::comm\text{-}ring\text{-}1)\ fps) + g) = fps\text{-}deriv\ f + fps\text{-}deriv\ g$
⟨*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* (λ*i*. *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*⟩

39.8 Powers

instantiation *fps* :: (*semiring-1*) *power*
begin

fun *fps-pow* :: *nat* ⇒ '*a fps* ⇒ '*a fps* **where**
fps-pow 0 f = *1*
| *fps-pow* (*Suc n*) *f* = *f* ∗ *fps-pow n f*

definition *fps-power-def*: *power* (*f*::'*a fps*) *n* = *fps-pow n f*
instance ⟨*proof*⟩
end

instantiation *fps* :: (*comm-ring-1*) *recpower*
begin
instance
⟨*proof*⟩
end

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*, *recpower*}) ⟹ *a*^*n* $*0* = *v*^*n*
⟨*proof*⟩

lemma *startsby-zero-power-iff* [*simp*]:
a^*n* $*0* = (*0*::'*a*::{*idom*, *recpower*}) ⟷ (*n* ≠ *0* ∧ *a*$*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*::{*recpower*, *idom*})
shows *a*^*n* $ *n* = (*a*$*1*) ^ *n*
⟨*proof*⟩

lemma *fps-inverse-power*:
fixes *a* :: ('*a*::{*field*, *recpower*}) *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*⟩

39.9 The eXtractor series X

lemma *minus-one-power-iff*: (− (*1*::*'a* :: {*recpower*, *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 *fps-inverse-gp'*: *inverse* (*Abs-fps*(λ*n*. (*1*::*'a*::*field*)))
= *1* − *X*
⟨*proof*⟩

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 (λn. 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*⟩
lemma *fps-deriv-X*[*simp*]: *fps-deriv X = 1*
⟨*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 *X-nth*[*simp*]: *X$n = (if n = 1 then 1 else 0)* ⟨*proof*⟩
lemma *X-power-nth*[*simp*]: *(X^k) $n = (if n = k then 1 else (0::'a::comm-ring-1))*
⟨*proof*⟩

lemma *fps-inverse-X-plus1*:
inverse (1 + X) = Abs-fps (λn. (− (1::'a::{recpower, field})) ^ n) (**is** *- = ?r*)
⟨*proof*⟩

39.10 Integration

definition *fps-integral a a0 = Abs-fps (λn. if n = 0 then a0 else (a$(n − 1) / of-nat n))*

lemma *fps-deriv-fps-integral*: *fps-deriv (fps-integral a (a0 :: 'a :: {field, ring-char-0})) = a*
⟨*proof*⟩

lemma *fps-integral-linear*: *fps-integral (fps-const (a::'a::{field, ring-char-0}) * 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*⟩

39.11 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 (λn. setsum (λi. a$i * (b^i$n)) {0..n})*

lemma *fps-compose-nth*: *(a oo b)$n = setsum (λi. a$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 *X-fps-compose-startby0*[*simp*]: *a*$*0* = *0* ⟹ *X oo a* = (*a* :: (*'a* :: *comm-ring-1*) *fps*)
⟨*proof*⟩

39.12 Rules from Herbert Wilf's Generatingfunctionology

39.12.1 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*:: *field*) ∗ *X*^*i*) {*0* .. *k*} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

39.12.2 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*, *recpower*, *ring-char-0*}) *fps*) = *Abs-fps* (λ*n*. (*of-nat n* ^ *k*) ∗ *a*$*n*)
⟨*proof*⟩

39.12.3 Rule 3 is trivial and is given by *fps-times-def*

39.12.4 Rule 5 — summation and "division" by (1 - X)

lemma *fps-divide-X-minus1-setsum-lemma*:
$a = ((1::('a::comm\text{-}ring\text{-}1)\ fps) - X) * Abs\text{-}fps\ (\lambda n.\ setsum\ (\lambda i.\ a\ \$\ i)\ \{0..n\})$
⟨*proof*⟩

lemma *fps-divide-X-minus1-setsum*:
$a\ /((1::('a::field)\ fps) - X) = Abs\text{-}fps\ (\lambda n.\ setsum\ (\lambda i.\ a\ \$\ i)\ \{0..n\})$
⟨*proof*⟩

39.12.5 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 $natpermute\ n\ k = \{l::\ nat\ list.\ length\ l = k \wedge foldl\ op\ +\ 0\ l = n\}$

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: $xs@ys \in natpermute\ n\ k$ **shows** $foldl\ op\ +\ 0\ xs \leq n$ **and** $foldl\ op\ +\ 0\ ys \leq n$
⟨*proof*⟩

lemma *natpermute-split*:
assumes mn: $h \leq k$
shows $natpermute\ n\ k = (\bigcup m \in\{0..n\}.\ \{l1\ @\ l2\ |l1\ l2.\ l1 \in natpermute\ m\ h \wedge l2 \in natpermute\ (n - m)\ (k - h)\})$ (**is** $?L = ?R$ **is** $?L = (\bigcup m \in\{0..n\}.\ ?S\ m)$)
⟨*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-ring-1*, *recpower*}) *fps*
shows (*a* ^*m*)$*0* = (*a*$*0*) ^ *m*
⟨*proof*⟩

lemma *fps-compose-inj-right*:
assumes *a0*: *a*$*0* = (*0*::'*a*::{*recpower*,*idom*})
and *a1*: *a*$*1* ≠ *0*
shows (*b oo a* = *c oo a*) ⟷ *b* = *c* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

39.13 Radicals

declare *setprod-cong*[*fundef-cong*]
function *radical* :: (*nat* ⇒ '*a* ⇒ '*a*) ⇒ *nat* ⇒ ('*a*::{*field*, *recpower*}) *fps* ⇒ *nat* ⇒ '*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)) ^ 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≠0*
shows *card {xs ∈ natpermute n (k+1). n ∈ set xs} = k+1*
⟨*proof*⟩

lemma *power-radical*:
fixes *a*:: *'a* ::{*field*, *ring-char-0*, *recpower*} *fps*
assumes *r0*: *(r (Suc k) (a$0)) ^ Suc k = a$0* **and** *a0*: *a$0 ≠ 0*
shows *(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, ring-char-0, recpower}) = 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, ring-char-0, recpower}) ≠ 0*
shows *(fps-radical r (Suc k) (a ^ Suc k)) = a*
⟨*proof*⟩

lemma *fps-deriv-radical*:
fixes *a*:: *'a* ::{*field*, *ring-char-0*, *recpower*} *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*, *ring-char-0*, *recpower*} *fps*
assumes
ra0: *r* (*k*) (*a* $ *0*) ^ *k* = *a* $ *0*
and *rb0*: *r* (*k*) (*b* $ *0*) ^ *k* = *b* $ *0*
and *r0'*: *r* (*k*) ((*a* * *b*) $ *0*) = *r* (*k*) (*a* $ *0*) * *r* (*k*) (*b* $ *0*)
and *a0*: *a$0* ≠ *0*
and *b0*: *b$0* ≠ *0*
shows *fps-radical r* (*k*) (*a*b*) = *fps-radical r* (*k*) *a* * *fps-radical r* (*k*) (*b*)
⟨*proof*⟩

lemma *radical-inverse*:
fixes *a*:: *'a* ::{*field*, *ring-char-0*, *recpower*} *fps*
assumes
ra0: *r* (*k*) (*a* $ *0*) ^ *k* = *a* $ *0*
and *ria0*: *r* (*k*) (*inverse* (*a* $ *0*)) = *inverse* (*r* (*k*) (*a* $ *0*))
and *r1*: (*r* (*k*) *1*) = *1*
and *a0*: *a$0* ≠ *0*
shows *fps-radical r* (*k*) (*inverse a*) = *inverse* (*fps-radical r* (*k*) *a*)
⟨*proof*⟩

lemma *fps-divide-inverse*: (*a*::(*'a*::*field*) *fps*) / *b* = *a* * *inverse b*
⟨*proof*⟩

lemma *radical-divide*:
fixes *a*:: *'a* ::{*field*, *ring-char-0*, *recpower*} *fps*
assumes
ra0: *r k* (*a* $ *0*) ^ *k* = *a* $ *0*
and *rb0*: *r k* (*b* $ *0*) ^ *k* = *b* $ *0*
and *r1*: *r k 1* = *1*
and *rb0'*: *r k* (*inverse* (*b* $ *0*)) = *inverse* (*r k* (*b* $ *0*))
and *raib'*: *r k* (*a$0* / (*b$0*)) = *r k* (*a$0*) / *r k* (*b$0*)
and *a0*: *a$0* ≠ *0*
and *b0*: *b$0* ≠ *0*
shows *fps-radical r k* (*a*/*b*) = *fps-radical r k a* / *fps-radical r k b*
⟨*proof*⟩

39.14 Derivative of composition

lemma *fps-compose-deriv*:
fixes *a*:: (*'a*::*idom*) *fps*

assumes *b0*: *b$0 = 0*
shows *fps-deriv (a oo b) = ((fps-deriv a) oo b) * (fps-deriv b)*
⟨*proof*⟩

lemma *fps-mult-X-plus-1-nth*:
*((1+X)*a) $n = (if n = 0 then (a$n :: 'a::comm-ring-1) else a$n + a$(n − 1))*
⟨*proof*⟩

39.15 Finite FPS (i.e. polynomials) and X

lemma *fps-poly-sum-X*:
assumes *z*: *∀ i > n. a$i = (0::'a::comm-ring-1)*
shows *a = setsum (λi. fps-const (a$i) * X^i) {0..n}* (**is** *a = ?r*)
⟨*proof*⟩

39.16 Compositional inverses

fun *compinv* :: *'a fps ⇒ nat ⇒ 'a::{recpower,field}* **where**
compinv a 0 = X$0
| *compinv a (Suc n) = (X$ Suc n − setsum (λi. (compinv a i) * (a^i)$Suc n) {0 .. n}) / (a$1) ^ Suc n*

definition *fps-inv a = Abs-fps (compinv a)*

lemma *fps-inv*: **assumes** *a0*: *a$0 = 0* **and** *a1*: *a$1 ≠ 0*
shows *fps-inv a oo a = X*
⟨*proof*⟩

fun *gcompinv* :: *'a fps ⇒ 'a fps ⇒ nat ⇒ 'a::{recpower,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-pow-0*: *fps-pow n 0 = (if n = 0 then 1 else 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* (λ*i*. *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 $(a\ oo\ c)\hat{}n = a\hat{}n\ oo\ c$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-const-mult-apply-left*:
$fps\text{-}const\ c * (a\ oo\ b) = (fps\text{-}const\ c * a)\ oo\ b$
⟨*proof*⟩

lemma *fps-const-mult-apply-right*:
$(a\ oo\ b) * fps\text{-}const\ (c::'a::comm\text{-}semiring\text{-}1) = (fps\text{-}const\ c * a)\ oo\ b$
⟨*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** $X\hat{}k\ oo\ a = (a::('a::idom\ fps))\hat{}k$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-inv-right*: **assumes** *a0*: $a\$0 = 0$ **and** *a1*: $a\$1 \neq 0$
shows $a\ oo\ fps\text{-}inv\ a = X$
⟨*proof*⟩

lemma *fps-inv-deriv*:
assumes *a0*:$a\$0 = (0::'a::\{recpower,field\})$ **and** *a1*: $a\$1 \neq 0$
shows $fps\text{-}deriv\ (fps\text{-}inv\ a) = inverse\ (fps\text{-}deriv\ a\ oo\ fps\text{-}inv\ a)$
⟨*proof*⟩

39.17 Elementary series

39.17.1 Exponential series

definition $E\ x = Abs\text{-}fps\ (\lambda n.\ x\hat{}n\ /\ of\text{-}nat\ (fact\ n))$

lemma *E-deriv*[*simp*]: $fps\text{-}deriv\ (E\ a) = fps\text{-}const\ (a::'a::\{field, recpower, ring\text{-}char\text{-}0\}) * E\ a$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *E-unique-ODE*:
$fps\text{-}deriv\ a = fps\text{-}const\ c * a \longleftrightarrow a = fps\text{-}const\ (a\$0) * E\ (c :: 'a::\{field, ring\text{-}char\text{-}0, recpower\})$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *E-add-mult*: $E\ (a + b) = E\ (a::'a::\{ring\text{-}char\text{-}0, field, recpower\}) * E\ b$ (**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, recpower}) = 1*
⟨*proof*⟩

lemma *E-neg*: *E (− a) = inverse (E (a::'a::{ring-char-0, field, recpower}))*
⟨*proof*⟩

lemma *E-nth-deriv*[*simp*]: *fps-nth-deriv n (E (a::'a::{field, recpower, ring-char-0})) = (fps-const a)^n * (E a)*
⟨*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 (λn. if n = 0 then (0::'a::comm-ring-1) else a$n)*
⟨*proof*⟩

lemma *X-compose-E*[*simp*]: *X oo E (a::'a::{field, recpower}) = 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*:
inverse (fps-const (a::'a::{field, division-by-zero})) = fps-const (inverse a)
⟨*proof*⟩

lemma *inverse-one-plus-X*:
inverse (1 + X) = Abs-fps (λn. (− 1 ::'a::{field, recpower})^n)
(**is** *inverse ?l = ?r*)
⟨*proof*⟩

lemma *E-power-mult*: *(E (c::'a::{field,recpower,ring-char-0}))^n = E (of-nat n * c)*
⟨*proof*⟩

39.17.2 Logarithmic series

definition (*L*::'*a*::{*field*, *ring-char-0*,*recpower*} *fps*)
= *Abs-fps* (λ*n*. (− *1*) ^ *Suc n* / *of-nat n*)

lemma *fps-deriv-L*: *fps-deriv L* = *inverse* (*1* + *X*)
⟨*proof*⟩

lemma *L-nth*: *L* $ *n* = (− *1*) ^ *Suc n* / *of-nat n*
⟨*proof*⟩

lemma *L-E-inv*:
assumes *a*: *a*≠ (*0*::'*a*::{*field*,*division-by-zero*,*ring-char-0*,*recpower*})
shows *L* = *fps-const a* ∗ *fps-inv* (*E a* − *1*) (**is** *?l* = *?r*)
⟨*proof*⟩

39.17.3 Formal trigonometric functions

definition *fps-sin* (*c*::'*a*::{*field*, *recpower*, *ring-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*, *recpower*, *ring-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*⟩

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*⟩

end

40 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*. ∀*x*. *x* ∈ *A* −−> *f x* ∈ *B x*}

definition
extensional :: *'a set* => (*'a => 'b*) *set* **where**
extensional A = {*f*. ∀*x*. *x*~:*A* −−> *f x* = *undefined*}

definition
restrict :: [*'a => 'b*, *'a set*] => (*'a => 'b*) **where**
restrict f A = (%*x*. *if x* ∈ *A then f x else undefined*)

abbreviation
funcset :: [*'a set*, *'b set*] => (*'a => 'b*) *set*
(**infixr** −> *60*) **where**
A −> *B* == *Pi A* (%-. *B*)

notation (*xsymbols*)
funcset (**infixr** → *60*)

syntax
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a => 'b*) *set* ((*3PI* -:-./ -) *10*)
-lam :: [*pttrn*, *'a set*, *'a => 'b*] => (*'a=>'b*) ((*3*%-:-./ -) [*0*,*0*,*3*] *3*)

syntax (*xsymbols*)
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a => 'b*) *set* ((*3*Π -∈-./ -) *10*)
-lam :: [*pttrn*, *'a set*, *'a => 'b*] => (*'a=>'b*) ((*3*λ-∈-./ -) [*0*,*0*,*3*] *3*)

syntax (*HTML* **output**)
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a => 'b*) *set* ((*3*Π -∈-./ -) *10*)
-lam :: [*pttrn*, *'a set*, *'a => 'b*] => (*'a=>'b*) ((*3*λ-∈-./ -) [*0*,*0*,*3*] *3*)

translations
PI x:*A*. *B* == *CONST Pi A* (%*x*. *B*)
%*x*:*A*. *f* == *CONST restrict* (%*x*. *f*) *A*

definition
compose :: [*'a set*, *'b => 'c*, *'a => 'b*] => (*'a => 'c*) **where**
compose A g f = (λ*x*∈*A*. *g* (*f x*))

40.1 Basic Properties of *Pi*

lemma *Pi-I*: (!!*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 *funcset-mem*: $[|f \in A -> B;\ x \in A|] ==> f\ x \in B$
⟨*proof*⟩

lemma *funcset-image*: $f \in A \to B ==> f\ `\ A \subseteq B$
⟨*proof*⟩

lemma *Pi-eq-empty*: $((PI\ x: A.\ B\ x) = \{\}) = (\exists x \in A.\ B(x) = \{\})$
⟨*proof*⟩

lemma *Pi-empty* [*simp*]: $Pi\ \{\}\ B = UNIV$
⟨*proof*⟩

lemma *Pi-UNIV* [*simp*]: $A -> UNIV = UNIV$
⟨*proof*⟩

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(!!x.\ x \in A ==> B\ x <= C\ x) ==> Pi\ A\ B <= Pi\ A\ C$
⟨*proof*⟩

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' <= A ==> Pi\ A\ B <= Pi\ A'\ B$
⟨*proof*⟩

40.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*:
$[|\ f \in A -> B;\ g \in B -> C\ |]==> compose\ A\ g\ f \in A -> C$
⟨*proof*⟩

lemma *compose-assoc*:
$[|\ f \in A -> B;\ g \in B -> C;\ h \in C -> D\ |]$
$==> compose\ A\ h\ (compose\ A\ g\ f) = compose\ A\ (compose\ B\ h\ g)\ f$
⟨*proof*⟩

lemma *compose-eq*: $x \in A ==> compose\ A\ g\ f\ x = g(f(x))$
⟨*proof*⟩

lemma *surj-compose*: $[|\ f\ `\ A = B;\ g\ `\ B = C\ |] ==> compose\ A\ g\ f\ `\ A = C$
⟨*proof*⟩

40.3 Bounded Abstraction: *restrict*

lemma *restrict-in-funcset*: $(!!x.\ x \in A ==> f\ x \in B) ==> (\lambda x{\in}A.\ f\ x) \in A -> B$
⟨*proof*⟩

lemma *restrictI*: $(!!x.\ x \in A ==> f\ x \in B\ x) ==> (\lambda x{\in}A.\ f\ x) \in Pi\ A\ B$
⟨*proof*⟩

lemma *restrict-apply* [*simp*]:
$(\lambda y{\in}A.\ f\ y)\ x = (if\ x \in A\ then\ f\ x\ else\ undefined)$
⟨*proof*⟩

lemma *restrict-ext*:
$(!!x.\ x \in A ==> f\ x = g\ x) ==> (\lambda x{\in}A.\ f\ x) = (\lambda x{\in}A.\ g\ x)$
⟨*proof*⟩

lemma *inj-on-restrict-eq* [*simp*]: *inj-on* $(restrict\ f\ A)\ A = inj\text{-}on\ f\ A$
⟨*proof*⟩

lemma *Id-compose*:
$[|f \in A -> B;\ f \in extensional\ A|] ==> compose\ A\ (\lambda y{\in}B.\ y)\ f = f$
⟨*proof*⟩

lemma *compose-Id*:
$[|g \in A -> B;\ g \in extensional\ A|] ==> compose\ A\ g\ (\lambda x{\in}A.\ x) = g$
⟨*proof*⟩

lemma *image-restrict-eq* [*simp*]: $(restrict\ f\ A)\ `\ A = f\ `\ A$
⟨*proof*⟩

40.4 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*: $bij\text{-}betw\ f\ A\ B \Longrightarrow f \in A \to B$
⟨*proof*⟩

lemma *inj-on-compose*:
$[|\ bij\text{-}betw\ f\ A\ B;\ inj\text{-}on\ g\ B\ |] ==> inj\text{-}on\ (compose\ A\ g\ f)\ A$
⟨*proof*⟩

lemma *bij-betw-compose*:
$[|\ bij\text{-}betw\ f\ A\ B;\ bij\text{-}betw\ g\ B\ C\ |] ==> bij\text{-}betw\ (compose\ A\ g\ f)\ A\ C$
⟨*proof*⟩

lemma *bij-betw-restrict-eq* [*simp*]:
$bij\text{-}betw\ (restrict\ f\ A)\ A\ B = bij\text{-}betw\ f\ A\ B$
⟨*proof*⟩

40.5 Extensionality

lemma *extensional-arb*: $[|f \in \mathit{extensional}\ A;\ x \notin A|] ==> f\ x = \mathit{undefined}$
⟨*proof*⟩

lemma *restrict-extensional* [*simp*]: $\mathit{restrict}\ f\ A \in \mathit{extensional}\ A$
⟨*proof*⟩

lemma *compose-extensional* [*simp*]: $\mathit{compose}\ A\ f\ g \in \mathit{extensional}\ A$
⟨*proof*⟩

lemma *extensionalityI*:
$[|\ f \in \mathit{extensional}\ A;\ g \in \mathit{extensional}\ A;$
$\ !!x.\ x \in A ==> f\ x = g\ x\ |] ==> f = g$
⟨*proof*⟩

lemma *Inv-funcset*: $f\ `\ A = B ==> (\lambda x \in B.\ \mathit{Inv}\ A\ f\ x) : B -> A$
⟨*proof*⟩

lemma *compose-Inv-id*:
$\mathit{bij\text{-}betw}\ f\ A\ B ==> \mathit{compose}\ A\ (\lambda y \in B.\ \mathit{Inv}\ A\ f\ y)\ f = (\lambda x \in A.\ x)$
⟨*proof*⟩

lemma *compose-id-Inv*:
$f\ `\ A = B ==> \mathit{compose}\ B\ f\ (\lambda y \in B.\ \mathit{Inv}\ A\ f\ y) = (\lambda x \in B.\ x)$
⟨*proof*⟩

40.6 Cardinality

lemma *card-inj*: $[|f \in A \to B;\ \mathit{inj\text{-}on}\ f\ A;\ \mathit{finite}\ B|] ==> \mathit{card}(A) \leq \mathit{card}(B)$
⟨*proof*⟩

lemma *card-bij*:
$[|f \in A \to B;\ \mathit{inj\text{-}on}\ f\ A;$
$\ g \in B \to A;\ \mathit{inj\text{-}on}\ g\ B;\ \mathit{finite}\ A;\ \mathit{finite}\ B|] ==> \mathit{card}(A) = \mathit{card}(B)$
⟨*proof*⟩

end

41 Polynomial: Univariate Polynomials

theory *Polynomial*
imports *Main*
begin

41.1 Definition of type *poly*

typedef (*Poly*) $'a\ \mathit{poly} = \{f :: \mathit{nat} \Rightarrow 'a :: \mathit{zero}.\ \exists n.\ \forall i > n.\ f\ i = 0\}$
morphisms *coeff Abs-poly*

⟨*proof*⟩

lemma *expand-poly-eq*: $p = q \longleftrightarrow (\forall n.\ coeff\ p\ n = coeff\ q\ n)$
⟨*proof*⟩

lemma *poly-ext*: $(\bigwedge n.\ coeff\ p\ n = coeff\ q\ n) \Longrightarrow p = q$
⟨*proof*⟩

41.2 Degree of a polynomial

definition
degree :: *'a*::*zero poly* ⇒ *nat* **where**
$degree\ p = (LEAST\ n.\ \forall i{>}n.\ coeff\ p\ i = 0)$

lemma *coeff-eq-0*: $degree\ p < n \Longrightarrow coeff\ p\ n = 0$
⟨*proof*⟩

lemma *le-degree*: $coeff\ p\ n \neq 0 \Longrightarrow n \leq degree\ p$
⟨*proof*⟩

lemma *degree-le*: $\forall i{>}n.\ coeff\ p\ i = 0 \Longrightarrow degree\ p \leq n$
⟨*proof*⟩

lemma *less-degree-imp*: $n < degree\ p \Longrightarrow \exists i{>}n.\ coeff\ p\ i \neq 0$
⟨*proof*⟩

41.3 The zero polynomial

instantiation *poly* :: (*zero*) *zero*
begin

definition
zero-poly-def: $0 = Abs\text{-}poly\ (\lambda n.\ 0)$

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 $p \neq 0$ **shows** $coeff\ p\ (degree\ p) \neq 0$
⟨*proof*⟩

lemma *leading-coeff-0-iff* [*simp*]: $coeff\ p\ (degree\ p) = 0 \longleftrightarrow p = 0$
⟨*proof*⟩

41.4 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*: $f \in Poly \Longrightarrow (\lambda n.\ f\ (Suc\ n)) \in Poly$
⟨*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*: $\bigwedge a\ p.\ P\ p \Longrightarrow P\ (pCons\ a\ p)$
shows $P\ p$
⟨*proof*⟩

41.5 Recursion combinator for polynomials

function
$poly\text{-}rec :: 'b \Rightarrow ('a::zero \Rightarrow 'a\ poly \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ poly \Rightarrow 'b$
where
poly-rec-pCons-eq-if [*simp del*, *code del*]:
$poly\text{-}rec\ z\ f\ (pCons\ a\ p) = f\ a\ p\ (if\ p = 0\ then\ z\ else\ poly\text{-}rec\ z\ f\ p)$
⟨*proof*⟩

termination *poly-rec*
⟨*proof*⟩

lemma *poly-rec-0*:
$f\ 0\ 0\ z = z \Longrightarrow poly\text{-}rec\ z\ f\ 0 = z$
⟨*proof*⟩

lemma *poly-rec-pCons*:
$f\ 0\ 0\ z = z \Longrightarrow poly\text{-}rec\ z\ f\ (pCons\ a\ p) = f\ a\ p\ (poly\text{-}rec\ z\ f\ p)$
⟨*proof*⟩

41.6 Monomials

definition
$monom :: 'a \Rightarrow nat \Rightarrow 'a::zero\ poly$ **where**
$monom\ a\ m = Abs\text{-}poly\ (\lambda n.\ if\ m = n\ then\ a\ else\ 0)$

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*⟩

41.7 Addition and subtraction

instantiation *poly* :: (*comm-monoid-add*) *comm-monoid-add*
begin

definition
plus-poly-def [*code del*]:
p + q = Abs-poly (λn. coeff p n + coeff q n)

lemma *Poly-add*:
fixes *f g* :: *nat ⇒ 'a*
shows ⟦*f ∈ Poly*; *g ∈ Poly*⟧ *⟹ (λn. 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 (λn. − coeff p n)

definition
minus-poly-def [*code del*]:

$p - q = Abs\text{-}poly\ (\lambda n.\ coeff\ p\ n - coeff\ q\ n)$

lemma *Poly-minus*:
fixes $f :: nat \Rightarrow 'a$
shows $f \in Poly \Longrightarrow (\lambda n.\ -f\ n) \in Poly$
⟨*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 ⟹ degree (p + q) = degree p
⟨*proof*⟩

lemma *degree-minus* [*simp*]: *degree (− p) = degree p*
⟨*proof*⟩

lemma *degree-diff-le-max*: *degree (p − q) ≤ max (degree p) (degree q)*
⟨*proof*⟩

lemma *degree-diff-le*:
⟦degree p ≤ n; degree q ≤ n⟧ ⟹ degree (p − q) ≤ n
⟨*proof*⟩

lemma *degree-diff-less*:
⟦degree p < n; degree q < n⟧ ⟹ 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 (∑ x∈A. p x) i = (∑ x∈A. coeff (p x) i)*
⟨*proof*⟩

lemma *monom-setsum*: *monom (∑ x∈A. a x) n = (∑ x∈A. monom (a x) n)*
⟨*proof*⟩

41.8 Multiplication by a constant

definition
smult :: 'a::comm-semiring-0 ⇒ 'a poly ⇒ 'a poly **where**
*smult a p = Abs-poly (λn. a * coeff p n)*

lemma *Poly-smult*:
fixes *f :: nat ⇒ 'a::comm-semiring-0*
shows *f ∈ Poly ⟹ (λn. a * f n) ∈ Poly*
⟨*proof*⟩

lemma *coeff-smult* [*simp*]: *coeff (smult a p) n = a * coeff p n*
⟨*proof*⟩

lemma *degree-smult-le*: *degree (smult a p) ≤ 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*⟩

41.9 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 $(p * q)$ $n = (\sum i{\leq}n.$ *coeff* p $i *$ *coeff* q $(n-i))$
⟨*proof*⟩

lemma *degree-mult-le*: *degree* $(p * q) \leq$ *degree* $p +$ *degree* q
⟨*proof*⟩

lemma *mult-monom*: *monom* a $m *$ *monom* b $n =$ *monom* $(a * b)$ $(m + n)$
⟨*proof*⟩

41.10 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* $q \Longrightarrow p$ *dvd smult* a q
⟨*proof*⟩

lemma *dvd-smult-cancel*:
fixes a :: *'a*::*field*
shows p *dvd smult* a $q \Longrightarrow a \neq 0 \Longrightarrow p$ *dvd* q
⟨*proof*⟩

lemma *dvd-smult-iff*:

fixes $a :: 'a$::*field*
shows $a \neq 0 \Longrightarrow p$ *dvd smult* $a\ q \longleftrightarrow p$ *dvd* q
⟨*proof*⟩

instantiation *poly* :: (*comm-semiring-1*) *recpower*
begin

primrec *power-poly* **where**
$(p::'a\ poly)\ \hat{}\ 0 = 1$
| $(p::'a\ poly)\ \hat{}\ (Suc\ n) = p * p\ \hat{}\ n$

instance
⟨*proof*⟩

declare *power-poly.simps* [*simp del*]

end

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 $k = (of\text{-}int\ k :: 'a\ poly)$

instance
⟨*proof*⟩

end

41.11 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* ≠ *0*⟧ ⟹ *degree p* ≤ *degree q*
⟨*proof*⟩

41.12 Polynomials form an ordered integral domain

definition
pos-poly :: *'a*::*ordered-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* :: (*ordered-idom*) *ordered-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

41.13 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*⟩

41.14 Evaluation of polynomials

definition
$poly :: 'a::comm\text{-}semiring\text{-}0\ poly \Rightarrow 'a \Rightarrow 'a$ **where**
$poly = poly\text{-}rec\ (\lambda x.\ 0)\ (\lambda a\ p\ f\ x.\ a + x * f\ x)$

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 $a\ x :: 'a::\{comm\text{-}semiring\text{-}1,recpower\}$
shows $poly\ (monom\ a\ n)\ x = a * x\ \hat{}\ n$
⟨*proof*⟩

lemma *poly-add* [*simp*]: $poly\ (p + q)\ x = poly\ p\ x + poly\ q\ x$
⟨*proof*⟩

lemma *poly-minus* [*simp*]:
fixes $x :: 'a::comm\text{-}ring$
shows $poly\ (-\ p)\ x = -\ poly\ p\ x$
⟨*proof*⟩

lemma *poly-diff* [*simp*]:
fixes $x :: 'a::comm\text{-}ring$
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*,*recpower*} *poly*
shows *poly* (*p* ^ *n*) *x* = *poly p x* ^ *n*
⟨*proof*⟩

41.15 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*) (λ*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* =
(λ(*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*:

$p + smult\ c\ (synthetic\text{-}div\ p\ c) = pCons\ (poly\ p\ c)\ (synthetic\text{-}div\ p\ c)$
⟨*proof*⟩

lemma *synthetic-div-unique-lemma*: $smult\ c\ p = pCons\ a\ p \Longrightarrow p = 0$
⟨*proof*⟩

lemma *synthetic-div-unique*:
$p + smult\ c\ q = pCons\ r\ q \Longrightarrow r = poly\ p\ c \wedge q = synthetic\text{-}div\ p\ c$
⟨*proof*⟩

lemma *synthetic-div-correct'*:
fixes $c :: 'a::comm\text{-}ring\text{-}1$
shows $[:-c,\ 1:] * synthetic\text{-}div\ p\ c + [:poly\ p\ c:] = p$
⟨*proof*⟩

lemma *poly-eq-0-iff-dvd*:
fixes $c :: 'a::idom$
shows $poly\ p\ c = 0 \longleftrightarrow [:-c,\ 1:]\ dvd\ p$
⟨*proof*⟩

lemma *dvd-iff-poly-eq-0*:
fixes $c :: 'a::idom$
shows $[:c,\ 1:]\ dvd\ p \longleftrightarrow poly\ p\ (-c) = 0$
⟨*proof*⟩

lemma *poly-roots-finite*:
fixes $p :: 'a::idom\ poly$
shows $p \neq 0 \Longrightarrow finite\ \{x.\ poly\ p\ x = 0\}$
⟨*proof*⟩

lemma *poly-zero*:
fixes $p :: 'a::\{idom,ring\text{-}char\text{-}0\}\ poly$
shows $poly\ p = poly\ 0 \longleftrightarrow p = 0$
⟨*proof*⟩

lemma *poly-eq-iff*:
fixes $p\ q :: 'a::\{idom,ring\text{-}char\text{-}0\}\ poly$
shows $poly\ p = poly\ q \longleftrightarrow p = q$
⟨*proof*⟩

41.16 Composition of polynomials

definition
$pcompose :: 'a::comm\text{-}semiring\text{-}0\ poly \Rightarrow 'a\ poly \Rightarrow 'a\ poly$
where
$pcompose\ p\ q = poly\text{-}rec\ 0\ (\lambda a\ c.\ [:a:] + q * c)\ p$

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*⟩

41.17 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 ≠ *0* ⟹ [:−*a*, *1*:] ^ *order a p dvd p* ∧ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
⟨*proof*⟩

lemma *order-degree*:
assumes *p*: *p* ≠ *0*
shows *order a p* ≤ *degree p*
⟨*proof*⟩

lemma *order-root*: *poly p a* = *0* ⟷ *p* = *0* ∨ *order a p* ≠ *0*
⟨*proof*⟩

41.18 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-class.eq (*p*::'*a poly*) *q* ⟷ *p* = *q*

instance
⟨*proof*⟩

end

lemma *eq-poly-code* [*code*]:
eq-class.eq (*0*::- *poly*) (*0*::- *poly*) ⟷ *True*
eq-class.eq (*0*::- *poly*) (*pCons b q*) ⟷ *eq-class.eq 0 b* ∧ *eq-class.eq 0 q*
eq-class.eq (*pCons a p*) (*0*::- *poly*) ⟷ *eq-class.eq a 0* ∧ *eq-class.eq p 0*
eq-class.eq (*pCons a p*) (*pCons b q*) ⟷ *eq-class.eq a b* ∧ *eq-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* :: - *poly*)
p + *0* = (*p* :: - *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*⟩

end

42 Fundamental-Theorem-Algebra: Fundamental Theorem of Algebra

theory *Fundamental-Theorem-Algebra*
imports *Polynomial Complex*
begin

42.1 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*⟩

42.2 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*⟩

42.3 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* (λ*a 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*⟩

42.4 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*^*n* if nonzero

lemma *reduce-poly-simple*:
assumes *b*: *b* ≠ *0* **and** *n*: *n*≠*0*
shows ∃ *z*. *cmod* (*1* + *b* ∗ *z*^*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*: ∀ *n*. *cmod* (*s n*) ≤ *r*
shows ∃ *f z*. *subseq f* ∧ (∀ *e* >*0*. ∃ *N*. ∀ *n* ≥ *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 $constant\ f = (\forall\, x\ y.\ f\ x = f\ y)$

lemma *nonconstant-length*: $\neg\ (constant\ (poly\ p)) \Longrightarrow psize\ p \geq 2$
⟨*proof*⟩

lemma *poly-replicate-append*:
$poly\ (monom\ 1\ n * p)\ (x::'a::\{recpower,\ comm\text{-}ring\text{-}1\}) = x\hat{}n * poly\ p\ x$
⟨*proof*⟩

Decomposition of polynomial, skipping zero coefficients after the first.

lemma *poly-decompose-lemma*:
assumes *nz*: $\neg(\forall\, z.\ z{\neq}0 \longrightarrow poly\ p\ z = (0::'a::\{recpower,idom\}))$
shows $\exists\, k\ a\ q.\ a{\neq}0 \wedge Suc\ (psize\ q + k) = psize\ p \wedge (\forall\, z.\ poly\ p\ z = z\hat{}k * poly\ (pCons\ a\ q)\ z)$
⟨*proof*⟩

lemma *poly-decompose*:
assumes *nc*: $\sim constant(poly\ p)$

shows $\exists k\ a\ q.\ a\neq(0::'a::\{recpower,idom\}) \wedge k\neq 0 \wedge$
$psize\ q + k + 1 = psize\ p \wedge$
$(\forall z.\ poly\ p\ z = poly\ p\ 0 + z\hat{}k * poly\ (pCons\ a\ q)\ z)$
⟨*proof*⟩

Fundamental theorem of algebral

lemma *fundamental-theorem-of-algebra*:
assumes *nc*: $\sim constant(poly\ p)$
shows $\exists z::complex.\ poly\ p\ z = 0$
⟨*proof*⟩

Alternative version with a syntactic notion of constant polynomial.

lemma *fundamental-theorem-of-algebra-alt*:
assumes *nc*: $\sim(\exists a\ l.\ a\neq 0 \wedge l = 0 \wedge p = pCons\ a\ l)$
shows $\exists z.\ poly\ p\ z = (0::complex)$
⟨*proof*⟩

42.5 Nullstellenstatz, degrees and divisibility of polynomials

lemma *nullstellensatz-lemma*:
fixes $p :: complex\ poly$
assumes $\forall x.\ poly\ p\ x = 0 \longrightarrow poly\ q\ x = 0$
and $degree\ p = n$ **and** $n \neq 0$
shows $p\ dvd\ (q\ \hat{}\ n)$
⟨*proof*⟩

lemma *nullstellensatz-univariate*:
$(\forall x.\ poly\ p\ x = (0::complex) \longrightarrow poly\ q\ x = 0) \longleftrightarrow$
$p\ dvd\ (q\ \hat{}\ (degree\ p)) \vee (p = 0 \wedge q = 0)$
⟨*proof*⟩

Useful lemma

lemma *constant-degree*:
fixes $p :: 'a::\{idom,ring\text{-}char\text{-}0\}\ poly$
shows $constant\ (poly\ p) \longleftrightarrow degree\ p = 0$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *divides-degree*: **assumes** *pq*: $p\ dvd\ (q::\ complex\ poly)$
shows $degree\ p \leq degree\ q \vee q = 0$
⟨*proof*⟩

lemma *mpoly-base-conv*:
$(0::complex) \equiv poly\ 0\ x\ c \equiv poly\ [:c:]\ x\ x \equiv poly\ [:0,1:]\ x$ ⟨*proof*⟩

lemma *mpoly-norm-conv*:
$poly\ [:0:]\ (x::complex) \equiv poly\ 0\ x\ poly\ [:poly\ 0\ y:]\ x \equiv poly\ 0\ x$ ⟨*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*≠ (*0*::*complex*) **and** *pp'*: (*p*::*complex poly*) *dvd p'*
and *qrp'*: *smult a q* − *p'* ≡ *r*
shows *p dvd q* ≡ *p dvd* (*r*::*complex poly*) (**is** *?lhs* ≡ *?rhs*)
⟨*proof*⟩

lemma *basic-cqe-conv1*:
(∃ *x*. *poly p x* = *0* ∧ *poly 0 x* ≠ *0*) ≡ *False*
(∃ *x*. *poly 0 x* ≠ *0*) ≡ *False*
(∃ *x*. *poly* [:*c*:] *x* ≠ *0*) ≡ *c*≠*0*
(∃ *x*. *poly 0 x* = *0*) ≡ *True*
(∃ *x*. *poly* [:*c*:] *x* = *0*) ≡ *c* = *0* ⟨*proof*⟩

lemma *basic-cqe-conv2*:
assumes *l*:*p* ≠ *0*
shows (∃ *x*. *poly* (*pCons a* (*pCons b p*)) *x* = (*0*::*complex*)) ≡ *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* ≡ *?r*)
⟨*proof*⟩
lemma *poly-const-conv*: $poly\ [:c:]\ (x::complex) = y \longleftrightarrow c = y$ ⟨*proof*⟩

end

43 Lattice-Syntax: Pretty syntax for lattice operations

44 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*⟩

44.1 + 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*⟩

44.2 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*: ∀ *c*∈*set cs*. *c* = *0* ⟹ ⟨*cs*,*xs*⟩ = *0*
⟨*proof*⟩

lemma *iprod-uminus*[*simp*]: ⟨−*xs*,*ys*⟩ = −⟨*xs*,*ys*⟩
⟨*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

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

theory *Mapping*
imports *Map Main*
begin

45.1 Type definition and primitive operations

datatype $('a,\ 'b)\ map = Map\ 'a \rightharpoonup 'b$

definition *empty* :: $('a,\ 'b)\ map$ **where**
$empty = Map\ (\lambda\text{-}.\ None)$

primrec *lookup* :: $('a,\ 'b)\ map \Rightarrow 'a \rightharpoonup 'b$ **where**
$lookup\ (Map\ f) = f$

primrec *update* :: $'a \Rightarrow 'b \Rightarrow ('a,\ 'b)\ map \Rightarrow ('a,\ 'b)\ map$ **where**
$update\ k\ v\ (Map\ f) = Map\ (f\ (k \mapsto v))$

primrec *delete* :: $'a \Rightarrow ('a,\ 'b)\ map \Rightarrow ('a,\ 'b)\ map$ **where**
$delete\ k\ (Map\ f) = Map\ (f\ (k := None))$

primrec *keys* :: $('a,\ 'b)\ map \Rightarrow 'a\ set$ **where**
$keys\ (Map\ f) = dom\ f$

45.2 Derived operations

definition *size* :: $('a,\ 'b)\ map \Rightarrow nat$ **where**
$size\ m = (if\ finite\ (keys\ m)\ then\ card\ (keys\ m)\ else\ 0)$

definition *replace* :: $'a \Rightarrow 'b \Rightarrow ('a,\ 'b)\ map \Rightarrow ('a,\ 'b)\ map$ **where**
$replace\ k\ v\ m = (if\ lookup\ m\ k = None\ then\ m\ else\ update\ k\ v\ m)$

definition *tabulate* :: $'a\ list \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a,\ 'b)\ map$ **where**
$tabulate\ ks\ f = Map\ (map\text{-}of\ (map\ (\lambda k.\ (k,\ f\ k))\ ks))$

definition *bulkload* :: $'a\ list \Rightarrow (nat,\ 'a)\ map$ **where**

bulkload xs = Map (λk. if k < length xs then Some (xs ! k) else None)

45.3 Properties

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

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

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

lemma *lookup-delete*:
lookup (delete k m) k = None
k ≠ l ⟹ lookup (delete k m) l = lookup m l
⟨*proof*⟩

lemma *lookup-tabulate*:
lookup (tabulate ks f) = (Some o f) |‘ set ks
⟨*proof*⟩

lemma *lookup-bulkload*:
lookup (bulkload xs) = (λk. 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 ≠ l ⟹ update k v (update l w m) = update l w (update k v m)
⟨*proof*⟩

lemma *replace-update*:
lookup m k = None ⟹ replace k v m = m
lookup m k ≠ None ⟹ replace k v m = update k v m
⟨*proof*⟩

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

lemma *delete-update*:
delete k (update k v m) = delete k m
k ≠ l ⟹ delete k (update l v m) = update l v (delete k m)
⟨*proof*⟩

lemma *update-delete* [*simp*]:

update k v (delete k m) = update k v 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-tabulate* [*simp*]:
keys (tabulate ks f) = set ks
⟨*proof*⟩

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

lemma *size-update*:
finite (keys m) ⟹ size (update k v m) =
(if k ∈ keys m then size m else Suc (size m))
⟨*proof*⟩

lemma *size-delete*:
size (delete k m) = (if k ∈ keys m then size m − 1 else size m)
⟨*proof*⟩

lemma *size-tabulate*:
size (tabulate ks f) = length (remdups ks)
⟨*proof*⟩

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

end

46 Multiset: Multisets

theory *Multiset*
imports *List Main*
begin

46.1 The type of multisets

typedef *'a multiset = {f::'a => nat. finite {x . f x > 0}}*
⟨*proof*⟩

lemmas *multiset-typedef* [*simp*] =
Abs-multiset-inverse Rep-multiset-inverse Rep-multiset
and [*simp*] = *Rep-multiset-inject* [*symmetric*]

definition *Mempty* :: *'a multiset* ({#}) **where**
[*code del*]: *{#} = Abs-multiset (λa. 0)*

definition *single* :: *'a => 'a multiset* **where**
[*code del*]: *single a = Abs-multiset (λb. if b = a then 1 else 0)*

definition *count* :: *'a multiset => 'a => nat* **where**
count = Rep-multiset

definition *MCollect* :: *'a multiset => ('a => bool) => 'a multiset* **where**
MCollect M P = Abs-multiset (λx. if P x then Rep-multiset M x else 0)

abbreviation *Melem* :: *'a => 'a multiset => bool* ((-/ :# -) [*50*, *51*] *50*) **where**
a :# M == 0 < count M a

notation (*xsymbols*)
Melem (**infix** ∈# *50*)

syntax
-MCollect :: *pttrn => 'a multiset => bool => 'a multiset* ((*1*{# - :# -./ -#}))
translations
{#x :# M. P#} == CONST MCollect M (λx. P)

definition *set-of* :: *'a multiset => 'a set* **where**
set-of M = {x. x :# M}

instantiation *multiset* :: (*type*) {*plus*, *minus*, *zero*, *size*}
begin

definition *union-def* [*code del*]:
M + N = Abs-multiset (λa. Rep-multiset M a + Rep-multiset N a)

definition *diff-def* [*code del*]:
M − N = Abs-multiset (λa. Rep-multiset M a − Rep-multiset N a)

definition *Zero-multiset-def* [*simp*]:
0 = {#}

definition *size-def*:
size M = setsum (count M) (set-of M)

instance ⟨*proof*⟩

end

definition
multiset-inter :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* (**infixl** #∩ *70*) **where**
multiset-inter A B = *A* − (*A* − *B*)

Multiset Enumeration

syntax
-multiset :: *args* => *'a multiset* ({#(-)#})
translations
{#*x*, *xs*#} == {#*x*#} + {#*xs*#}
{#*x*#} == *CONST single x*

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*: (λ*a*. *0*) ∈ *multiset*
⟨*proof*⟩

lemma *only1-in-multiset*: (λ*b*. *if b* = *a then 1 else 0*) ∈ *multiset*
⟨*proof*⟩

lemma *union-preserves-multiset*:
M ∈ *multiset* ==> *N* ∈ *multiset* ==> (λ*a*. *M a* + *N a*) ∈ *multiset*
⟨*proof*⟩

lemma *diff-preserves-multiset*:
M ∈ *multiset* ==> (λ*a*. *M a* − *N a*) ∈ *multiset*
⟨*proof*⟩

lemma *MCollect-preserves-multiset*:
M ∈ *multiset* ==> (λ*x*. *if P x then M x else 0*) ∈ *multiset*
⟨*proof*⟩

lemmas *in-multiset* = *const0-in-multiset only1-in-multiset*
union-preserves-multiset diff-preserves-multiset MCollect-preserves-multiset

46.2 Algebraic properties

46.2.1 Union

lemma *union-empty* [*simp*]: *M* + {#} = *M* ∧ {#} + *M* = *M*
⟨*proof*⟩

lemma *union-commute*: *M* + *N* = *N* + (*M*::*'a multiset*)
⟨*proof*⟩

lemma *union-assoc*: (*M* + *N*) + *K* = *M* + (*N* + (*K*::*'a multiset*))

⟨*proof*⟩

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
⟨*proof*⟩

lemmas *union-ac* = *union-assoc union-commute union-lcomm*

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

46.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
⟨*proof*⟩

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
⟨*proof*⟩

lemma *diff-cancel*: $A - A = \{\#\}$
⟨*proof*⟩

46.2.3 Count of elements

lemma *count-empty* [*simp*]: *count* $\{\#\}\ a = 0$
⟨*proof*⟩

lemma *count-single* [*simp*]: *count* $\{\#b\#\}\ a = (\textit{if}\ b = a\ \textit{then}\ 1\ \textit{else}\ 0)$
⟨*proof*⟩

lemma *count-union* [*simp*]: *count* $(M + N)\ a = \textit{count}\ M\ a + \textit{count}\ N\ a$
⟨*proof*⟩

lemma *count-diff* [*simp*]: *count* $(M - N)\ a = \textit{count}\ M\ a - \textit{count}\ N\ a$
⟨*proof*⟩

lemma *count-MCollect* [*simp*]:
count $\{\#\ x{:}\#M.\ P\ x\ \#\}\ a = (\textit{if}\ P\ a\ \textit{then count}\ M\ a\ \textit{else}\ 0)$
⟨*proof*⟩

46.2.4 Set of elements

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*]: (*set-of M* = {}) = (*M* = {#})
⟨*proof*⟩

lemma *mem-set-of-iff* [*simp*]: (*x* ∈ *set-of M*) = (*x* :# *M*)
⟨*proof*⟩

lemma *set-of-MCollect* [*simp*]: *set-of* {# *x*:#*M*. *P x* #} = *set-of M* ∩ {*x*. *P x*}
⟨*proof*⟩

46.2.5 Size

lemma *size-empty* [*simp*]: *size* {#} = *0*
⟨*proof*⟩

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
⟨*proof*⟩

lemma *finite-set-of* [*iff*]: *finite* (*set-of M*)
⟨*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*⟩

46.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: (*M* = *N*) = (∀ *a*. *count M a* = *count N a*)
⟨*proof*⟩

lemma *single-not-empty* [*simp*]: {#*a*#} ≠ {#} ∧ {#} ≠ {#*a*#}
⟨*proof*⟩

lemma *single-eq-single* [*simp*]: ({#*a*#} = {#*b*#}) = (*a* = *b*)
⟨*proof*⟩

lemma *union-eq-empty* [*iff*]: (*M* + *N* = {#}) = (*M* = {#} ∧ *N* = {#})
⟨*proof*⟩

lemma *empty-eq-union* [*iff*]: $(\{\#\} = M + N) = (M = \{\#\} \wedge N = \{\#\})$
⟨*proof*⟩

lemma *union-right-cancel* [*simp*]: $(M + K = N + K) = (M = (N::'a\ multiset))$
⟨*proof*⟩

lemma *union-left-cancel* [*simp*]: $(K + M = K + N) = (M = (N::'a\ multiset))$
⟨*proof*⟩

lemma *union-is-single*:
$(M + N = \{\#a\#\}) = (M = \{\#a\#\} \wedge N=\{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\})$
⟨*proof*⟩

lemma *single-is-union*:
$(\{\#a\#\} = M + N) \longleftrightarrow (\{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N)$
⟨*proof*⟩

lemma *add-eq-conv-diff*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee M = N - \{\#a\#\} + \{\#b\#\} \wedge N = M - \{\#b\#\} + \{\#a\#\})$
⟨*proof*⟩

declare *Rep-multiset-inject* [*symmetric*, *simp del*]

instance *multiset* :: (*type*) *cancel-ab-semigroup-add*
⟨*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 *multi-union-self-other-eq*:
$(A::'a\ multiset) + X = A + Y \Longrightarrow X = Y$
⟨*proof*⟩

lemma *multi-self-add-other-not-self* [*simp*]: $(A = A + \{\#x\#\}) = False$
⟨*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*⟩

lemma *empty-multiset-count*:
$(\forall x.\ count\ A\ x = 0) = (A = \{\#\})$
⟨*proof*⟩

46.2.7 Intersection

lemma *multiset-inter-count*:
$count\ (A\ \#\cap\ B)\ x = min\ (count\ A\ x)\ (count\ B\ x)$
⟨*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 *multiset-inter-single*: $a \neq b \Longrightarrow \{\#a\#\}\ \#\cap\ \{\#b\#\} = \{\#\}$
⟨*proof*⟩

lemma *multiset-union-diff-commute*: $B\ \#\cap\ C = \{\#\} \Longrightarrow A + B - C = A - C + B$
⟨*proof*⟩

46.2.8 Comprehension (filter)

lemma *MCollect-empty* [*simp*]: $MCollect\ \{\#\}\ P = \{\#\}$
⟨*proof*⟩

lemma *MCollect-single* [*simp*]:
$MCollect\ \{\#x\#\}\ P = (if\ P\ x\ then\ \{\#x\#\}\ else\ \{\#\})$
⟨*proof*⟩

lemma *MCollect-union* [*simp*]:
$MCollect\ (M+N)\ f = MCollect\ M\ f\ +\ MCollect\ N\ f$

⟨*proof*⟩

46.3 Induction and case splits

lemma *setsum-decr*:
finite F ==> (0::nat) < f a ==>
setsum (f (a := f a − 1)) F = (if a∈F then setsum f F − 1 else setsum f F)
⟨*proof*⟩

lemma *rep-multiset-induct-aux*:
assumes *1*: *P (λa. (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 ∈ multiset ==> P (λa. 0) ==>
(!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))) ==> P f
⟨*proof*⟩

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
assumes *empty*: *P {#}*
and *add*: *!!M x. P M ==> P (M + {#x#})*
shows *P M*
⟨*proof*⟩

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 *multiset-partition*: $M = \{\#\ x{:}\#M.\ P\ x\ \#\} + \{\#\ x{:}\#M.\ \neg\ P\ x\ \#\}$
⟨*proof*⟩

declare *multiset-typedef* [*simp del*]

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
⟨*proof*⟩

46.4 Orderings

46.4.1 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**
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*⟩

46.4.2 Closure-free presentation

lemma *diff-union-single-conv*: $a :\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

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 *elem-imp-eq-diff-union*: $a :\# M \Longrightarrow M = M - \{\#a\#\} + \{\#a\#\}$
⟨*proof*⟩

lemma *size-eq-Suc-imp-eq-union*: *size M* = *Suc n* $\Longrightarrow \exists a\ N.\ M = N + \{\#a\#\}$
⟨*proof*⟩

lemma *one-step-implies-mult-aux*:
trans r $\Longrightarrow$

$\forall I\ J\ K.\ (size\ J = n \wedge J \neq \{\#\} \wedge (\forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r))$
$\quad --> (I + K, I + J) \in mult\ r$
⟨*proof*⟩

lemma *one-step-implies-mult*:
$trans\ r ==> J \neq \{\#\} ==> \forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r$
$\quad ==> (I + K, I + J) \in mult\ r$
⟨*proof*⟩

46.4.3 Partial-order properties

instantiation *multiset* :: (*order*) *order*
begin

definition *less-multiset-def* [*code del*]:
$M' < M \longleftrightarrow (M', M) \in mult\ \{(x', x).\ x' < x\}$

definition *le-multiset-def* [*code del*]:
$M' <= M \longleftrightarrow M' = M \vee M' < (M::'a\ multiset)$

lemma *trans-base-order*: $trans\ \{(x', x).\ x' < (x::'a::order)\}$
⟨*proof*⟩

Irreflexivity.

lemma *mult-irrefl-aux*:
$finite\ A ==> (\forall x \in A.\ \exists y \in A.\ x < (y::'a::order)) \Longrightarrow A = \{\}$
⟨*proof*⟩

lemma *mult-less-not-refl*: $\neg\ M < (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *mult-less-irrefl* [*elim*!]: $M < (M::'a::order\ multiset) ==> R$
⟨*proof*⟩

Transitivity.

theorem *mult-less-trans*: $K < M ==> M < N ==> K < (N::'a::order\ multiset)$
⟨*proof*⟩

Asymmetry.

theorem *mult-less-not-sym*: $M < N ==> \neg\ N < (M::'a::order\ multiset)$
⟨*proof*⟩

theorem *mult-less-asym*:
$M < N ==> (\neg\ P ==> N < (M::'a::order\ multiset)) ==> P$
⟨*proof*⟩

theorem *mult-le-refl* [*iff*]: $M <= (M::'a::order\ multiset)$
⟨*proof*⟩

Anti-symmetry.

theorem *mult-le-antisym*:
$M <= N ==> N <= M ==> M = (N::'a::order\ multiset)$
⟨*proof*⟩

Transitivity.

theorem *mult-le-trans*:
$K <= M ==> M <= N ==> K <= (N::'a::order\ multiset)$
⟨*proof*⟩

theorem *mult-less-le*: $(M < N) = (M <= N \wedge M \neq (N::'a::order\ multiset))$
⟨*proof*⟩

instance ⟨*proof*⟩

end

46.4.4 Monotonicity of multiset union

lemma *mult1-union*:
$(B, D) \in mult1\ r ==> trans\ r ==> (C + B, C + D) \in mult1\ r$
⟨*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*⟩

lemma *union-le-mono*:
$A <= C ==> B <= D ==> A + B <= C + (D::'a::order\ multiset)$
⟨*proof*⟩

lemma *empty-leI* [*iff*]: $\{\#\} <= (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-upper1*: $A <= A + (B::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-upper2*: $B <= A + (B::'a::order\ multiset)$
⟨*proof*⟩

instance *multiset* :: (*order*) *pordered-ab-semigroup-add*
⟨*proof*⟩

46.5 Link with lists

primrec *multiset-of* :: *'a list* ⇒ *'a multiset* **where**
multiset-of [] = {#} |
multiset-of (*a* # *x*) = *multiset-of x* + {# *a* #}

lemma *multiset-of-zero-iff*[*simp*]: (*multiset-of x* = {#}) = (*x* = [])
⟨*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*: *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*⟩

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

lemma *properties-for-sort*:
multiset-of ys = *multiset-of xs* ⟹ *sorted ys* ⟹ *sort xs* = *ys*
⟨*proof*⟩

46.6 Pointwise ordering induced by count

definition *mset-le* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ≤# *50*) **where**
[*code del*]: *A* ≤# *B* ⟷ (∀ *a*. *count A a* ≤ *count B a*)

definition *mset-less* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** <# *50*) **where**
[*code del*]: *A* <# *B* ⟷ *A* ≤# *B* ∧ *A* ≠ *B*

notation *mset-le* (**infix** ⊆# *50*)
notation *mset-less* (**infix** ⊂# *50*)

lemma *mset-le-refl*[*simp*]: *A* ≤# *A*
⟨*proof*⟩

lemma *mset-le-trans*: *A* ≤# *B* ⟹ *B* ≤# *C* ⟹ *A* ≤# *C*
⟨*proof*⟩

lemma *mset-le-antisym*: *A* ≤# *B* ⟹ *B* ≤# *A* ⟹ *A* = *B*
⟨*proof*⟩

lemma *mset-le-exists-conv*: (*A* ≤# *B*) = (∃ *C*. *B* = *A* + *C*)
⟨*proof*⟩

lemma *mset-le-mono-add-right-cancel*[*simp*]: (*A* + *C* ≤# *B* + *C*) = (*A* ≤# *B*)
⟨*proof*⟩

lemma *mset-le-mono-add-left-cancel*[*simp*]: (*C* + *A* ≤# *C* + *B*) = (*A* ≤# *B*)
⟨*proof*⟩

lemma *mset-le-mono-add*: $[\![A \leq\# B; C \leq\# D]\!] \Longrightarrow A + C \leq\# B + D$
⟨*proof*⟩

lemma *mset-le-add-left*[*simp*]: $A \leq\# A + B$
⟨*proof*⟩

lemma *mset-le-add-right*[*simp*]: $B \leq\# A + B$
⟨*proof*⟩

lemma *mset-le-single*: $a :\# B \Longrightarrow \{\#a\#\} \leq\# B$
⟨*proof*⟩

lemma *multiset-diff-union-assoc*: $C \leq\# B \Longrightarrow A + B - C = A + (B - C)$
⟨*proof*⟩

lemma *mset-le-multiset-union-diff-commute*:
assumes $B \leq\# A$
shows $A - B + C = A + C - B$
⟨*proof*⟩

lemma *multiset-of-remdups-le*: *multiset-of* (*remdups xs*) $\leq\#$ *multiset-of xs*
⟨*proof*⟩

lemma *multiset-of-update*:
$i <$ *length ls* $\Longrightarrow$ *multiset-of* (*ls*[$i :=$ *v*]) = *multiset-of ls* $- \{\#$*ls* ! *i*$\#\} + \{\#$*v*$\#\}$
⟨*proof*⟩

lemma *multiset-of-swap*:
$i <$ *length ls* $\Longrightarrow$ $j <$ *length ls* $\Longrightarrow$
multiset-of (*ls*[*j* := *ls* ! *i*, *i* := *ls* ! *j*]) = *multiset-of ls*
⟨*proof*⟩

interpretation *mset-order*: *order op* $\leq\#$ *op* $<\#$
⟨*proof*⟩

interpretation *mset-order-cancel-semigroup*:
pordered-cancel-ab-semigroup-add op + *op* $\leq\#$ *op* $<\#$
⟨*proof*⟩

interpretation *mset-order-semigroup-cancel*:
pordered-ab-semigroup-add-imp-le op + *op* $\leq\#$ *op* $<\#$
⟨*proof*⟩

lemma *mset-lessD*: $A \subset\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
⟨*proof*⟩

lemma *mset-leD*: $A \subseteq\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
⟨*proof*⟩

lemma *mset-less-insertD*: $(A + \{\#x\#\} \subset\# B) \Longrightarrow (x \in\# B \wedge A \subset\# B)$
⟨*proof*⟩

lemma *mset-le-insertD*: $(A + \{\#x\#\} \subseteq\# B) \Longrightarrow (x \in\# B \wedge A \subseteq\# B)$
⟨*proof*⟩

lemma *mset-less-of-empty*[*simp*]: $A \subset\# \{\#\} = \mathit{False}$
⟨*proof*⟩

lemma *multi-psub-of-add-self*[*simp*]: $A \subset\# A + \{\#x\#\}$
⟨*proof*⟩

lemma *multi-psub-self*[*simp*]: $A \subset\# A = \mathit{False}$
⟨*proof*⟩

lemma *mset-less-add-bothsides*:
$T + \{\#x\#\} \subset\# S + \{\#x\#\} \Longrightarrow T \subset\# S$
⟨*proof*⟩

lemma *mset-less-empty-nonempty*: $(\{\#\} \subset\# S) = (S \neq \{\#\})$
⟨*proof*⟩

lemma *mset-less-size*: $A \subset\# B \Longrightarrow \mathit{size}\ A < \mathit{size}\ B$
⟨*proof*⟩

lemmas *mset-less-trans* = *mset-order.less-trans*

lemma *mset-less-diff-self*: $c \in\# B \Longrightarrow B - \{\#c\#\} \subset\# B$
⟨*proof*⟩

46.7 Strong induction and subset induction for multisets

Well-foundedness of proper subset operator:

proper multiset subset

definition
mset-less-rel :: (*'a multiset* × *'a multiset*) *set* **where**
mset-less-rel = $\{(A,B).\ A \subset\# B\}$

lemma *multiset-add-sub-el-shuffle*:
assumes $c \in\# B$ **and** $b \neq c$
shows $B - \{\#c\#\} + \{\#b\#\} = B + \{\#b\#\} - \{\#c\#\}$
⟨*proof*⟩

lemma *wf-mset-less-rel*: *wf mset-less-rel*
⟨*proof*⟩

The induction rules:

lemma *full-multiset-induct* [*case-names less*]:

assumes *ih*: $\bigwedge B.\ \forall A.\ A \subset\# B \longrightarrow P\ A \Longrightarrow P\ B$
shows $P\ B$
⟨*proof*⟩

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \subseteq\# A$
and *empty*: $P\ \{\#\}$
and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (F + \{\#a\#\})$
shows $P\ F$
⟨*proof*⟩

A consequence: Extensionality.

lemma *multi-count-eq*: $(\forall x.\ count\ A\ x = count\ B\ x) = (A = B)$
⟨*proof*⟩

lemmas *multi-count-ext* = *multi-count-eq* [*THEN iffD1*, *rule-format*]

46.8 The fold combinator

The intended behaviour is *fold-mset* $f\ z\ \{\#x_1, ..., x_n\#\} = f\ x_1\ (\ldots\ (f\ x_n\ z)\ldots)$ if f is associative-commutative.

The graph of *fold-mset*, z: the start element, f: folding function, A: the multiset, y: the result.

inductive
fold-msetG :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b \Rightarrow bool$
for $f :: 'a \Rightarrow 'b \Rightarrow 'b$
and $z :: 'b$
where
emptyI [*intro*]: *fold-msetG f z* $\{\#\}$ *z*
| *insertI* [*intro*]: *fold-msetG f z A y* $\Longrightarrow$ *fold-msetG f z* $(A + \{\#x\#\})$ $(f\ x\ y)$

inductive-cases *empty-fold-msetGE* [*elim*!]: *fold-msetG f z* $\{\#\}$ *x*
inductive-cases *insert-fold-msetGE*: *fold-msetG f z* $(A + \{\#\})$ *y*

definition
fold-mset :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b$ **where**
fold-mset f z A = (*THE x*. *fold-msetG f z A x*)

lemma *Diff1-fold-msetG*:
fold-msetG f z $(A - \{\#x\#\})$ *y* $\Longrightarrow$ $x \in\# A \Longrightarrow$ *fold-msetG f z A* $(f\ x\ y)$
⟨*proof*⟩

lemma *fold-msetG-nonempty*: $\exists x.$ *fold-msetG f z A x*
⟨*proof*⟩

lemma *fold-mset-empty*[*simp*]: *fold-mset f z* $\{\#\} = z$
⟨*proof*⟩

locale *left-commutative* =
fixes $f :: 'a => 'b => 'b$
assumes *left-commute*: $f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$
begin

lemma *fold-msetG-determ*:
$fold\text{-}msetG\ f\ z\ A\ x \Longrightarrow fold\text{-}msetG\ f\ z\ A\ y \Longrightarrow y = x$
⟨*proof*⟩

lemma *fold-mset-insert-aux*:
$(fold\text{-}msetG\ f\ z\ (A + \{\#x\#\})\ v) =$
$(\exists y.\ fold\text{-}msetG\ f\ z\ A\ y \wedge v = f\ x\ y)$
⟨*proof*⟩

lemma *fold-mset-equality*: $fold\text{-}msetG\ f\ z\ A\ y \Longrightarrow fold\text{-}mset\ f\ z\ A = y$
⟨*proof*⟩

lemma *fold-mset-insert*:
$fold\text{-}mset\ f\ z\ (A + \{\#x\#\}) = f\ x\ (fold\text{-}mset\ f\ z\ A)$
⟨*proof*⟩

lemma *fold-mset-insert-idem*:
$fold\text{-}mset\ f\ z\ (A + \{\#a\#\}) = f\ a\ (fold\text{-}mset\ f\ z\ A)$
⟨*proof*⟩

lemma *fold-mset-commute*: $f\ x\ (fold\text{-}mset\ f\ z\ A) = fold\text{-}mset\ f\ (f\ x\ z)\ A$
⟨*proof*⟩

lemma *fold-mset-single* [*simp*]: $fold\text{-}mset\ f\ z\ \{\#x\#\} = f\ x\ z$
⟨*proof*⟩

lemma *fold-mset-union* [*simp*]:
$fold\text{-}mset\ f\ z\ (A+B) = fold\text{-}mset\ f\ (fold\text{-}mset\ f\ z\ A)\ B$
⟨*proof*⟩

lemma *fold-mset-fusion*:
assumes *left-commutative g*
shows $(\bigwedge x\ y.\ h\ (g\ x\ y) = f\ x\ (h\ y)) \Longrightarrow h\ (fold\text{-}mset\ g\ w\ A) = fold\text{-}mset\ f\ (h\ w)\ A$ (**is** *PROP ?P*)
⟨*proof*⟩

lemma *fold-mset-rec*:
assumes $a \in\# A$
shows $fold\text{-}mset\ f\ z\ A = f\ a\ (fold\text{-}mset\ f\ z\ (A - \{\#a\#\}))$
⟨*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.

46.9 Image

definition [*code del*]:
image-mset f = *fold-mset* (*op* + *o single o f*) {#}

interpretation *image-left-comm*: *left-commutative op* + *o single o 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*#}#}.

46.10 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in\# XS \Longrightarrow x \in\# \{\# y \#\} + XS$
and *multi-member-this*: $x \in\# \{\# x \#\} + XS$
and *multi-member-last*: $x \in\# \{\# x \#\}$
⟨*proof*⟩

definition *ms-strict* = *mult pair-less*
definition [*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*) ∈ *pair-leq* ⟹ *pw-leq* {#*x*#} {#*y*#}
⟨*proof*⟩

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows ∃ *A B Z*. *X* = *A* + *Z* ∧ *Y* = *B* + *Z* ∧ ((*set-of A*, *set-of B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#}))
⟨*proof*⟩

lemma
assumes *pwleq*: *pw-leq Z Z′*
shows *ms-strictI*: (*set-of A*, *set-of B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z′* + *B*) ∈ *ms-strict*
and *ms-weakI1*: (*set-of A*, *set-of B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z′* + *B*) ∈ *ms-weak*
and *ms-weakI2*: (*Z* + {#}, *Z′* + {#}) ∈ *ms-weak*
⟨*proof*⟩

lemma *empty-idemp*: {#} + *x* = *x x* + {#} = *x*
and *nonempty-plus*: {# *x* #} + *rs* ≠ {#}
and *nonempty-single*: {# *x* #} ≠ {#}

⟨*proof*⟩

⟨*ML*⟩

end

47 Nat-Infinity: Natural numbers with infinity

theory *Nat-Infinity*
imports *Main*
begin

47.1 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 (∞)

47.2 Constructors and numbers

instantiation *inat* :: {*zero*, *one*, *number*}
begin

definition
0 = *Fin 0*

definition
[*code inline*]: *1* = *Fin 1*

definition
[*code inline*, *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 ≠ ∞*
⟨*proof*⟩

lemma *zero-inat-eq* [*simp*]:
number-of k = (0::inat) ⟷ number-of k = (0::nat)
(0::inat) = number-of k ⟷ number-of k = (0::nat)
⟨*proof*⟩

lemma *one-inat-eq* [*simp*]:
number-of k = (1::inat) ⟷ number-of k = (1::nat)
(1::inat) = number-of k ⟷ number-of k = (1::nat)
⟨*proof*⟩

lemma *zero-one-inat-neq* [*simp*]:
¬ 0 = (1::inat)
¬ 1 = (0::inat)
⟨*proof*⟩

lemma *Infty-ne-i1* [*simp*]: *∞ ≠ 1*
⟨*proof*⟩

lemma *i1-ne-Infty* [*simp*]: *1 ≠ ∞*
⟨*proof*⟩

lemma *Infty-ne-number* [*simp*]: *∞ ≠ number-of k*
⟨*proof*⟩

lemma *number-ne-Infty* [*simp*]: *number-of k ≠ ∞*
⟨*proof*⟩

lemma *iSuc-Fin*: *iSuc (Fin n) = Fin (Suc n)*
⟨*proof*⟩

lemma *iSuc-number-of*: *iSuc (number-of k) = Fin (Suc (number-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* $\longleftrightarrow$ (*number-of k* :: *nat*) = *number-of l*
⟨*proof*⟩

47.3 Addition

instantiation *inat* :: *comm-monoid-add*
begin

definition
[*code del*]: *m* + *n* = (*case m of* $\infty \Rightarrow \infty$ | *Fin m* $\Rightarrow$ (*case n of* $\infty \Rightarrow \infty$ | *Fin n* $\Rightarrow$ *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*⟩

47.4 Multiplication

instantiation *inat* :: *comm-semiring-1*
begin

definition
times-inat-def [*code del*]:
*m * n = (case m of ∞ ⇒ if n = 0 then 0 else ∞ | Fin m ⇒*
*(case n of ∞ ⇒ if m = 0 then 0 else ∞ | Fin n ⇒ Fin (m * n)))*

lemma *times-inat-simps* [*simp*, *code*]:
*Fin m * Fin n = Fin (m * n)*
*∞ * ∞ = ∞*
*∞ * Fin n = (if n = 0 then 0 else ∞)*
*Fin m * ∞ = (if m = 0 then 0 else ∞)*
⟨*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-nat n = Fin n*
⟨*proof*⟩

instance *inat* :: *semiring-char-0*
⟨*proof*⟩

47.5 Ordering

instantiation *inat* :: *ordered-ab-semigroup-add*
begin

definition

[*code del*]: $m \leq n = (\mathit{case}\ n\ \mathit{of}\ \mathit{Fin}\ n1 \Rightarrow (\mathit{case}\ m\ \mathit{of}\ \mathit{Fin}\ m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow \mathit{False}) \mid \infty \Rightarrow \mathit{True})$

definition
[*code del*]: $m < n = (\mathit{case}\ m\ \mathit{of}\ \mathit{Fin}\ m1 \Rightarrow (\mathit{case}\ n\ \mathit{of}\ \mathit{Fin}\ n1 \Rightarrow m1 < n1 \mid \infty \Rightarrow \mathit{True}) \mid \infty \Rightarrow \mathit{False})$

lemma *inat-ord-simps* [*simp*]:
$\mathit{Fin}\ m \leq \mathit{Fin}\ n \longleftrightarrow m \leq n$
$\mathit{Fin}\ m < \mathit{Fin}\ n \longleftrightarrow m < n$
$q \leq \infty$
$q < \infty \longleftrightarrow q \neq \infty$
$\infty \leq q \longleftrightarrow q = \infty$
$\infty < q \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *inat-ord-code* [*code*]:
$\mathit{Fin}\ m \leq \mathit{Fin}\ n \longleftrightarrow m \leq n$
$\mathit{Fin}\ m < \mathit{Fin}\ n \longleftrightarrow m < n$
$q \leq \infty \longleftrightarrow \mathit{True}$
$\mathit{Fin}\ m < \infty \longleftrightarrow \mathit{True}$
$\infty \leq \mathit{Fin}\ n \longleftrightarrow \mathit{False}$
$\infty < q \longleftrightarrow \mathit{False}$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *inat* :: *pordered-comm-semiring*
⟨*proof*⟩

lemma *inat-ord-number* [*simp*]:
$(\mathit{number\text{-}of}\ m :: \mathit{inat}) \leq \mathit{number\text{-}of}\ n \longleftrightarrow (\mathit{number\text{-}of}\ m :: \mathit{nat}) \leq \mathit{number\text{-}of}\ n$
$(\mathit{number\text{-}of}\ m :: \mathit{inat}) < \mathit{number\text{-}of}\ n \longleftrightarrow (\mathit{number\text{-}of}\ m :: \mathit{nat}) < \mathit{number\text{-}of}\ n$
⟨*proof*⟩

lemma *i0-lb* [*simp*]: $(0::\mathit{inat}) \leq n$
⟨*proof*⟩

lemma *i0-neq* [*simp*]: $n \leq (0::\mathit{inat}) \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *Infty-ileE* [*elim!*]: $\infty \leq \mathit{Fin}\ m \Longrightarrow R$
⟨*proof*⟩

lemma *Infty-ilessE* [*elim!*]: $\infty < \mathit{Fin}\ m \Longrightarrow R$

⟨*proof*⟩

lemma *not-ilessi0* [*simp*]: ¬ *n* < (*0*::*inat*)
⟨*proof*⟩

lemma *i0-eq* [*simp*]: (*0*::*inat*) < *n* ⟷ *n* ≠ *0*
⟨*proof*⟩

lemma *iSuc-ile-mono* [*simp*]: *iSuc n* ≤ *iSuc m* ⟷ *n* ≤ *m*
⟨*proof*⟩

lemma *iSuc-mono* [*simp*]: *iSuc n* < *iSuc m* ⟷ *n* < *m*
⟨*proof*⟩

lemma *ile-iSuc* [*simp*]: *n* ≤ *iSuc n*
⟨*proof*⟩

lemma *not-iSuc-ilei0* [*simp*]: ¬ *iSuc n* ≤ *0*
⟨*proof*⟩

lemma *i0-iless-iSuc* [*simp*]: *0* < *iSuc n*
⟨*proof*⟩

lemma *ileI1*: *m* < *n* ⟹ *iSuc m* ≤ *n*
⟨*proof*⟩

lemma *Suc-ile-eq*: *Fin* (*Suc m*) ≤ *n* ⟷ *Fin m* < *n*
⟨*proof*⟩

lemma *iless-Suc-eq* [*simp*]: *Fin m* < *iSuc n* ⟷ *Fin m* ≤ *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 ∞ = *q*
min ∞ *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 ∞ = ∞
max ∞ *q* = ∞
⟨*proof*⟩

lemma *Fin-ile*: *n* ≤ *Fin m* ⟹ ∃ *k*. *n* = *Fin k*

⟨*proof*⟩

lemma *Fin-iless*: $n < Fin\ m \Longrightarrow \exists k.\ n = Fin\ k$
⟨*proof*⟩

lemma *chain-incr*: $\forall i.\ \exists j.\ Y\ i < Y\ j$ ==> $\exists j.\ Fin\ k < Y\ j$
⟨*proof*⟩

instantiation *inat* :: {*bot*, *top*}
begin

definition *bot-inat* :: *inat* **where**
bot-inat = *0*

definition *top-inat* :: *inat* **where**
top-inat = ∞

instance ⟨*proof*⟩

end

47.6 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*: !!$n.\ \forall m$::*inat*. $m < n$ −−> $P\ m$ ==> $P\ n$ **shows** $P\ n$
⟨*proof*⟩

instance *inat* :: *wellorder*
⟨*proof*⟩

47.7 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

48 Nested-Environment: Nested environments

theory *Nested-Environment*
imports *Main*
begin

Consider a partial function $e :: 'a \Rightarrow 'b\ option$; 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 $('a, 'b, 'c)\ env =$
  $Val\ 'a$
| $Env\ 'b\ \ 'c \Rightarrow ('a, 'b, 'c)\ env\ option$

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.

48.1 The lookup operation

Lookup in nested environments works by following a given path of index elements, leading to an optional result (a terminal value or nested environment). A *defined position* within a nested environment is one where *lookup* at its path does not yield *None*.

consts
  $lookup :: ('a, 'b, 'c)\ env \Rightarrow 'c\ list \Rightarrow ('a, 'b, 'c)\ env\ option$
  $lookup\text{-}option :: ('a, 'b, 'c)\ env\ option \Rightarrow 'c\ list \Rightarrow ('a, 'b, 'c)\ env\ option$

primrec $(lookup)$
  $lookup\ (Val\ a)\ xs = (if\ xs = []\ then\ Some\ (Val\ a)\ else\ None)$
  $lookup\ (Env\ b\ es)\ xs =$
    $(case\ xs\ of$
      $[] \Rightarrow Some\ (Env\ b\ es)$
    $| y \# ys \Rightarrow lookup\text{-}option\ (es\ y)\ ys)$
  $lookup\text{-}option\ None\ xs = None$
  $lookup\text{-}option\ (Some\ e)\ xs = lookup\ e\ xs$

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.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 $\exists\, b'\ es'\ env'$.
lookup env xs = Some (Env b' es') $\wedge$
es' y = Some env' $\wedge$
lookup env' ys = Some e
⟨*proof*⟩

48.2 The update operation

Update at a certain position in a nested environment may either delete an existing entry, or overwrite an existing one. Note that update at undefined positions is simple absorbed, i.e. the environment is left unchanged.

consts
update :: *'c list => ('a, 'b, 'c) env option*
=> ('a, 'b, 'c) env => ('a, 'b, 'c) env
update-option :: *'c list => ('a, 'b, 'c) env option*
=> ('a, 'b, 'c) env option => ('a, 'b, 'c) env option

primrec (*update*)
update xs opt (Val a) =
(if xs = [] then (case opt of None => Val a | Some e => e)
else Val a)
update xs opt (Env b es) =
(case xs of
[] => (case opt of None => Env b es | Some e => e)
| y # ys => Env b (es (y := update-option ys opt (es y))))
update-option xs opt None =
(if xs = [] then opt else None)
update-option xs opt (Some e) =
(if xs = [] then opt else Some (update xs opt e))

hide *const update-option*

The characteristic cases of *update* are expressed by the following equalities.

theorem *update-nil-none*: *update [] None env = env*
⟨*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.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 ≠ (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-Eval.term-of* :: (*'a::{term-of, type}*, *'b::{term-of, type}*, *'c::{term-of, type}*)
env ⇒ term) = *Code-Eval.term-of* ⟨*proof*⟩

end

⟨*ML*⟩

49 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

50 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*⟩

50.1 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*⟩

50.2 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*⟩

50.3 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*⟩

50.4 Removing elements

consts
remove :: *'a => 'a list => 'a list*
primrec
remove x [] = []
remove x (y # ys) = (if x = y then ys else y # remove x ys)

lemma *perm-remove*: *x ∈ set ys ==> ys <~~> x # remove x ys*
⟨*proof*⟩

lemma *remove-commute*: *remove x (remove y l) = remove y (remove x l)*
⟨*proof*⟩

lemma *multiset-of-remove* [*simp*]:
multiset-of (remove a x) = multiset-of x - {#a#}
⟨*proof*⟩

Congruence rule

lemma *perm-remove-perm*: *xs <~~> ys ==> remove z xs <~~> remove z ys*
⟨*proof*⟩

lemma *remove-hd* [*simp*]: *remove z (z # xs) = xs*
⟨*proof*⟩

lemma *cons-perm-imp-perm*: *z # xs <~~> z # ys ==> xs <~~> ys*
⟨*proof*⟩

lemma *cons-perm-eq* [*iff*]: *(z#xs <~~> z#ys) = (xs <~~> ys)*
⟨*proof*⟩

lemma *append-perm-imp-perm*: *zs @ xs <~~> zs @ ys ==> xs <~~> ys*
⟨*proof*⟩

lemma *perm-append1-eq* [*iff*]: *(zs @ xs <~~> zs @ ys) = (xs <~~> ys)*
⟨*proof*⟩

lemma *perm-append2-eq* [*iff*]: *(xs @ zs <~~> ys @ zs) = (xs <~~> ys)*
⟨*proof*⟩

lemma *multiset-of-eq-perm*: *(multiset-of xs = multiset-of ys) = (xs <~~> ys)*
⟨*proof*⟩

lemma *multiset-of-le-perm-append*:
(multiset-of xs ≤# multiset-of ys) = (∃ zs. xs @ zs <~~> ys)
⟨*proof*⟩

lemma *perm-set-eq*: *xs <~~> ys ==> set xs = set ys*
⟨*proof*⟩

lemma *perm-distinct-iff*: *xs* <~~> *ys* ==> *distinct xs* = *distinct ys*
⟨*proof*⟩

lemma *eq-set-perm-remdups*: *set xs* = *set ys* ==> *remdups xs* <~~> *remdups ys*
⟨*proof*⟩

lemma *perm-remdups-iff-eq-set*: *remdups x* <~~> *remdups y* = (*set x* = *set y*)
⟨*proof*⟩

end

51 Primes: Primality on nat

theory *Primes*
imports *Complex-Main*
begin

definition
coprime :: *nat* => *nat* => *bool* **where**
coprime m n ⟷ *gcd m n* = *1*

definition
prime :: *nat* ⇒ *bool* **where**
[*code del*]: *prime p* ⟷ (*1* < *p* ∧ (∀ *m*. *m dvd p* −−> *m* = *1* ∨ *m* = *p*))

lemma *two-is-prime*: *prime 2*
⟨*proof*⟩

lemma *prime-imp-relprime*: *prime p* ==> ¬ *p dvd n* ==> *gcd p n* = *1*
⟨*proof*⟩

This theorem leads immediately to a proof of the uniqueness of factorization. If p divides a product of primes then it is one of those primes.

lemma *prime-dvd-mult*: *prime p* ==> *p dvd m* ∗ *n* ==> *p dvd m* ∨ *p dvd n*
⟨*proof*⟩

lemma *prime-dvd-square*: *prime p* ==> *p dvd m*^*Suc* (*Suc 0*) ==> *p dvd m*
⟨*proof*⟩

lemma *prime-dvd-power-two*: *prime p* ==> *p dvd* m^2 ==> *p dvd m*
⟨*proof*⟩

lemma *exp-eq-1*:(*x*::*nat*) ^*n* = *1* ⟷ *x* = *1* ∨ *n* = *0*
⟨*proof*⟩

lemma *exp-mono-lt*: (*x*::*nat*) ^ (*Suc n*) < *y* ^ (*Suc n*) ⟷ *x* < *y*

⟨*proof*⟩

lemma *exp-mono-le*: (*x*::*nat*) ^ (*Suc n*) ≤ *y* ^ (*Suc n*) ⟷ *x* ≤ *y*
⟨*proof*⟩

lemma *exp-mono-eq*: (*x*::*nat*) ^ *Suc n* = *y* ^ *Suc n* ⟷ *x* = *y*
⟨*proof*⟩

lemma *even-square*: **assumes** *e*: *even* (*n*::*nat*) **shows** ∃ *x*. *n* ^ *2* = *4*∗*x*
⟨*proof*⟩

lemma *odd-square*: **assumes** *e*: *odd* (*n*::*nat*) **shows** ∃ *x*. *n* ^ *2* = *4*∗*x* + *1*
⟨*proof*⟩

lemma *diff-square*: (*x*::*nat*)^*2* − *y*^*2* = (*x*+*y*)∗(*x* − *y*)
⟨*proof*⟩

Elementary theory of divisibility

lemma *divides-ge*: (*a*::*nat*) *dvd b* ⟹ *b* = *0* ∨ *a* ≤ *b* ⟨*proof*⟩
lemma *divides-antisym*: (*x*::*nat*) *dvd y* ∧ *y dvd x* ⟷ *x* = *y*
⟨*proof*⟩

lemma *divides-add-revr*: **assumes** *da*: (*d*::*nat*) *dvd a* **and** *dab*:*d dvd* (*a* + *b*)
shows *d dvd b*
⟨*proof*⟩

declare *nat-mult-dvd-cancel-disj*[*presburger*]
lemma *nat-mult-dvd-cancel-disj'*[*presburger*]:
(*m*::*nat*)∗*k dvd n*∗*k* ⟷ *k* = *0* ∨ *m dvd n* ⟨*proof*⟩

lemma *divides-mul-l*: (*a*::*nat*) *dvd b* ==> (*c* ∗ *a*) *dvd* (*c* ∗ *b*)
⟨*proof*⟩

lemma *divides-mul-r*: (*a*::*nat*) *dvd b* ==> (*a* ∗ *c*) *dvd* (*b* ∗ *c*) ⟨*proof*⟩
lemma *divides-cases*: (*n*::*nat*) *dvd m* ==> *m* = *0* ∨ *m* = *n* ∨ *2* ∗ *n* <= *m*
⟨*proof*⟩

lemma *divides-div-not*: (*x*::*nat*) = (*q* ∗ *n*) + *r* ⟹ *0* < *r* ⟹ *r* < *n* ==> ~(*n dvd x*)
⟨*proof*⟩
lemma *divides-exp*: (*x*::*nat*) *dvd y* ==> *x* ^ *n dvd y* ^ *n*
⟨*proof*⟩

lemma *divides-exp2*: *n* ≠ *0* ⟹ (*x*::*nat*) ^ *n dvd y* ⟹ *x dvd y*
⟨*proof*⟩

fun *fact* :: *nat* ⇒ *nat* **where**
fact 0 = *1*

| *fact* (*Suc n*) = *Suc n* ∗ *fact n*

lemma *fact-lt*: *0* < *fact n* ⟨*proof*⟩
lemma *fact-le*: *fact n* ≥ *1* ⟨*proof*⟩
lemma *fact-mono*: **assumes** *le*: *m* ≤ *n* **shows** *fact m* ≤ *fact n*
⟨*proof*⟩

lemma *divides-fact*: *1* <= *p* ⟹ *p* <= *n* ==> *p dvd fact n*
⟨*proof*⟩

declare *dvd-triv-left*[*presburger*]
declare *dvd-triv-right*[*presburger*]
lemma *divides-rexp*:
x dvd y ⟹ (*x*::*nat*) *dvd* (*y*^(*Suc n*)) ⟨*proof*⟩

Coprimality

lemma *coprime*: *coprime a b* ⟷ (∀ *d*. *d dvd a* ∧ *d dvd b* ⟷ *d* = *1*)
⟨*proof*⟩
lemma *coprime-commute*: *coprime a b* ⟷ *coprime b a* ⟨*proof*⟩

lemma *coprime-bezout*: *coprime a b* ⟷ (∃ *x y*. *a* ∗ *x* − *b* ∗ *y* = *1* ∨ *b* ∗ *x* − *a*
∗ *y* = *1*)
⟨*proof*⟩

lemma *coprime-divprod*: *d dvd a* ∗ *b* ⟹ *coprime d a* ⟹ *d dvd b*
⟨*proof*⟩

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

lemma *gcd-coprime*:
assumes *z*: *gcd a b* ≠ *0* **and** *a*: *a* = *a'* ∗ *gcd a b* **and** *b*: *b* = *b'* ∗ *gcd a b*
shows *coprime a' b'*
⟨*proof*⟩
lemma *coprime-0*: *coprime d 0* ⟷ *d* = *1* ⟨*proof*⟩
lemma *coprime-mul*: **assumes** *da*: *coprime d a* **and** *db*: *coprime d b*
shows *coprime d* (*a* ∗ *b*)
⟨*proof*⟩
lemma *coprime-lmul2*: **assumes** *dab*: *coprime d* (*a* ∗ *b*) **shows** *coprime d b*
⟨*proof*⟩

lemma *coprime-rmul2*: *coprime d* (*a* ∗ *b*) ⟹ *coprime d a*
⟨*proof*⟩
lemma *coprime-mul-eq*: *coprime d* (*a* ∗ *b*) ⟷ *coprime d a* ∧ *coprime d b*
⟨*proof*⟩

lemma *gcd-coprime-exists*:

assumes *nz*: *gcd a b* ≠ *0*
shows ∃ *a' b'*. *a* = *a'* ∗ *gcd a b* ∧ *b* = *b'* ∗ *gcd a b* ∧ *coprime a' b'*
⟨*proof*⟩

lemma *coprime-exp*: *coprime d a* ==> *coprime d* (*a*^*n*)
⟨*proof*⟩

lemma *coprime-exp-imp*: *coprime a b* ==> *coprime* (*a* ^*n*) (*b* ^*n*)
⟨*proof*⟩
lemma *coprime-refl*[*simp*]: *coprime n n* ⟷ *n* = *1* ⟨*proof*⟩
lemma *coprime-plus1*[*simp*]: *coprime* (*n* + *1*) *n*
⟨*proof*⟩
lemma *coprime-minus1*: *n* ≠ *0* ==> *coprime* (*n* − *1*) *n*
⟨*proof*⟩

lemma *bezout-gcd-pow*: ∃ *x y*. *a* ^*n* ∗ *x* − *b* ^ *n* ∗ *y* = *gcd a b* ^ *n* ∨ *b* ^ *n* ∗ *x* − *a* ^ *n* ∗ *y* = *gcd a b* ^ *n*
⟨*proof*⟩

lemma *gcd-exp*: *gcd* (*a*^*n*) (*b*^*n*) = *gcd a b*^*n*
⟨*proof*⟩

lemma *coprime-exp2*: *coprime* (*a* ^ *Suc n*) (*b*^ *Suc n*) ⟷ *coprime a b*
⟨*proof*⟩

lemma *division-decomp*: **assumes** *dc*: (*a*::*nat*) *dvd b* ∗ *c*
shows ∃ *b' c'*. *a* = *b'* ∗ *c'* ∧ *b' dvd b* ∧ *c' dvd c*
⟨*proof*⟩

lemma *nat-power-eq-0-iff*: (*m*::*nat*) ^ *n* = *0* ⟷ *n* ≠ *0* ∧ *m* = *0* ⟨*proof*⟩

lemma *divides-rev*: **assumes** *ab*: (*a*::*nat*) ^ *n dvd b* ^*n* **and** *n*:*n* ≠ *0* **shows** *a dvd b*
⟨*proof*⟩

lemma *divides-mul*: **assumes** *mr*: *m dvd r* **and** *nr*: *n dvd r* **and** *mn*:*coprime m n*
shows *m* ∗ *n dvd r*
⟨*proof*⟩

A binary form of the Chinese Remainder Theorem.

lemma *chinese-remainder*: **assumes** *ab*: *coprime a b* **and** *a*:*a* ≠ *0* **and** *b*:*b* ≠ *0*
shows ∃ *x q1 q2*. *x* = *u* + *q1* ∗ *a* ∧ *x* = *v* + *q2* ∗ *b*
⟨*proof*⟩

Primality

A few useful theorems about primes

lemma *prime-0*[*simp*]: ~*prime 0* ⟨*proof*⟩
lemma *prime-1*[*simp*]: ~ *prime 1* ⟨*proof*⟩

lemma *prime-Suc0*[*simp*]: ~ *prime* (*Suc 0*) ⟨*proof*⟩

lemma *prime-ge-2*: *prime p* ==> *p* ≥ *2* ⟨*proof*⟩
lemma *prime-factor*: **assumes** *n*: *n* ≠ *1* **shows** ∃ *p*. *prime p* ∧ *p dvd n*
⟨*proof*⟩

lemma *prime-factor-lt*: **assumes** *p*: *prime p* **and** *n*: *n* ≠ *0* **and** *npm*:*n* = *p* ∗ *m*
shows *m* < *n*
⟨*proof*⟩

lemma *euclid-bound*: ∃ *p*. *prime p* ∧ *n* < *p* ∧ *p* <= *Suc* (*fact n*)
⟨*proof*⟩

lemma *euclid*: ∃ *p*. *prime p* ∧ *p* > *n* ⟨*proof*⟩
lemma *primes-infinite*: ¬ (*finite* {*p*. *prime p*})
⟨*proof*⟩

lemma *coprime-prime*: **assumes** *ab*: *coprime a b*
shows ~(*prime p* ∧ *p dvd a* ∧ *p dvd b*)
⟨*proof*⟩
lemma *coprime-prime-eq*: *coprime a b* ⟷ (∀ *p*. ~(*prime p* ∧ *p dvd a* ∧ *p dvd b*))

(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *prime-coprime*: **assumes** *p*: *prime p*
shows *n* = *1* ∨ *p dvd n* ∨ *coprime p n*
⟨*proof*⟩

lemma *prime-coprime-strong*: *prime p* ⟹ *p dvd n* ∨ *coprime p n*
⟨*proof*⟩

declare *coprime-0*[*simp*]

lemma *coprime-0'*[*simp*]: *coprime 0 d* ⟷ *d* = *1* ⟨*proof*⟩
lemma *coprime-bezout-strong*: **assumes** *ab*: *coprime a b* **and** *b*: *b* ≠ *1*
shows ∃ *x y*. *a* ∗ *x* = *b* ∗ *y* + *1*
⟨*proof*⟩

lemma *bezout-prime*: **assumes** *p*: *prime p* **and** *pa*: ¬ *p dvd a*
shows ∃ *x y*. *a*∗*x* = *p*∗*y* + *1*
⟨*proof*⟩
lemma *prime-divprod*: **assumes** *p*: *prime p* **and** *pab*: *p dvd a*∗*b*
shows *p dvd a* ∨ *p dvd b*
⟨*proof*⟩

lemma *prime-divprod-eq*: **assumes** *p*: *prime p*
shows *p dvd a*∗*b* ⟷ *p dvd a* ∨ *p dvd b*
⟨*proof*⟩

lemma *prime-divexp*: **assumes** *p*:*prime p* **and** *px*: *p dvd x^n*
shows *p dvd x*
⟨*proof*⟩

lemma *prime-divexp-n*: *prime p* ⟹ *p dvd x^n* ⟹ *p^n dvd x^n*
⟨*proof*⟩

lemma *coprime-prime-dvd-ex*: **assumes** *xy*: ¬*coprime x y*
shows ∃*p*. *prime p* ∧ *p dvd x* ∧ *p dvd y*
⟨*proof*⟩
lemma *coprime-sos*: **assumes** *xy*: *coprime x y*
shows *coprime* (*x* ∗ *y*) (*x^2* + *y^2*)
⟨*proof*⟩

lemma *distinct-prime-coprime*: *prime p* ⟹ *prime q* ⟹ *p* ≠ *q* ⟹ *coprime p q*
⟨*proof*⟩

lemma *prime-coprime-lt*: **assumes** *p*: *prime p* **and** *x*: *0* < *x* **and** *xp*: *x* < *p*
shows *coprime x p*
⟨*proof*⟩

lemma *even-dvd*[*simp*]: *even* (*n*::*nat*) ⟷ *2 dvd n* ⟨*proof*⟩
lemma *prime-odd*: *prime p* ⟹ *p* = *2* ∨ *odd p* ⟨*proof*⟩

One property of coprimality is easier to prove via prime factors.

lemma *prime-divprod-pow*:
assumes *p*: *prime p* **and** *ab*: *coprime a b* **and** *pab*: *p^n dvd a* ∗ *b*
shows *p^n dvd a* ∨ *p^n dvd b*
⟨*proof*⟩

lemma *nat-mult-eq-one*: (*n*::*nat*) ∗ *m* = *1* ⟷ *n* = *1* ∧ *m* = *1* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *power-Suc0*[*simp*]: *Suc 0 ^ n* = *Suc 0*
⟨*proof*⟩
lemma *coprime-pow*: **assumes** *ab*: *coprime a b* **and** *abcn*: *a* ∗ *b* = *c ^n*
shows ∃*r s*. *a* = *r^n* ∧ *b* = *s ^n*
⟨*proof*⟩

More useful lemmas.

lemma *prime-product*:
assumes *prime* (*p* ∗ *q*)
shows *p* = *1* ∨ *q* = *1*
⟨*proof*⟩

lemma *prime-exp*: *prime* (*p^n*) ⟷ *prime p* ∧ *n* = *1*
⟨*proof*⟩

lemma *prime-power-mult*:
assumes *p*: *prime p* **and** *xy*: $x * y = p \hat{} k$
shows $\exists i\, j.\ x = p \hat{} i \wedge y = p \hat{} j$
⟨*proof*⟩

lemma *prime-power-exp*: **assumes** *p*: *prime p* **and** *n*:$n \neq 0$
and *xn*: $x \hat{} n = p \hat{} k$ **shows** $\exists i.\ x = p \hat{} i$
⟨*proof*⟩

lemma *divides-primepow*: **assumes** *p*: *prime p*
shows *d dvd p*$\hat{}k \longleftrightarrow (\exists\ i.\ i \leq k \wedge d = p \hat{} i)$
⟨*proof*⟩

lemma *coprime-divisors*: *d dvd a* $\Longrightarrow$ *e dvd b* $\Longrightarrow$ *coprime a b* $\Longrightarrow$ *coprime d e*
⟨*proof*⟩

declare *power-Suc0*[*simp del*]
declare *even-dvd*[*simp del*]

end

52 Pocklington: Pocklington's Theorem for Primes

theory *Pocklington*
imports *Main Primes*
begin

definition *modeq*:: *nat => nat => nat => bool* ((1[- = -] '(*mod* -')))
where $[a = b]\ (mod\ p) == ((a\ mod\ p) = (b\ mod\ p))$

definition *modneq*:: *nat => nat => nat => bool* ((1[- ≠ -] '(*mod* -')))
where $[a \neq b]\ (mod\ p) == ((a\ mod\ p) \neq (b\ mod\ p))$

lemma *modeq-trans*:
$[\![\ [a = b]\ (mod\ p);\ [b = c]\ (mod\ p)\ ]\!] \Longrightarrow [a = c]\ (mod\ p)$
⟨*proof*⟩

lemma *nat-mod-lemma*: **assumes** *xyn*: $[x = y]\ (mod\ n)$ **and** *xy*:$y \leq x$
shows $\exists q.\ x = y + n * q$
⟨*proof*⟩

lemma *nat-mod*[*algebra*]: $[x = y]\ (mod\ n) \longleftrightarrow (\exists\, q1\ q2.\ x + n * q1 = y + n * q2)$
⟨*proof*⟩

lemma *prime*: $prime\ p \longleftrightarrow p \neq 0 \wedge p{\neq}1 \wedge (\forall m.\ 0 < m \wedge m < p \longrightarrow coprime\ p\ m)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *finite-number-segment*: $card\ \{\ m.\ 0 < m \wedge m < n\ \} = n - 1$
⟨*proof*⟩

lemma *coprime-mod*: **assumes** *n*: $n \neq 0$ **shows** $coprime\ (a\ mod\ n)\ n \longleftrightarrow coprime\ a\ n$
⟨*proof*⟩

lemma *cong-mod-01*[*simp*,*presburger*]:
$[x = y]\ (mod\ 0) \longleftrightarrow x = y\ [x = y]\ (mod\ 1)\ [x = 0]\ (mod\ n) \longleftrightarrow n\ dvd\ x$
⟨*proof*⟩

lemma *cong-sub-cases*:
$[x = y]\ (mod\ n) \longleftrightarrow (if\ x <= y\ then\ [y - x = 0]\ (mod\ n)\ else\ [x - y = 0]\ (mod\ n))$
⟨*proof*⟩

lemma *cong-mult-lcancel*: **assumes** *an*: $coprime\ a\ n$ **and** *axy*:$[a * x = a * y]\ (mod\ n)$
shows $[x = y]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-mult-rcancel*: **assumes** *an*: $coprime\ a\ n$ **and** *axy*:$[x{*}a = y{*}a]\ (mod\ n)$
shows $[x = y]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-refl*: $[x = x]\ (mod\ n)$ ⟨*proof*⟩

lemma *eq-imp-cong*: $a = b \Longrightarrow [a = b]\ (mod\ n)$ ⟨*proof*⟩

lemma *cong-commute*: $[x = y]\ (mod\ n) \longleftrightarrow [y = x]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-trans*[*trans*]: $[x = y]\ (mod\ n) \Longrightarrow [y = z]\ (mod\ n) \Longrightarrow [x = z]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-add*: **assumes** *xx'*: $[x = x']\ (mod\ n)$ **and** *yy'*:$[y = y']\ (mod\ n)$
shows $[x + y = x' + y']\ (mod\ n)$
⟨*proof*⟩

lemma *cong-mult*: **assumes** *xx'*: $[x = x']\ (mod\ n)$ **and** *yy'*:$[y = y']\ (mod\ n)$
shows $[x * y = x' * y']\ (mod\ n)$
⟨*proof*⟩

lemma *cong-exp*: $[x = y]\ (mod\ n) \Longrightarrow [x\hat{}k = y\hat{}k]\ (mod\ n)$
⟨*proof*⟩
lemma *cong-sub*: **assumes** *xx'*: $[x = x']\ (mod\ n)$ **and** *yy'*: $[y = y']\ (mod\ n)$
and *yx*: $y <= x$ **and** *yx'*: $y' <= x'$
shows $[x - y = x' - y']\ (mod\ n)$
⟨*proof*⟩

lemma *cong-mult-lcancel-eq*: **assumes** *an*: *coprime a n*
shows $[a * x = a * y]\ (mod\ n) \longleftrightarrow [x = y]\ (mod\ n)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *cong-mult-rcancel-eq*: **assumes** *an*: *coprime a n*
shows $[x * a = y * a]\ (mod\ n) \longleftrightarrow [x = y]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-add-lcancel-eq*: $[a + x = a + y]\ (mod\ n) \longleftrightarrow [x = y]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-add-rcancel-eq*: $[x + a = y + a]\ (mod\ n) \longleftrightarrow [x = y]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-add-rcancel*: $[x + a = y + a]\ (mod\ n) \Longrightarrow [x = y]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-add-lcancel*: $[a + x = a + y]\ (mod\ n) \Longrightarrow [x = y]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-add-lcancel-eq-0*: $[a + x = a]\ (mod\ n) \longleftrightarrow [x = 0]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-add-rcancel-eq-0*: $[x + a = a]\ (mod\ n) \longleftrightarrow [x = 0]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-imp-eq*: **assumes** *xn*: $x < n$ **and** *yn*: $y < n$ **and** *xy*: $[x = y]\ (mod\ n)$
shows $x = y$
⟨*proof*⟩

lemma *cong-divides-modulus*: $[x = y]\ (mod\ m) \Longrightarrow n\ dvd\ m ==> [x = y]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-0-divides*: $[x = 0]\ (mod\ n) \longleftrightarrow n\ dvd\ x$ ⟨*proof*⟩

lemma *cong-1-divides*:$[x = 1]\ (mod\ n) ==> n\ dvd\ x - 1$

⟨*proof*⟩

lemma *cong-divides*: $[x = y]\ (mod\ n) \Longrightarrow n\ dvd\ x \longleftrightarrow n\ dvd\ y$
⟨*proof*⟩

lemma *cong-coprime*: **assumes** *xy*: $[x = y]\ (mod\ n)$
shows $coprime\ n\ x \longleftrightarrow coprime\ n\ y$
⟨*proof*⟩

lemma *cong-mod*: $\sim(n = 0) \Longrightarrow [a\ mod\ n = a]\ (mod\ n)$ ⟨*proof*⟩

lemma *mod-mult-cong*: $\sim(a = 0) \Longrightarrow \sim(b = 0)$
$\Longrightarrow [x\ mod\ (a * b) = y]\ (mod\ a) \longleftrightarrow [x = y]\ (mod\ a)$
⟨*proof*⟩

lemma *cong-mod-mult*: $[x = y]\ (mod\ n) \Longrightarrow m\ dvd\ n \Longrightarrow [x = y]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-le*: $y <= x \Longrightarrow [x = y]\ (mod\ n) \longleftrightarrow (\exists q.\ x = q * n + y)$
⟨*proof*⟩

lemma *cong-to-1*: $[a = 1]\ (mod\ n) \longleftrightarrow a = 0 \wedge n = 1 \vee (\exists m.\ a = 1 + m * n)$
⟨*proof*⟩

lemma *cong-solve*: **assumes** *an*: $coprime\ a\ n$ **shows** $\exists x.\ [a * x = b]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-solve-unique*: **assumes** *an*: $coprime\ a\ n$ **and** *nz*: $n \neq 0$
shows $\exists!x.\ x < n \wedge [a * x = b]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-solve-unique-nontrivial*:
assumes *p*: $prime\ p$ **and** *pa*: $coprime\ p\ a$ **and** *x0*: $0 < x$ **and** *xp*: $x < p$
shows $\exists!y.\ 0 < y \wedge y < p \wedge [x * y = a]\ (mod\ p)$
⟨*proof*⟩
lemma *cong-unique-inverse-prime*:
assumes *p*: $prime\ p$ **and** *x0*: $0 < x$ **and** *xp*: $x < p$
shows $\exists!y.\ 0 < y \wedge y < p \wedge [x * y = 1]\ (mod\ p)$
⟨*proof*⟩

lemma *cong-chinese*:
assumes *ab*: $coprime\ a\ b$ **and** *xya*: $[x = y]\ (mod\ a)$

and *xyb*: $[x = y]$ $(mod\ b)$
shows $[x = y]$ $(mod\ a*b)$
⟨*proof*⟩

lemma *chinese-remainder-unique*:
assumes *ab*: *coprime a b* **and** *az*: $a \neq 0$ **and** *bz*: $b \neq 0$
shows $\exists!x.\ x < a * b \wedge [x = m]\ (mod\ a) \wedge [x = n]\ (mod\ b)$
⟨*proof*⟩

lemma *chinese-remainder-coprime-unique*:
assumes *ab*: *coprime a b* **and** *az*: $a \neq 0$ **and** *bz*: $b \neq 0$
and *ma*: *coprime m a* **and** *nb*: *coprime n b*
shows $\exists!x.$ *coprime* $x\ (a * b) \wedge x < a * b \wedge [x = m]\ (mod\ a) \wedge [x = n]\ (mod\ b)$
⟨*proof*⟩

definition *phi-def*: $\varphi\ n = card\ \{\ m.\ 0 < m \wedge m <= n \wedge coprime\ m\ n\ \}$
lemma *phi-0*[*simp*]: $\varphi\ 0 = 0$
⟨*proof*⟩

lemma *phi-finite*[*simp*]: *finite* $(\{\ m.\ 0 < m \wedge m <= n \wedge coprime\ m\ n\ \})$
⟨*proof*⟩

declare *coprime-1*[*presburger*]
lemma *phi-1*[*simp*]: $\varphi\ 1 = 1$
⟨*proof*⟩

lemma [*simp*]: $\varphi\ (Suc\ 0) = Suc\ 0$ ⟨*proof*⟩

lemma *phi-alt*: $\varphi(n) = card\ \{\ m.\ coprime\ m\ n \wedge m < n\}$
⟨*proof*⟩

lemma *phi-finite-lemma*[*simp*]: *finite* $\{m.\ coprime\ m\ n \wedge\ m < n\}$ (**is** *finite ?S*)
⟨*proof*⟩

lemma *phi-another*: **assumes** *n*: $n \neq 1$
shows $\varphi\ n = card\ \{m.\ 0 < m \wedge m < n \wedge coprime\ m\ n\ \}$
⟨*proof*⟩

lemma *phi-limit*: $\varphi\ n \leq n$
⟨*proof*⟩

lemma *stupid*[*simp*]: $\{m.\ (0::nat) < m \wedge m < n\} = \{1..<n\}$
⟨*proof*⟩

lemma *phi-limit-strong*: **assumes** *n*: $n \neq 1$
shows $\varphi(n) \leq n - 1$

⟨*proof*⟩

lemma *phi-lowerbound-1-strong*: **assumes** *n*: $n \geq 1$
shows $\varphi(n) \geq 1$
⟨*proof*⟩

lemma *phi-lowerbound-1*: *2 <= n ==> 1 <= φ(n)*
⟨*proof*⟩

lemma *phi-lowerbound-2*: **assumes** *n*: *3 <= n* **shows** *2 <= φ (n)*
⟨*proof*⟩

lemma *phi-prime*: $\varphi\ n = n - 1 \wedge n \neq 0 \wedge n \neq 1 \longleftrightarrow$ *prime n*
⟨*proof*⟩

lemma *phi-multiplicative*: **assumes** *ab*: *coprime a b*
shows $\varphi\ (a * b) = \varphi\ a * \varphi\ b$
⟨*proof*⟩

lemma *nproduct-mod*:
assumes *fS*: *finite S* **and** *n0*: $n \neq 0$
shows [*setprod* (λ*m*. *a*(*m*) *mod n*) *S* = *setprod a S*] (*mod n*)
⟨*proof*⟩

lemma *nproduct-cmul*:
assumes *fS*:*finite S*
shows *setprod* (λ*m*. (*c*::'*a*::{*comm-monoid-mult*,*recpower*})∗ *a*(*m*)) *S* = *c* ^ (*card S*) ∗ *setprod a S*
⟨*proof*⟩

lemma *coprime-nproduct*:
assumes *fS*: *finite S* **and** *Sn*: ∀ *x*∈*S*. *coprime n* (*a x*)
shows *coprime n* (*setprod a S*)
⟨*proof*⟩

lemma *fermat-little*: **assumes** *an*: *coprime a n*
shows [*a* ^ (*φ n*) = *1*] (*mod n*)
⟨*proof*⟩

lemma *fermat-little-prime*: **assumes** *p*: *prime p* **and** *ap*: *coprime a p*
shows [*a* ^ (*p* − *1*) = *1*] (*mod p*)
⟨*proof*⟩

lemma *lucas-coprime-lemma*:
assumes *m*: $m \neq 0$ **and** *am*: $[a \hat{} m = 1] \ (mod\ n)$
shows *coprime a n*
⟨*proof*⟩

lemma *lucas-weak*:
assumes *n*: $n \geq 2$ **and** *an*:$[a \hat{} (n - 1) = 1] \ (mod\ n)$
and *nm*: $\forall m.\ 0 < m \wedge m < n - 1 \longrightarrow \neg\ [a \hat{} m = 1] \ (mod\ n)$
shows *prime n*
⟨*proof*⟩

lemma *nat-exists-least-iff*: $(\exists\,(n::nat).\ P\ n) \longleftrightarrow (\exists\, n.\ P\ n \wedge (\forall\, m < n.\ \neg\ P\ m))$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *nat-exists-least-iff'*: $(\exists\,(n::nat).\ P\ n) \longleftrightarrow (P\ (Least\ P) \wedge (\forall\, m < (Least\ P).\ \neg\ P\ m))$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *power-mod*: $((x::nat)\ mod\ m) \hat{} n\ mod\ m = x \hat{} n\ mod\ m$
⟨*proof*⟩

lemma *lucas*:
assumes *n2*: $n \geq 2$ **and** *an1*: $[a \hat{} (n - 1) = 1] \ (mod\ n)$
and *pn*: $\forall p.\ prime\ p \wedge p\ dvd\ n - 1 \longrightarrow \neg\ [a \hat{} ((n - 1)\ div\ p) = 1] \ (mod\ n)$
shows *prime n*
⟨*proof*⟩

definition $ord\ n\ a = (if\ coprime\ n\ a\ then\ Least\ (\lambda d.\ d > 0 \wedge [a\ \hat{}\ d = 1] \ (mod\ n))\ else\ 0)$

lemma *coprime-ord*:
assumes *na*: *coprime n a*
shows $ord\ n\ a > 0 \wedge [a\ \hat{}\ (ord\ n\ a) = 1] \ (mod\ n) \wedge (\forall\, m.\ 0 < m \wedge m < ord\ n\ a \longrightarrow \neg\ [a \hat{}\ m = 1] \ (mod\ n))$
⟨*proof*⟩

lemma *ord-works*:
$[a\ \hat{}\ (ord\ n\ a) = 1] \ (mod\ n) \wedge (\forall\, m.\ 0 < m \wedge m < ord\ n\ a \longrightarrow \sim [a \hat{}\ m = 1] \ (mod\ n))$
⟨*proof*⟩

lemma *ord*: $[a\hat{}(ord\ n\ a) = 1]\ (mod\ n)$ ⟨*proof*⟩
lemma *ord-minimal*: $0 < m \Longrightarrow m < ord\ n\ a \Longrightarrow {\sim}[a\hat{}m = 1]\ (mod\ n)$
⟨*proof*⟩
lemma *ord-eq-0*: $ord\ n\ a = 0 \longleftrightarrow {\sim} coprime\ n\ a$
⟨*proof*⟩

lemma *ord-divides*:
$[a\ \hat{}\ d = 1]\ (mod\ n) \longleftrightarrow ord\ n\ a\ dvd\ d$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *order-divides-phi*: $coprime\ n\ a \Longrightarrow ord\ n\ a\ dvd\ \varphi\ n$
⟨*proof*⟩
lemma *order-divides-expdiff*:
assumes *na*: *coprime n a*
shows $[a\hat{}d = a\hat{}e]\ (mod\ n) \longleftrightarrow [d = e]\ (mod\ (ord\ n\ a))$
⟨*proof*⟩

lemma *prime-prime-factor*:
$prime\ n \longleftrightarrow n \neq 1 \wedge (\forall p.\ prime\ p \wedge p\ dvd\ n \longrightarrow p = n)$
⟨*proof*⟩

lemma *prime-divisor-sqrt*:
$prime\ n \longleftrightarrow n \neq 1 \wedge (\forall d.\ d\ dvd\ n \wedge d\hat{}2 \leq n \longrightarrow d = 1)$
⟨*proof*⟩
lemma *prime-prime-factor-sqrt*:
$prime\ n \longleftrightarrow n \neq 0 \wedge n \neq 1 \wedge \neg (\exists p.\ prime\ p \wedge p\ dvd\ n \wedge p\hat{}2 \leq n)$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *pocklington-lemma*:
assumes *n*: $n \geq 2$ **and** *nqr*: $n - 1 = q*r$ **and** *an*: $[a\hat{}\ (n - 1) = 1]\ (mod\ n)$
and *aq*:$\forall p.\ prime\ p \wedge p\ dvd\ q \longrightarrow coprime\ (a\hat{}\ ((n - 1)\ div\ p) - 1)\ n$
and *pp*: *prime p* **and** *pn*: *p dvd n*
shows $[p = 1]\ (mod\ q)$
⟨*proof*⟩

lemma *pocklington*:
assumes *n*: $n \geq 2$ **and** *nqr*: $n - 1 = q*r$ **and** *sqr*: $n \leq q\hat{}2$
and *an*: $[a\hat{}\ (n - 1) = 1]\ (mod\ n)$
and *aq*:$\forall p.\ prime\ p \wedge p\ dvd\ q \longrightarrow coprime\ (a\hat{}\ ((n - 1)\ div\ p) - 1)\ n$
shows *prime n*
⟨*proof*⟩

lemma *pocklington-alt*:
assumes *n*: $n \geq 2$ **and** *nqr*: $n - 1 = q*r$ **and** *sqr*: $n \leq q\hat{}2$

and *an*: $[a\hat{}\ (n - 1) = 1]\ (mod\ n)$
and *aq*:$\forall p.\ prime\ p \wedge p\ dvd\ q \longrightarrow (\exists b.\ [a\hat{}((n - 1)\ div\ p) = b]\ (mod\ n) \wedge coprime\ (b - 1)\ n)$
shows *prime n*
⟨*proof*⟩

definition *primefact ps n* = (*foldr op* ∗ *ps* 1 = *n* ∧ (∀ *p*∈ *set ps*. *prime p*))

lemma *primefact*: **assumes** *n*: $n \neq 0$
shows ∃ *ps*. *primefact ps n*
⟨*proof*⟩

lemma *primefact-contains*:
assumes *pf*: *primefact ps n* **and** *p*: *prime p* **and** *pn*: *p dvd n*
shows $p \in set\ ps$
⟨*proof*⟩

lemma *primefact-variant*: *primefact ps n* ⟷ *foldr op* ∗ *ps 1* = *n* ∧ *list-all prime ps* ⟨*proof*⟩

lemma *lucas-primefact*:
assumes *n*: $n \geq 2$ **and** *an*: $[a\hat{}(n - 1) = 1]\ (mod\ n)$
and *psn*: *foldr op* ∗ *ps 1* = *n* − *1*
and *psp*: *list-all* (λ*p*. *prime p* ∧ ¬ $[a\hat{}((n - 1)\ div\ p) = 1]\ (mod\ n)$) *ps*
shows *prime n*
⟨*proof*⟩

lemma *mod-le*: **assumes** *n*: $n \neq (0::nat)$ **shows** $m\ mod\ n \leq m$
⟨*proof*⟩

lemma *pocklington-primefact*:
assumes *n*: $n \geq 2$ **and** *qrn*: $q*r = n - 1$ **and** *nq2*: $n \leq q\hat{}2$
and *arnb*: $(a\hat{}r)\ mod\ n = b$ **and** *psq*: *foldr op* ∗ *ps 1* = *q*
and *bqn*: $(b\hat{}q)\ mod\ n = 1$
and *psp*: *list-all* (λ*p*. *prime p* ∧ *coprime* $((b\hat{}(q\ div\ p))\ mod\ n - 1)\ n)$ *ps*
shows *prime n*
⟨*proof*⟩

end

53 Poly-Deriv: Polynomials and Differentiation

theory *Poly-Deriv*
imports *Deriv Polynomial*
begin

53.1 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

54 Product-plus: Additive group operations on product types

theory *Product-plus*
imports *Main*
begin

54.1 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*

54.2 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*⟩

end

55 Product-Vector: Cartesian Products as Vector Spaces

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

55.1 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

55.2 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* × '*b*) = *scaleR* (*inverse* (*norm x*)) *x*

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

instance ⟨*proof*⟩

end

55.3 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

55.4 Pair operations are linear and continuous

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*⟩

TODO: The next three proofs are nearly identical to each other. Is there a good way to factor out the common parts?

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

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

lemma *LIM-Pair*:
assumes f $--$ x $-->$ a **and** g $--$ x $-->$ b
shows $(\lambda x.\ (f\ x,\ g\ x))$ $--$ x $-->$ $(a,\ b)$
⟨*proof*⟩

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

55.5 Product is a complete vector space

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

55.6 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

56 Random: A HOL random engine

theory *Random*
imports *Code-Index*
begin

notation *fcomp* (**infixl** *o>* 60)
notation *scomp* (**infixl** *o→* 60)

56.1 Auxiliary functions

definition *inc-shift* :: *index* ⇒ *index* ⇒ *index* **where**
inc-shift v k = (*if v* = *k then 1 else k* + *1*)

definition *minus-shift* :: *index* ⇒ *index* ⇒ *index* ⇒ *index* **where**
minus-shift r k l = (*if k* < *l then r* + *k* − *l else k* − *l*)

fun *log* :: *index* ⇒ *index* ⇒ *index* **where**
log b i = (*if b* ≤ *1* ∨ *i* < *b then 1 else 1* + *log b* (*i div b*))

56.2 Random seeds

types *seed* = *index* × *index*

primrec *next* :: *seed* ⇒ *index* × *seed* **where**
next (*v*, *w*) = (*let*
k = *v div 53668*;
v' = *minus-shift 2147483563* (*40014* ∗ (*v mod 53668*)) (*k* ∗ *12211*);
l = *w div 52774*;
w' = *minus-shift 2147483399* (*40692* ∗ (*w mod 52774*)) (*l* ∗ *3791*);
z = *minus-shift 2147483562 v'* (*w'* + *1*) + *1*
in (*z*, (*v'*, *w'*)))

lemma *next-not-0*:
fst (*next s*) ≠ *0*
⟨*proof*⟩

primrec *seed-invariant* :: *seed* ⇒ *bool* **where**
seed-invariant (*v*, *w*) ⟷ *0* < *v* ∧ *v* < *9438322952* ∧ *0* < *w* ∧ *True*

lemma *if-same*: (*if b then f x else f y*) = *f* (*if b then x else y*)
⟨*proof*⟩

definition *split-seed* :: *seed* ⇒ *seed* × *seed* **where**
split-seed s = (*let*
(*v*, *w*) = *s*;
(*v'*, *w'*) = *snd* (*next s*);
v'' = *inc-shift 2147483562 v*;
s'' = (*v''*, *w'*);
w'' = *inc-shift 2147483398 w*;
s''' = (*v'*, *w''*)
in (*s''*, *s'''*))

56.3 Base selectors

fun *iterate* :: *index* ⇒ ('*b* ⇒ '*a* ⇒ '*b* × '*a*) ⇒ '*b* ⇒ '*a* ⇒ '*b* × '*a* **where**
iterate k f x = (*if k* = *0 then Pair x else f x o*→ *iterate* (*k* − *1*) *f*)

definition *range* :: *index* ⇒ *seed* ⇒ *index* × *seed* **where**
range k = *iterate* (*log 2147483561 k*)
(λ*l*. *next o*→ (λ*v*. *Pair* (*v* + *l* ∗ *2147483561*))) *1*
o→ (λ*v*. *Pair* (*v mod k*))

lemma *range*:
k > *0* ⟹ *fst* (*range k s*) < *k*
⟨*proof*⟩

definition *select* :: '*a list* ⇒ *seed* ⇒ '*a* × *seed* **where**
select xs = *range* (*Code-Index.of-nat* (*length xs*))
o→ (λ*k*. *Pair* (*nth xs* (*Code-Index.nat-of k*)))

lemma *select*:
assumes *xs* ≠ []
shows *fst* (*select xs s*) ∈ *set xs*
⟨*proof*⟩

definition *select-default* :: *index* ⇒ '*a* ⇒ '*a* ⇒ *seed* ⇒ '*a* × *seed* **where**
[*code del*]: *select-default k x y* = *range k*
o→ (λ*l*. *Pair* (*if l* + *1* < *k then x else y*))

lemma *select-default-zero*:
fst (*select-default 0 x y s*) = *y*
⟨*proof*⟩

lemma *select-default-code* [*code*]:
select-default k x y = (*if k* = *0*
then range 1 o→ (λ-. *Pair y*)
else range k o→ (λ*l*. *Pair* (*if l* + *1* < *k then x else y*)))
⟨*proof*⟩

56.4 *ML* interface

⟨*ML*⟩

no-notation *fcomp* (**infixl** *o>* *60*)
no-notation *scomp* (**infixl** *o→* *60*)

end

57 Quickcheck: A simple counterexample generator

theory *Quickcheck*
imports *Random Code-Eval Map*
begin

57.1 The *random* class

class *random* = *typerep* +
  fixes *random* :: *index* ⇒ *seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *seed*

  Type *'a itself*

instantiation *itself* :: ({*type*, *typerep*}) *random*
begin

definition
  random - = *Pair* (*TYPE*(*'a*), λ*u*. *Code-Eval.Const* (*STR ''TYPE''*) *TYPE-REP*(*'a*))

instance ⟨*proof*⟩

end

57.2 Quickcheck generator

⟨*ML*⟩

end

58 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 *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*⟩

end

end

59 Quotient: Quotient types

theory *Quotient*
imports *Main*
begin

We introduce the notion of quotient types over equivalence relations via type classes.

59.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations ∼ :: *'a* => *'a* => *bool*.

class *eqv* =
fixes *eqv* :: *'a* ⇒ *'a* ⇒ *bool* (**infixl** ∼ *50*)

class *equiv* = *eqv* +
assumes *equiv-refl* [*intro*]: *x* ∼ *x*
assumes *equiv-trans* [*trans*]: *x* ∼ *y* ⟹ *y* ∼ *z* ⟹ *x* ∼ *z*
assumes *equiv-sym* [*sym*]: *x* ∼ *y* ⟹ *y* ∼ *x*

lemma *equiv-not-sym* [*sym*]: ¬ (*x* ∼ *y*) ==> ¬ (*y* ∼ (*x*::*'a*::*equiv*))
⟨*proof*⟩

lemma *not-equiv-trans1* [*trans*]: ¬ (*x* ∼ *y*) ==> *y* ∼ *z* ==> ¬ (*x* ∼ (*z*::*'a*::*equiv*))
⟨*proof*⟩

lemma *not-equiv-trans2* [*trans*]: *x* ∼ *y* ==> ¬ (*y* ∼ *z*) ==> ¬ (*x* ∼ (*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*]: $R \in quot \Longrightarrow (!!a.\ R = \{x.\ a \sim x\} \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

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

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

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

lemma *quot-cases* [*cases type*: *quot*]: $(!!a.\ A = \lfloor a \rfloor \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

59.2 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*⟩

59.3 Picking representing elements

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

theorem *pick-equiv* [*intro*]: $pick\ \lfloor a \rfloor \sim a$
⟨*proof*⟩

theorem *pick-inverse* [*intro*]: $\lfloor pick\ A \rfloor = A$
⟨*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*: $!!X\ Y.\ P\ X\ Y \Longrightarrow f\ X\ Y == g\ (pick\ X)\ (pick\ Y)$
and *cong*: $!!x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor$

$$\Longrightarrow P\ \lfloor x\rfloor\ \lfloor y\rfloor \Longrightarrow P\ \lfloor x'\rfloor\ \lfloor y'\rfloor \Longrightarrow g\ x\ y = g\ x'\ y'$$
and P: $P\ \lfloor a\rfloor\ \lfloor b\rfloor$
shows $f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ a\ b$
⟨*proof*⟩

theorem *quot-function*:
assumes $!!X\ Y.\ f\ X\ Y == g\ (pick\ X)\ (pick\ Y)$
and $!!x\ x'\ y\ y'.\ \lfloor x\rfloor = \lfloor x'\rfloor \Longrightarrow \lfloor y\rfloor = \lfloor y'\rfloor \Longrightarrow g\ x\ y = g\ x'\ y'$
shows $f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ a\ b$
⟨*proof*⟩

theorem *quot-function'*:
$(!!X\ Y.\ f\ X\ Y == g\ (pick\ X)\ (pick\ Y)) \Longrightarrow$
$(!!x\ x'\ y\ y'.\ x \sim x' \Longrightarrow y \sim y' \Longrightarrow g\ x\ y = g\ x'\ y') \Longrightarrow$
$f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ a\ b$
⟨*proof*⟩

end

60 Ramsey: Ramsey's Theorem

theory *Ramsey*
imports *Main Infinite-Set*
begin

60.1 Preliminaries

60.1.1 "Axiom" of Dependent Choice

consts *choice* :: $('a \Rightarrow bool) \Rightarrow ('a * 'a)\ set \Rightarrow nat \Rightarrow 'a$
— An integer-indexed chain of choices
primrec
choice-0: $choice\ P\ r\ 0 = (SOME\ x.\ P\ x)$

choice-Suc: $choice\ P\ r\ (Suc\ n) = (SOME\ y.\ P\ y\ \&\ (choice\ P\ r\ n,\ y) \in r)$

lemma *choice-n*:
assumes *P0*: *P x0*
and *Pstep*: $!!x.\ P\ x \Longrightarrow \exists y.\ P\ y\ \&\ (x,y) \in r$
shows $P\ (choice\ P\ r\ n)$
⟨*proof*⟩

lemma *dependent-choice*:
assumes *trans*: *trans r*
and *P0*: *P x0*
and *Pstep*: $!!x.\ P\ x \Longrightarrow \exists y.\ P\ y\ \&\ (x,y) \in r$
obtains $f :: nat \Rightarrow 'a$ **where**

!!n. P $(f\ n)$ **and** !!n m. $n < m$ ==> $(f\ n, f\ m) \in r$
⟨*proof*⟩

60.1.2 Partitions of a Set

definition
part :: *nat* => *nat* => *'a set* => (*'a set* => *nat*) => *bool*
— the function f partitions the r-subsets of the typically infinite set Y into s distinct categories.
where
part r s Y f = (∀ *X*. *X* ⊆ *Y* & *finite X* & *card X* = *r* --> *f X* < *s*)

For induction, we decrease the value of r in partitions.

lemma *part-Suc-imp-part*:
[| *infinite Y*; *part* (*Suc r*) *s Y f*; *y* ∈ *Y* |]
==> *part r s* (*Y* − {*y*}) (%*u*. *f* (*insert y u*))
⟨*proof*⟩

lemma *part-subset*: *part r s YY f* ==> *Y* ⊆ *YY* ==> *part r s Y f*
⟨*proof*⟩

60.2 Ramsey's Theorem: Infinitary Version

lemma *Ramsey-induction*:
fixes *s* **and** *r*::*nat*
shows
!!(*YY*::*'a set*) (*f*::*'a set* => *nat*).
[|*infinite YY*; *part r s YY f*|]
==> ∃ *Y' t'*. *Y'* ⊆ *YY* & *infinite Y'* & *t'* < *s* &
(∀ *X*. *X* ⊆ *Y'* & *finite X* & *card X* = *r* --> *f X* = *t'*)
⟨*proof*⟩

theorem *Ramsey*:
fixes *s r* :: *nat* **and** *Z*::*'a set* **and** *f*::*'a set* => *nat*
shows
[|*infinite Z*;
∀ *X*. *X* ⊆ *Z* & *finite X* & *card X* = *r* --> *f X* < *s*|]
==> ∃ *Y t*. *Y* ⊆ *Z* & *infinite Y* & *t* < *s*
& (∀ *X*. *X* ⊆ *Y* & *finite X* & *card X* = *r* --> *f X* = *t*)
⟨*proof*⟩

corollary *Ramsey2*:
fixes *s*::*nat* **and** *Z*::*'a set* **and** *f*::*'a set* => *nat*
assumes *infZ*: *infinite Z*
and *part*: ∀ *x*∈*Z*. ∀ *y*∈*Z*. *x*≠*y* --> *f*{*x*,*y*} < *s*
shows
∃ *Y t*. *Y* ⊆ *Z* & *infinite Y* & *t* < *s* & (∀ *x*∈*Y*. ∀ *y*∈*Y*. *x*≠*y* --> *f*{*x*,*y*} = *t*)
⟨*proof*⟩

60.3 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [4].

definition
disj-wf :: $('a * 'a)set \Rightarrow bool$
where
disj-wf r = $(\exists T.\ \exists n{::}nat.\ (\forall i{<}n.\ wf(T\ i))\ \&\ r = (\bigcup i{<}n.\ T\ i))$

definition
transition-idx :: $[nat \Rightarrow 'a,\ nat \Rightarrow ('a*'a)set,\ nat\ set] \Rightarrow nat$
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*:
$[|i{<}j;\ (s\ j,\ s\ i) \in T\ k;\ k{<}n|] \Longrightarrow transition\text{-}idx\ s\ T\ \{i,j\} < n$
⟨*proof*⟩

lemma *transition-idx-in*:
$[|i{<}j;\ (s\ j,\ s\ i) \in T\ k|] \Longrightarrow (s\ j,\ s\ i) \in T\ (transition\text{-}idx\ s\ T\ \{i,j\})$
⟨*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*:
$disj\text{-}wf(r) = (\exists T.\ \exists n{::}nat.\ (\forall i{<}n.\ wf(T\ i))\ \&\ r \subseteq (\bigcup i{<}n.\ T\ i))$
⟨*proof*⟩

theorem *trans-disj-wf-implies-wf*:
assumes *transr*: *trans r*
and *dwf*: *disj-wf*(*r*)
shows *wf r*
⟨*proof*⟩

end

61 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

62 RBT: Red-Black Trees

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨

This theory defines purely functional red-black trees which can be used as an efficient representation of finite maps.

62.1 Data type and invariant

The type $('k, 'v)$ *rbt* denotes red-black trees with keys of type $'k$ and values of type $'v$. To function properly, the key type must belong to the *linorder* class.

A value t of this type is a valid red-black tree if it satisfies the invariant *isrbt t*. This theory provides lemmas to prove that the invariant is satisfied throughout the computation.

The interpretation function *RBT.map-of* returns the partial map represented by a red-black tree:

RBT.map-of::$('a, 'b)$ *rbt* ⇒ $'a \rightharpoonup 'b$

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 $O(\log n)$.

62.2 Operations

Currently, the following operations are supported:

Empty::$('a, 'b)$ *rbt*

Returns the empty tree. $O(1)$

insrt::$'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *rbt* ⇒ $('a, 'b)$ *rbt*

Updates the map at a given position. $O(\log n)$

RBT.delete::$'a \Rightarrow ('a, 'b)$ *rbt* ⇒ $('a, 'b)$ *rbt*

Deletes a map entry at a given position. $O(\log n)$

union::$('a, 'b)$ *rbt* ⇒ $('a, 'b)$ *rbt* ⇒ $('a, 'b)$ *rbt*

Forms the union of two trees, preferring entries from the first one.

RBT.map::$('a \Rightarrow 'b) \Rightarrow ('c, 'a)$ *rbt* ⇒ $('c, 'b)$ *rbt*

Maps a function over the values of a map. $O(n)$

62.3 Invariant preservation

isrbt Empty (*Empty-isrbt*)
isrbt ?t $\Longrightarrow$ *isrbt* (*insrt ?k ?v ?t*) (*insrt-isrbt*)
isrbt ?t $\Longrightarrow$ *isrbt* (*RBT.delete ?k ?t*) (*delete-isrbt*)
isrbt ?lt $\Longrightarrow$ *isrbt* (*union ?lt ?rt*) (*union-isrbt*)
isrbt (*RBT.map ?f ?t*) = *isrbt ?t* (*map-isrbt*)

62.4 Map Semantics

map-of-Empty

RBT.map-of Empty = *Map.empty*

map-of-insert

isrbt ?t $\Longrightarrow$ *RBT.map-of* (*insrt ?k ?v ?t*) = *RBT.map-of ?t*(*?k* $\mapsto$ *?v*)

map-of-delete

isrbt ?t $\Longrightarrow$ *RBT.map-of* (*RBT.delete ?k ?t*) = *RBT.map-of ?t*$\restriction_{(-\ \{?k\})}$

map-of-union

$[\![$*isrbt ?s*; *st ?t*$]\!]$
$\Longrightarrow$ *RBT.map-of* (*union ?s ?t*) = *RBT.map-of ?s* ++ *RBT.map-of ?t*

map-of-map

RBT.map-of (*RBT.map ?f ?t*) = *Option.map ?f* $\circ$ *RBT.map-of ?t*

end

63 State-Monad: Combinator syntax for generic, open state monads (single threaded monads)

theory *State-Monad*
imports *Main*
begin

63.1 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.

63.2 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→* 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.

63.3 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*

63.4 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/ex/Random.thy.

end

64 Topology-Euclidean-Space: Elementary topology in Euclidean space.

theory *Topology-Euclidean-Space*
imports *SEQ Euclidean-Space*
begin

declare *fstcart-pastecart*[*simp*] *sndcart-pastecart*[*simp*]

64.1 General notion of a topology

definition *istopology L* $\longleftrightarrow$ $\{\} \in L \wedge (\forall S \in L.\ \forall T \in L.\ S \cap T \in L) \wedge (\forall K.\ K \subseteq L \longrightarrow \bigcup K \in L)$
typedef (**open**) *'a topology* = $\{L::('a\ set)\ set.\ istopology\ L\}$
morphisms *openin topology*
⟨*proof*⟩

lemma *istopology-open-in*[*intro*]: *istopology*(*openin U*)
⟨*proof*⟩

lemma *topology-inverse'*: *istopology U* $\Longrightarrow$ *openin* (*topology U*) = *U*
⟨*proof*⟩

lemma *topology-inverse-iff*: *istopology U* $\longleftrightarrow$ *openin* (*topology U*) = *U*
⟨*proof*⟩

lemma *topology-eq*: *T1* = *T2* $\longleftrightarrow$ $(\forall S.\ openin\ T1\ S \longleftrightarrow openin\ T2\ S)$
⟨*proof*⟩

Infer the "universe" from union of all sets in the topology.

definition *topspace T* = $\bigcup\{S.\ openin\ T\ S\}$

64.2 Main properties of open sets

lemma *openin-clauses*:
fixes *U* :: *'a topology*
shows *openin U* {}
$\bigwedge S\ T.\ openin\ U\ S \Longrightarrow openin\ U\ T \Longrightarrow openin\ U\ (S \cap T)$
$\bigwedge K.\ (\forall S \in K.\ openin\ U\ S) \Longrightarrow openin\ U\ (\bigcup K)$
⟨*proof*⟩

lemma *openin-subset*[*intro*]: *openin U S* $\Longrightarrow$ *S* $\subseteq$ *topspace U*
⟨*proof*⟩
lemma *openin-empty*[*simp*]: *openin U* {} ⟨*proof*⟩

lemma *openin-Int*[*intro*]: *openin U S* $\Longrightarrow$ *openin U T* $\Longrightarrow$ *openin U* $(S \cap T)$

⟨*proof*⟩

lemma *openin-Union*[*intro*]: (∀ *S* ∈*K*. *openin U S*) ⟹ *openin U* (⋃ *K*) ⟨*proof*⟩

lemma *openin-Un*[*intro*]: *openin U S* ⟹ *openin U T* ⟹ *openin U* (*S* ∪ *T*)
⟨*proof*⟩

lemma *openin-topspace*[*intro*, *simp*]: *openin U* (*topspace U*) ⟨*proof*⟩

lemma *openin-subopen*: *openin U S* ⟷ (∀ *x* ∈ *S*. ∃ *T*. *openin U T* ∧ *x* ∈ *T* ∧ *T* ⊆ *S*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

64.3 Closed sets

definition *closedin U S* ⟷ *S* ⊆ *topspace U* ∧ *openin U* (*topspace U* − *S*)

lemma *closedin-subset*: *closedin U S* ⟹ *S* ⊆ *topspace U* ⟨*proof*⟩
lemma *closedin-empty*[*simp*]: *closedin U* {} ⟨*proof*⟩
lemma *closedin-topspace*[*intro*,*simp*]:
closedin U (*topspace U*) ⟨*proof*⟩
lemma *closedin-Un*[*intro*]: *closedin U S* ⟹ *closedin U T* ⟹ *closedin U* (*S* ∪ *T*)
⟨*proof*⟩

lemma *Diff-Inter*[*intro*]: *A* − ⋂ *S* = ⋃ {*A* − *s*|*s*. *s*∈*S*} ⟨*proof*⟩
lemma *closedin-Inter*[*intro*]: **assumes** *Ke*: *K* ≠ {} **and** *Kc*: ∀ *S* ∈*K*. *closedin U S*
shows *closedin U* (⋂ *K*) ⟨*proof*⟩

lemma *closedin-Int*[*intro*]: *closedin U S* ⟹ *closedin U T* ⟹ *closedin U* (*S* ∩ *T*)
⟨*proof*⟩

lemma *Diff-Diff-Int*: *A* − (*A* − *B*) = *A* ∩ *B* ⟨*proof*⟩
lemma *openin-closedin-eq*: *openin U S* ⟷ *S* ⊆ *topspace U* ∧ *closedin U* (*topspace U* − *S*)
⟨*proof*⟩

lemma *openin-closedin*: *S* ⊆ *topspace U* ⟹ (*openin U S* ⟷ *closedin U* (*topspace U* − *S*))
⟨*proof*⟩

lemma *openin-diff*[*intro*]: **assumes** *oS*: *openin U S* **and** *cT*: *closedin U T* **shows** *openin U* (*S* − *T*)
⟨*proof*⟩

lemma *closedin-diff*[*intro*]: **assumes** *oS*: *closedin U S* **and** *cT*: *openin U T* **shows** *closedin U* (*S* − *T*)

⟨*proof*⟩

64.4 Subspace topology.

definition *subtopology U V* = *topology* {*S* ∩ *V* |*S*. *openin U S*}

lemma *istopology-subtopology*: *istopology* {*S* ∩ *V* |*S*. *openin U S*} (**is** *istopology ?L*)
⟨*proof*⟩

lemma *openin-subtopology*:
openin (*subtopology U V*) *S* ⟷ (∃ *T*. (*openin U T*) ∧ (*S* = *T* ∩ *V*))
⟨*proof*⟩

lemma *topspace-subtopology*: *topspace*(*subtopology U V*) = *topspace U* ∩ *V*
⟨*proof*⟩

lemma *closedin-subtopology*:
closedin (*subtopology U V*) *S* ⟷ (∃ *T*. *closedin U T* ∧ *S* = *T* ∩ *V*)
⟨*proof*⟩

lemma *openin-subtopology-refl*: *openin* (*subtopology U V*) *V* ⟷ *V* ⊆ *topspace U*
⟨*proof*⟩

lemma *subtopology-superset*: **assumes** *UV*: *topspace U* ⊆ *V*
shows *subtopology U V* = *U*
⟨*proof*⟩

lemma *subtopology-topspace*[*simp*]: *subtopology U* (*topspace U*) = *U*
⟨*proof*⟩

lemma *subtopology-UNIV*[*simp*]: *subtopology U UNIV* = *U*
⟨*proof*⟩

64.5 The universal Euclidean versions are what we use most of the time

definition *open S* ⟷ (∀ *x* ∈ *S*. ∃ *e* >*0*. ∀ *x'*. *dist x' x* < *e* ⟶ *x'* ∈ *S*)
definition *closed S* ⟷ *open*(*UNIV* − *S*)
definition *euclidean* = *topology open*

lemma *open-empty*[*intro*,*simp*]: *open* {} ⟨*proof*⟩
lemma *open-UNIV*[*intro*,*simp*]: *open UNIV*
⟨*proof*⟩

lemma *open-inter*[*intro*]: **assumes** *S*: *open S* **and** *T*: *open T*
shows *open* (*S* ∩ *T*)

⟨*proof*⟩

lemma *open-Union*[*intro*]: $(\forall S{\in}K.\ open\ S) \Longrightarrow open\ (\bigcup\ K)$
⟨*proof*⟩

lemma *open-openin*: $open\ S \longleftrightarrow openin\ euclidean\ S$
⟨*proof*⟩

lemma *topspace-euclidean*: $topspace\ euclidean = UNIV$
⟨*proof*⟩

lemma *topspace-euclidean-subtopology*[*simp*]: $topspace\ (subtopology\ euclidean\ S) = S$
⟨*proof*⟩

lemma *closed-closedin*: $closed\ S \longleftrightarrow closedin\ euclidean\ S$
⟨*proof*⟩

lemma *open-Un*[*intro*]: $open\ S \Longrightarrow open\ T \Longrightarrow open\ (S{\cup}T)$
⟨*proof*⟩

lemma *open-subopen*: $open\ S \longleftrightarrow (\forall x{\in}S.\ \exists T.\ open\ T \wedge x \in T \wedge T \subseteq S)$
⟨*proof*⟩

lemma *closed-empty*[*intro*, *simp*]: $closed\ \{\}$ ⟨*proof*⟩

lemma *closed-UNIV*[*simp*,*intro*]: $closed\ UNIV$
⟨*proof*⟩

lemma *closed-Un*[*intro*]: $closed\ S \Longrightarrow closed\ T \Longrightarrow closed\ (S{\cup}T)$
⟨*proof*⟩

lemma *closed-Int*[*intro*]: $closed\ S \Longrightarrow closed\ T \Longrightarrow closed\ (S{\cap}T)$
⟨*proof*⟩

lemma *closed-Inter*[*intro*]: **assumes** H: $\forall\ S\ {\in}K.\ closed\ S$ **shows** $closed\ (\bigcap K)$
⟨*proof*⟩

lemma *open-closed*: $open\ S \longleftrightarrow closed\ (UNIV\ -\ S)$
⟨*proof*⟩

lemma *closed-open*: $closed\ S \longleftrightarrow open(UNIV\ -\ S)$
⟨*proof*⟩

lemma *open-diff*[*intro*]: $open\ S \Longrightarrow closed\ T \Longrightarrow open\ (S\ -\ T)$
⟨*proof*⟩

lemma *closed-diff*[*intro*]: $closed\ S \Longrightarrow open\ T \Longrightarrow closed(S{-}T)$
⟨*proof*⟩

lemma *open-Inter*[*intro*]: **assumes** *fS*: *finite S* **and** *h*: $\forall\, T{\in}S.\ open\ T$ **shows** *open* $(\bigcap S)$
⟨*proof*⟩

lemma *closed-Union*[*intro*]: **assumes** *fS*: *finite S* **and** *h*: $\forall\, T{\in}S.\ closed\ T$ **shows** *closed* $(\bigcup S)$
⟨*proof*⟩

64.6 Open and closed balls.

definition *ball x e* = $\{y.\ dist\ x\ y < e\}$
definition *cball x e* = $\{y.\ dist\ x\ y \leq e\}$

lemma *mem-ball*[*simp*]: $y \in ball\ x\ e \longleftrightarrow dist\ x\ y < e$ ⟨*proof*⟩
lemma *mem-cball*[*simp*]: $y \in cball\ x\ e \longleftrightarrow dist\ x\ y \leq e$ ⟨*proof*⟩
lemma *mem-ball-0*[*simp*]: $x \in ball\ 0\ e \longleftrightarrow norm\ x < e$ ⟨*proof*⟩
lemma *mem-cball-0*[*simp*]: $x \in cball\ 0\ e \longleftrightarrow norm\ x \leq e$ ⟨*proof*⟩
lemma *centre-in-cball*[*simp*]: $x \in cball\ x\ e \longleftrightarrow 0 \leq e$ ⟨*proof*⟩
lemma *ball-subset-cball*[*simp*,*intro*]: $ball\ x\ e \subseteq cball\ x\ e$ ⟨*proof*⟩
lemma *subset-ball*[*intro*]: *d* <= *e* ==> $ball\ x\ d \subseteq ball\ x\ e$ ⟨*proof*⟩
lemma *subset-cball*[*intro*]: *d* <= *e* ==> $cball\ x\ d \subseteq cball\ x\ e$ ⟨*proof*⟩
lemma *ball-max-Un*: $ball\ a\ (max\ r\ s) = ball\ a\ r \cup ball\ a\ s$
⟨*proof*⟩

lemma *ball-min-Int*: $ball\ a\ (min\ r\ s) = ball\ a\ r \cap ball\ a\ s$
⟨*proof*⟩

64.7 Topological properties of open balls

lemma *diff-less-iff*: $(a{::}real) - b > 0 \longleftrightarrow a > b$
$(a{::}real) - b < 0 \longleftrightarrow a < b$
$a - b < c \longleftrightarrow a < c + b$ $a - b > c \longleftrightarrow a > c + b$ ⟨*proof*⟩
lemma *diff-le-iff*: $(a{::}real) - b \geq 0 \longleftrightarrow a \geq b$ $(a{::}real) - b \leq 0 \longleftrightarrow a \leq b$
$a - b \leq c \longleftrightarrow a \leq c + b$ $a - b \geq c \longleftrightarrow a \geq c + b$ ⟨*proof*⟩

lemma *open-ball*[*intro*, *simp*]: *open* (*ball x e*)
⟨*proof*⟩

lemma *centre-in-ball*[*simp*]: $x \in ball\ x\ e \longleftrightarrow e > 0$ ⟨*proof*⟩
lemma *open-contains-ball*: $open\ S \longleftrightarrow (\forall\, x{\in}S.\ \exists\, e{>}0.\ ball\ x\ e \subseteq S)$
⟨*proof*⟩

lemma *open-contains-ball-eq*: $open\ S \Longrightarrow \forall\, x.\ x{\in}S \longleftrightarrow (\exists\, e{>}0.\ ball\ x\ e \subseteq S)$
⟨*proof*⟩

lemma *ball-eq-empty*[*simp*]: $ball\ x\ e = \{\} \longleftrightarrow e \leq 0$
⟨*proof*⟩

lemma *ball-empty*[*intro*]: $e \leq 0$ ==> $ball\ x\ e = \{\}$ ⟨*proof*⟩

64.8 Basic "localization" results are handy for connectedness.

lemma *openin-open*: *openin* (*subtopology euclidean U*) *S* ⟷ (∃ *T*. *open T* ∧ (*S* = *U* ∩ *T*))
⟨*proof*⟩

lemma *openin-open-Int*[*intro*]: *open S* ⟹ *openin* (*subtopology euclidean U*) (*U* ∩ *S*)
⟨*proof*⟩

lemma *open-openin-trans*[*trans*]:
open S ⟹ *open T* ⟹ *T* ⊆ *S* ⟹ *openin* (*subtopology euclidean S*) *T*
⟨*proof*⟩

lemma *open-subset*: *S* ⊆ *T* ⟹ *open S* ⟹ *openin* (*subtopology euclidean T*) *S*
⟨*proof*⟩

lemma *closedin-closed*: *closedin* (*subtopology euclidean U*) *S* ⟷ (∃ *T*. *closed T* ∧ *S* = *U* ∩ *T*)
⟨*proof*⟩

lemma *closedin-closed-Int*: *closed S* ==> *closedin* (*subtopology euclidean U*) (*U* ∩ *S*)
⟨*proof*⟩

lemma *closed-closedin-trans*: *closed S* ⟹ *closed T* ⟹ *T* ⊆ *S* ⟹ *closedin* (*subtopology euclidean S*) *T*
⟨*proof*⟩

lemma *closed-subset*: *S* ⊆ *T* ⟹ *closed S* ⟹ *closedin* (*subtopology euclidean T*) *S*
⟨*proof*⟩

lemma *openin-euclidean-subtopology-iff*: *openin* (*subtopology euclidean U*) *S*
⟷ *S* ⊆ *U* ∧ (∀ *x*∈*S*. ∃ *e*>*0*. ∀ *x'*∈*U*. *dist x' x* < *e* ⟶ *x'*∈ *S*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

These "transitivity" results are handy too.

lemma *openin-trans*[*trans*]: *openin* (*subtopology euclidean T*) *S* ⟹ *openin* (*subtopology euclidean U*) *T*
⟹ *openin* (*subtopology euclidean U*) *S*
⟨*proof*⟩

lemma *openin-open-trans*: *openin* (*subtopology euclidean T*) *S* ⟹ *open T* ⟹ *open S*
⟨*proof*⟩

lemma *closedin-trans*[*trans*]:

closedin (*subtopology euclidean T*) *S* ⟹
closedin (*subtopology euclidean U*) *T*
==> *closedin* (*subtopology euclidean U*) *S*
⟨*proof*⟩

lemma *closedin-closed-trans*: *closedin* (*subtopology euclidean T*) *S* ⟹ *closed T* ⟹ *closed S*
⟨*proof*⟩

64.9 Connectedness

definition *connected S* ⟷
~(∃ *e1 e2*. *open e1* ∧ *open e2* ∧ *S* ⊆ (*e1* ∪ *e2*) ∧ (*e1* ∩ *e2* ∩ *S* = {})
∧ ~(*e1* ∩ *S* = {}) ∧ ~(*e2* ∩ *S* = {}))

lemma *connected-local*:
connected S ⟷ ~(∃ *e1 e2*.
openin (*subtopology euclidean S*) *e1* ∧
openin (*subtopology euclidean S*) *e2* ∧
S ⊆ *e1* ∪ *e2* ∧
e1 ∩ *e2* = {} ∧
~(*e1* = {}) ∧
~(*e2* = {}))
⟨*proof*⟩

lemma *exists-diff*: (∃ *S*. *P*(*UNIV* − *S*)) ⟷ (∃ *S*. *P S*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *connected-clopen*: *connected S* ⟷
(∀ *T*. *openin* (*subtopology euclidean S*) *T* ∧
closedin (*subtopology euclidean S*) *T* ⟶ *T* = {} ∨ *T* = *S*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *connected-empty*[*simp*, *intro*]: *connected* {}
⟨*proof*⟩

64.10 Hausdorff and other separation properties

lemma *hausdorff*:
assumes *xy*: *x* ≠ *y*
shows ∃ *U V*. *open U* ∧ *open V* ∧ *x*∈ *U* ∧ *y* ∈ *V* ∧ (*U* ∩ *V* = {}) (**is** ∃ *U V*. *?P U V*)
⟨*proof*⟩

lemma *separation-t2*: *x* ≠ *y* ⟷ (∃ *U V*. *open U* ∧ *open V* ∧ *x* ∈ *U* ∧ *y* ∈ *V* ∧ *U* ∩ *V* = {})
⟨*proof*⟩

lemma *separation-t1*: $x \neq y \longleftrightarrow (\exists\, U\ V.\ open\ U \wedge open\ V \wedge x \in U \wedge y \notin U \wedge x \notin V \wedge y \in V)$
⟨*proof*⟩

lemma *separation-t0*: $x \neq y \longleftrightarrow (\exists\, U.\ open\ U \wedge \sim(x \in U \longleftrightarrow y \in U))$ ⟨*proof*⟩

64.11 Limit points

definition *islimpt*:: *real ^'n::finite* ⇒ (*real^'n*) *set* ⇒ *bool* (**infixr** *islimpt* 60) **where**
islimpt-def: *x islimpt S* ⟷ (∀ *T*. *x*∈*T* ⟶ *open T* ⟶ (∃ *y*∈*S*. *y*∈*T* ∧ *y*≠*x*))

lemma *islimptE*: **assumes** *x islimpt S* **and** $x \in T$ **and** *open T*
obtains (∃ *y*∈*S*. *y*∈*T* ∧ *y*≠*x*)
⟨*proof*⟩

lemma *islimpt-subset*: *x islimpt S* ⟹ *S* ⊆ *T* ==> *x islimpt T* ⟨*proof*⟩
lemma *islimpt-approachable*: *x islimpt S* ⟷ (∀ *e*>0. ∃ *x'*∈*S*. *x'* ≠ *x* ∧ *dist x' x* < *e*)
⟨*proof*⟩

lemma *islimpt-approachable-le*: *x islimpt S* ⟷ (∀ *e*>0. ∃ *x'*∈ *S*. *x'* ≠ *x* ∧ *dist x'* *x* <= *e*)
⟨*proof*⟩

lemma *islimpt-UNIV*[*simp*, *intro*]: (*x*:: *real ^'n::finite*) *islimpt UNIV*
⟨*proof*⟩

lemma *closed-limpt*: *closed S* ⟷ (∀ *x*. *x islimpt S* ⟶ *x* ∈ *S*)
⟨*proof*⟩

lemma *islimpt-EMPTY*[*simp*]: ¬ *x islimpt* {}
⟨*proof*⟩

lemma *closed-positive-orthant*: *closed* {*x*::*real^'n*::*finite*. ∀ *i*. 0 ≤*x*$*i*}
⟨*proof*⟩

lemma *finite-set-avoid*: **assumes** *fS*: *finite S* **shows** ∃ *d*>0. ∀ *x*∈*S*. *x* ≠ *a* ⟶ *d* <= *dist a x*
⟨*proof*⟩

lemma *islimpt-finite*: **assumes** *fS*: *finite S* **shows** ¬ *a islimpt S*
⟨*proof*⟩

lemma *islimpt-Un*: *x islimpt* (*S* ∪ *T*) ⟷ *x islimpt S* ∨ *x islimpt T*
⟨*proof*⟩

lemma *discrete-imp-closed*:

assumes *e*: $0 < e$ **and** *d*: $\forall\, x \in S.\ \forall\, y \in S.\ norm(y - x) < e \longrightarrow y = x$
shows *closed S*
⟨*proof*⟩

64.12 Interior of a Set

definition $interior\ S = \{x.\ \exists\, T.\ open\ T \wedge x \in T \wedge T \subseteq S\}$

lemma *interior-eq*: $interior\ S = S \longleftrightarrow open\ S$
⟨*proof*⟩

lemma *interior-open*: $open\ S \Longrightarrow (interior\ S = S)$ ⟨*proof*⟩

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

lemma *open-interior*[*simp*, *intro*]: $open(interior\ S)$
⟨*proof*⟩

lemma *interior-interior*[*simp*]: $interior(interior\ S) = interior\ S$ ⟨*proof*⟩
lemma *interior-subset*: $interior\ S \subseteq S$ ⟨*proof*⟩
lemma *subset-interior*: $S \subseteq T \Longrightarrow (interior\ S) \subseteq (interior\ T)$ ⟨*proof*⟩
lemma *interior-maximal*: $T \subseteq S \Longrightarrow open\ T \Longrightarrow T \subseteq (interior\ S)$ ⟨*proof*⟩
lemma *interior-unique*: $T \subseteq S \Longrightarrow open\ T \Longrightarrow (\forall\, T'.\ T' \subseteq S \wedge open\ T' \longrightarrow T' \subseteq T) \Longrightarrow interior\ S = T$
⟨*proof*⟩
lemma *mem-interior*: $x \in interior\ S \longleftrightarrow (\exists\, e.\ 0 < e \wedge ball\ x\ e \subseteq S)$
⟨*proof*⟩

lemma *open-subset-interior*: $open\ S \Longrightarrow S \subseteq interior\ T \longleftrightarrow S \subseteq T$
⟨*proof*⟩

lemma *interior-inter*[*simp*]: $interior(S \cap T) = interior\ S \cap interior\ T$
⟨*proof*⟩

lemma *interior-limit-point*[*intro*]: **assumes** *x*: $x \in interior\ S$ **shows** *x islimpt S*
⟨*proof*⟩

lemma *interior-closed-Un-empty-interior*:
assumes *cS*: *closed S* **and** *iT*: $interior\ T = \{\}$
shows $interior(S \cup T) = interior\ S$
⟨*proof*⟩

64.13 Closure of a Set

definition $closure\ S = S \cup \{x \mid x.\ x\ islimpt\ S\}$

lemma *closure-interior*: $closure\ S = UNIV - interior\ (UNIV - S)$
⟨*proof*⟩

lemma *interior-closure*: $interior\ S = UNIV - (closure\ (UNIV - S))$

⟨*proof*⟩

lemma *closed-closure*[*simp*, *intro*]: *closed* (*closure S*)
⟨*proof*⟩

lemma *closure-hull*: *closure S* = *closed hull S*
⟨*proof*⟩

lemma *closure-eq*: *closure S* = *S* ⟷ *closed S*
⟨*proof*⟩

lemma *closure-closed*[*simp*]: *closed S* ⟹ *closure S* = *S*
⟨*proof*⟩

lemma *closure-closure*[*simp*]: *closure* (*closure S*) = *closure S*
⟨*proof*⟩

lemma *closure-subset*: *S* ⊆ *closure S*
⟨*proof*⟩

lemma *subset-closure*: *S* ⊆ *T* ⟹ *closure S* ⊆ *closure T*
⟨*proof*⟩

lemma *closure-minimal*: *S* ⊆ *T* ⟹ *closed T* ⟹ *closure S* ⊆ *T*
⟨*proof*⟩

lemma *closure-unique*: *S* ⊆ *T* ∧ *closed T* ∧ (∀ *T'*. *S* ⊆ *T'* ∧ *closed T'* ⟶ *T* ⊆ *T'*) ⟹ *closure S* = *T*
⟨*proof*⟩

lemma *closure-empty*[*simp*]: *closure* {} = {}
⟨*proof*⟩

lemma *closure-univ*[*simp*]: *closure UNIV* = *UNIV*
⟨*proof*⟩

lemma *closure-eq-empty*: *closure S* = {} ⟷ *S* = {}
⟨*proof*⟩

lemma *closure-subset-eq*: *closure S* ⊆ *S* ⟷ *closed S*
⟨*proof*⟩

lemma *open-inter-closure-eq-empty*:
open S ⟹ (*S* ∩ *closure T*) = {} ⟷ *S* ∩ *T* = {}
⟨*proof*⟩

lemma *open-inter-closure-subset*: *open S* ⟹ (*S* ∩ (*closure T*)) ⊆ *closure*(*S* ∩ *T*)
⟨*proof*⟩

lemma *closure-complement*: $closure(UNIV - S) = UNIV - interior(S)$
⟨*proof*⟩

lemma *interior-complement*: $interior(UNIV - S) = UNIV - closure(S)$
⟨*proof*⟩

64.14 Frontier (aka boundary)

definition $frontier\ S = closure\ S - interior\ S$

lemma *frontier-closed*: $closed(frontier\ S)$
⟨*proof*⟩

lemma *frontier-closures*: $frontier\ S = (closure\ S) \cap (closure(UNIV - S))$
⟨*proof*⟩

lemma *frontier-straddle*: $a \in frontier\ S \longleftrightarrow (\forall e{>}0.\ (\exists x{\in}S.\ dist\ a\ x < e) \wedge (\exists x.\ x \notin S \wedge dist\ a\ x < e))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *frontier-subset-closed*: $closed\ S \Longrightarrow frontier\ S \subseteq S$
⟨*proof*⟩

lemma *frontier-empty*: $frontier\ \{\} = \{\}$
⟨*proof*⟩

lemma *frontier-subset-eq*: $frontier\ S \subseteq S \longleftrightarrow closed\ S$
⟨*proof*⟩

lemma *frontier-complement*: $frontier(UNIV - S) = frontier\ S$
⟨*proof*⟩

lemma *frontier-disjoint-eq*: $frontier\ S \cap S = \{\} \longleftrightarrow open\ S$
⟨*proof*⟩

64.15 A variant of nets (Slightly non-standard but good for our purposes).

typedef (**open**) $'a\ net =$
$\{g :: 'a \Rightarrow 'a \Rightarrow bool.\ \forall x\ y.\ (\forall z.\ g\ z\ x \longrightarrow g\ z\ y) \vee (\forall z.\ g\ z\ y \longrightarrow g\ z\ x)\}$
morphisms *netord mknet* ⟨*proof*⟩
lemma *net*: $(\forall z.\ netord\ n\ z\ x \longrightarrow netord\ n\ z\ y) \vee (\forall z.\ netord\ n\ z\ y \longrightarrow netord\ n\ z\ x)$
⟨*proof*⟩

lemma *oldnet*: $netord\ n\ x\ x \Longrightarrow netord\ n\ y\ y \Longrightarrow$
$\exists z.\ netord\ n\ z\ z \wedge (\forall w.\ netord\ n\ w\ z \longrightarrow netord\ n\ w\ x \wedge netord\ n\ w\ y)$
⟨*proof*⟩

lemma *net-dilemma*:
$\exists\, a.\ (\exists\, x.\ \mathit{netord}\ \mathit{net}\ x\ a) \wedge (\forall\, x.\ \mathit{netord}\ \mathit{net}\ x\ a \longrightarrow P\ x) \Longrightarrow$
$\exists\, b.\ (\exists\, x.\ \mathit{netord}\ \mathit{net}\ x\ b) \wedge (\forall\, x.\ \mathit{netord}\ \mathit{net}\ x\ b \longrightarrow Q\ x)$
$\Longrightarrow \exists\, c.\ (\exists\, x.\ \mathit{netord}\ \mathit{net}\ x\ c) \wedge (\forall\, x.\ \mathit{netord}\ \mathit{net}\ x\ c \longrightarrow P\ x \wedge Q\ x)$
⟨*proof*⟩

64.16 Common nets and The "within" modifier for nets.

definition *at a* = *mknet*($\lambda x\ y.\ 0 < \mathit{dist}\ x\ a \wedge \mathit{dist}\ x\ a <= \mathit{dist}\ y\ a$)
definition *at-infinity* = *mknet*($\lambda x\ y.\ \mathit{norm}\ x \geq \mathit{norm}\ y$)
definition *sequentially* = *mknet*($\lambda(m{::}\mathit{nat})\ n.\ m >= n$)

definition *within* :: *'a net* ⇒ *'a set* ⇒ *'a net* (**infixr** *within 70*) **where**
within-def: *net within S* = *mknet* ($\lambda x\ y.\ \mathit{netord}\ \mathit{net}\ x\ y \wedge x \in S$)

definition *indirection* :: *real ^'n::finite* ⇒ *real ^'n* ⇒ (*real ^'n*) *net* (**infixr** *indirection 70*) **where**
indirection-def: *a indirection v* = (*at a*) *within* $\{b.\ \exists\, c{\geq}0.\ b - a = c{*s}\ v\}$

Prove That They are all nets.

lemma *mknet-inverse'*: *netord* (*mknet r*) = *r* ⟷ $(\forall\, x\ y.\ (\forall\, z.\ r\ z\ x \longrightarrow r\ z\ y) \vee (\forall\, z.\ r\ z\ y \longrightarrow r\ z\ x))$
⟨*proof*⟩

⟨*ML*⟩

lemma *at*: $\bigwedge x\ y.$ *netord* (*at a*) *x y* ⟷ *0* < *dist x a* ∧ *dist x a* <= *dist y a*
⟨*proof*⟩

lemma *at-infinity*:
$\bigwedge x\ y.$ *netord at-infinity x y* ⟷ *norm x* >= *norm y*
⟨*proof*⟩

lemma *sequentially*: $\bigwedge m\ n.$ *netord sequentially m n* ⟷ *m* >= *n*
⟨*proof*⟩

lemma *within*: *netord* (*n within S*) *x y* ⟷ *netord n x y* ∧ *x* ∈ *S*
⟨*proof*⟩

lemma *in-direction*: *netord* (*a indirection v*) *x y* ⟷ *0* < *dist x a* ∧ *dist x a* ≤ *dist y a* ∧ ($\exists\, c \geq 0.\ x - a = c *s\ v$)
⟨*proof*⟩

lemma *within-UNIV*: *at x within UNIV* = *at x*
⟨*proof*⟩

64.17 Identify Trivial limits, where we can't approach arbitrarily closely.

definition *trivial-limit* (*net*:: *'a net*) ⟷

$(\forall\,(a::'a)\ b.\ a = b) \vee (\exists\,(a::'a)\ b.\ a \neq b \wedge (\forall\,x.\ \sim(netord\ (net)\ x\ a) \wedge \sim(netord(net)\ x\ b)))$

lemma *trivial-limit-within*: *trivial-limit* $(at\ (a::real\hat{}'n::finite)\ within\ S) \longleftrightarrow \sim(a\ islimpt\ S)$
⟨*proof*⟩

lemma *trivial-limit-at*: $\sim(trivial\text{-}limit\ (at\ a))$
⟨*proof*⟩

lemma *trivial-limit-at-infinity*: $\sim(trivial\text{-}limit\ (at\text{-}infinity :: ('a::\{norm,zero\text{-}neq\text{-}one\}))\ net))$
⟨*proof*⟩

lemma *trivial-limit-sequentially*: $\sim(trivial\text{-}limit\ sequentially)$
⟨*proof*⟩

64.18 Some property holds "sufficiently close" to the limit point.

definition $eventually\ P\ net \longleftrightarrow trivial\text{-}limit\ net \vee (\exists\,y.\ (\exists\,x.\ netord\ net\ x\ y) \wedge (\forall\,x.\ netord\ net\ x\ y \longrightarrow P\ x))$

lemma *eventually-happens*: $eventually\ P\ net ==> trivial\text{-}limit\ net \vee (\exists\,x.\ P\ x)$
⟨*proof*⟩

lemma *eventually-within-le*: $eventually\ P\ (at\ a\ within\ S) \longleftrightarrow$
$(\exists\,d{>}0.\ \forall\,x{\in}S.\ 0 < dist\ x\ a \wedge dist\ x\ a <= d \longrightarrow P\ x)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *eventually-within*: $eventually\ P\ (at\ a\ within\ S) \longleftrightarrow$
$(\exists\,d{>}0.\ \forall\,x{\in}S.\ 0 < dist\ x\ a \wedge dist\ x\ a < d \longrightarrow P\ x)$
⟨*proof*⟩

lemma *eventually-at*: $eventually\ P\ (at\ a) \longleftrightarrow (\exists\,d{>}0.\ \forall\,x.\ 0 < dist\ x\ a \wedge dist\ x\ a < d \longrightarrow P\ x)$
⟨*proof*⟩

lemma *eventually-sequentially*: $eventually\ P\ sequentially \longleftrightarrow (\exists\,N.\ \forall\,n{\geq}N.\ P\ n)$
⟨*proof*⟩

lemma *eventually-at-infinity*: $eventually\ (P::(real\hat{}'n::finite \Rightarrow bool))\ at\text{-}infinity \longleftrightarrow (\exists\,b.\ \forall\,x.\ norm\ x >= b \longrightarrow P\ x)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *always-eventually*: $(\forall\,(x::'a::zero\text{-}neq\text{-}one).\ P\ x) ==> eventually\ P\ net$
⟨*proof*⟩

Combining theorems for "eventually"

lemma *eventually-and*: *eventually* $(\lambda x.\ P\ x \wedge Q\ x)$ *net* $\longleftrightarrow$ *eventually* P *net* $\wedge$ *eventually* Q *net*
⟨*proof*⟩

lemma *eventually-mono*: $(\forall x.\ P\ x \longrightarrow Q\ x) \Longrightarrow$ *eventually* P *net* $\Longrightarrow$ *eventually* Q *net*
⟨*proof*⟩

lemma *eventually-mp*: *eventually* $(\lambda x.\ P\ x \longrightarrow Q\ x)$ *net* $\Longrightarrow$ *eventually* P *net* $\Longrightarrow$ *eventually* Q *net*
⟨*proof*⟩

lemma *eventually-false*: *eventually* $(\lambda x.\ False)$ *net* $\longleftrightarrow$ *trivial-limit net*
⟨*proof*⟩

lemma *not-eventually*: $(\forall x.\ \neg\ P\ x\ ) \Longrightarrow$ ~(*trivial-limit net*) ==> ~(*eventually* P *net*)
⟨*proof*⟩

64.19 Limits, defined as vacuously true when the limit is trivial.

definition *tendsto*:: $('a \Rightarrow$ *real* ^$'n$::*finite*$) \Rightarrow$ *real* ^$'n \Rightarrow\ 'a$ *net* $\Rightarrow$ *bool* (**infixr** −−−> *55*) **where**
tendsto-def: $(f$ −−−> $l)$ *net* $\longleftrightarrow (\forall e{>}0.$ *eventually* $(\lambda x.$ *dist* $(f\ x)\ l < e)$ *net*)

lemma *tendstoD*: $(f$ −−−> $l)$ *net* $\Longrightarrow e{>}0 \Longrightarrow$ *eventually* $(\lambda x.$ *dist* $(f\ x)\ l < e)$ *net*
⟨*proof*⟩

Notation Lim to avoid collition with lim defined in analysis

definition *Lim net* f = $(THE\ l.\ (f$ −−−> $l)$ *net*)

lemma *Lim*:
$(f$ −−−> $l)$ *net* $\longleftrightarrow$
trivial-limit net $\vee$
$(\forall e{>}0.\ \exists y.\ (\exists x.$ *netord net* $x\ y) \wedge$
$(\forall x.$ *netord*$(net)\ x\ y \longrightarrow$ *dist* $(f\ x)\ l < e))$
⟨*proof*⟩

Show that they yield usual definitions in the various cases.

lemma *Lim-within-le*: $(f$ −−−> $l)$(*at a within S*) $\longleftrightarrow$
$(\forall e{>}0.\ \exists d{>}0.\ \forall x{\in}S.\ 0 < dist\ x\ a\ \wedge\ dist\ x\ a\ <= d \longrightarrow$ *dist* $(f\ x)\ l < e)$
⟨*proof*⟩

lemma *Lim-within*: $(f$ −−−> $l)$ (*at a within S*) $\longleftrightarrow$
$(\forall e > 0.\ \exists d{>}0.\ \forall x \in S.\ 0 < dist\ x\ a\ \wedge dist\ x\ a\ < d\ \longrightarrow$ *dist* $(f\ x)\ l < e)$

⟨*proof*⟩

lemma *Lim-at*: $(f \longrightarrow l)\ (at\ a) \longleftrightarrow$
$(\forall e>0.\ \exists d>0.\ \forall x.\ 0 < dist\ x\ a \ \wedge\ dist\ x\ a < d \longrightarrow dist\ (f\ x)\ l < e)$
⟨*proof*⟩

lemma *Lim-at-infinity*:
$(f \longrightarrow l)\ at\text{-}infinity \longleftrightarrow (\forall e>0.\ \exists b.\ \forall x::real\hat{}'n::finite.\ norm\ x >= b \longrightarrow dist\ (f\ x)\ l < e)$
⟨*proof*⟩

lemma *Lim-sequentially*:
$(S \longrightarrow l)\ sequentially \longleftrightarrow$
$(\forall e>0.\ \exists N.\ \forall n\geq N.\ dist\ (S\ n)\ l < e)$
⟨*proof*⟩

lemma *Lim-eventually*: $eventually\ (\lambda x.\ f\ x = l)\ net \Longrightarrow (f \longrightarrow l)\ net$
⟨*proof*⟩

The expected monotonicity property.

lemma *Lim-within-empty*: $(f \longrightarrow l)\ (at\ x\ within\ \{\})$
⟨*proof*⟩

lemma *Lim-within-subset*: $(f \longrightarrow l)\ (at\ a\ within\ S) \Longrightarrow T \subseteq S \Longrightarrow (f \longrightarrow l)\ (at\ a\ within\ T)$
⟨*proof*⟩

lemma *Lim-Un*: **assumes** $(f \longrightarrow l)\ (at\ x\ within\ S)\ (f \longrightarrow l)\ (at\ x\ within\ T)$
shows $(f \longrightarrow l)\ (at\ x\ within\ (S \cup T))$
⟨*proof*⟩

lemma *Lim-Un-univ*:
$(f \longrightarrow l)\ (at\ x\ within\ S) \Longrightarrow (f \longrightarrow l)\ (at\ x\ within\ T) \Longrightarrow S \cup T = (UNIV::(real\hat{}'n::finite)\ set)$
$==> (f \longrightarrow l)\ (at\ x)$
⟨*proof*⟩

Interrelations between restricted and unrestricted limits.

lemma *Lim-at-within*: $(f \longrightarrow l)(at\ a) ==> (f \longrightarrow l)(at\ a\ within\ S)$
⟨*proof*⟩

lemma *Lim-within-open*:
assumes $a \in S\ open\ S$
shows $(f \longrightarrow l)(at\ a\ within\ S) \longleftrightarrow (f \longrightarrow l)(at\ a)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

Another limit point characterization.

lemma *islimpt-sequential*:

x islimpt S ⟷ (∃*f*. (∀ *n*::*nat*. *f n* ∈ *S* −{*x*}) ∧ (*f* −−−> *x*) *sequentially*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

Basic arithmetical combining theorems for limits.

lemma *Lim-linear*: **fixes** *f* :: ('*a* ⇒ *real^'n*::*finite*) **and** *h* :: (*real^'n* ⇒ *real^'m*::*finite*)
assumes (*f* −−−> *l*) *net* *linear h*
shows ((λ*x*. *h* (*f x*)) −−−> *h l*) *net*
⟨*proof*⟩

lemma *Lim-const*: ((λ*x*. *a*) −−−> *a*) *net*
⟨*proof*⟩

lemma *Lim-cmul*: (*f* −−−> *l*) *net* ==> ((λ*x*. *c* ∗*s f x*) −−−> *c* ∗*s l*) *net*
⟨*proof*⟩

lemma *Lim-neg*: (*f* −−−> *l*) *net* ==> ((λ*x*. −(*f x*)) −−−> −*l*) *net*
⟨*proof*⟩

lemma *Lim-add*: **fixes** *f* :: '*a* ⇒ *real^'n*::*finite* **shows**
(*f* −−−> *l*) *net* ⟹ (*g* −−−> *m*) *net* ⟹ ((λ*x*. *f*(*x*) + *g*(*x*)) −−−> *l* + *m*) *net*
⟨*proof*⟩

lemma *Lim-sub*: (*f* −−−> *l*) *net* ⟹ (*g* −−−> *m*) *net* ⟹ ((λ*x*. *f*(*x*) − *g*(*x*)) −−−> *l* − *m*) *net*
⟨*proof*⟩

lemma *Lim-null*: (*f* −−−> *l*) *net* ⟷ ((λ*x*. *f*(*x*) − *l*) −−−> *0*) *net* ⟨*proof*⟩
lemma *Lim-null-norm*: (*f* −−−> *0*) *net* ⟷ ((λ*x*. *vec1*(*norm*(*f x*))) −−−> *0*) *net*
⟨*proof*⟩

lemma *Lim-null-comparison*:
assumes *eventually* (λ*x*. *norm*(*f x*) <= *g x*) *net* ((λ*x*. *vec1*(*g x*)) −−−> *0*) *net*
shows (*f* −−−> *0*) *net*
⟨*proof*⟩

lemma *Lim-component*: (*f* −−−> *l*) *net*
==> ((λ*a*. *vec1*((*f a* :: *real ^'n*::*finite*)$*i*)) −−−> *vec1*(*l*$*i*)) *net*
⟨*proof*⟩

lemma *Lim-transform-bound*:
assumes *eventually* (λ*n*. *norm*(*f n*) <= *norm*(*g n*)) *net* (*g* −−−> *0*) *net*
shows (*f* −−−> *0*) *net*
⟨*proof*⟩

Deducing things about the limit from the elements.

lemma *Lim-in-closed-set*:

assumes *closed S eventually* ($\lambda x.\ f(x) \in S$) *net* ¬(*trivial-limit net*) (*f* ---> *l*) *net*
shows $l \in S$
⟨*proof*⟩

Need to prove closed(cball(x,e)) before deducing this as a corollary.

lemma *Lim-norm-ubound*:
assumes ¬(*trivial-limit net*) (*f* ---> *l*) *net eventually* ($\lambda x.$ *norm*(*f x*) <= *e*) *net*
shows *norm*(*l*) <= *e*
⟨*proof*⟩

lemma *Lim-norm-lbound*:
assumes ¬ (*trivial-limit net*) (*f* ---> *l*) *net eventually* ($\lambda x.$ *e* <= *norm*(*f x*)) *net*
shows $e \leq$ *norm l*
⟨*proof*⟩

Uniqueness of the limit, when nontrivial.

lemma *Lim-unique*:
fixes *l*::*real^'a*::*finite* **and** *net*::*'b*::*zero-neq-one net*
assumes ¬(*trivial-limit net*) (*f* ---> *l*) *net* (*f* ---> *l'*) *net*
shows $l = l'$
⟨*proof*⟩

lemma *tendsto-Lim*:
~(*trivial-limit* (*net*::(*'b*::*zero-neq-one net*))) ⟹ (*f* ---> *l*) *net* ==> *Lim net f* = *l*
⟨*proof*⟩

Limit under bilinear function (surprisingly tedious, but important)

lemma *norm-bound-lemma*:
$0 < e \Longrightarrow \exists\, d{>}0.\ \forall (x'{::}real\hat{}'b{::}finite)\ y'{::}real\hat{}'a{::}finite.\ norm(x' - (x{::}real\hat{}'b)) < d \wedge norm(y' - y) < d \longrightarrow norm(x') * norm(y' - y) + norm(x' - x) * norm(y) < e$
⟨*proof*⟩

lemma *Lim-bilinear*:
fixes *net* :: *'a net* **and** *h*:: *real ^'m*::*finite* ⇒ *real ^'n*::*finite* ⇒ *real ^'p*::*finite*
assumes (*f* ---> *l*) *net* **and** (*g* ---> *m*) *net* **and** *bilinear h*
shows (($\lambda x.$ *h* (*f x*) (*g x*)) ---> (*h l m*)) *net*
⟨*proof*⟩

These are special for limits out of the same vector space.

lemma *Lim-within-id*: (*id* ---> *a*) (*at a within s*) ⟨*proof*⟩
lemma *Lim-at-id*: (*id* ---> *a*) (*at a*)
⟨*proof*⟩

lemma *Lim-at-zero*: (*f* ---> *l*) (*at* (*a*::*real^'a*::*finite*)) ⟷ (($\lambda x.$ *f*(*a* + *x*)) ---> *l*) (*at 0*) (**is** *?lhs* = *?rhs*)

⟨*proof*⟩

It's also sometimes useful to extract the limit point from the net.

definition *netlimit net* = (*SOME a*. ∀ *x*. ~(*netord net x a*))

lemma *netlimit-within*: **assumes**~(*trivial-limit* (*at a within S*))
shows (*netlimit* (*at a within S*) = *a*)
⟨*proof*⟩

lemma *netlimit-at*: *netlimit*(*at a*) = *a*
⟨*proof*⟩

Transformation of limit.

lemma *Lim-transform*: **assumes** ((λ*x*. *f x* − *g x*) −−−> *0*) *net* (*f* −−−> *l*) *net*
shows (*g* −−−> *l*) *net*
⟨*proof*⟩

lemma *Lim-transform-eventually*: *eventually* (λ*x*. *f x* = *g x*) *net* ⟹ (*f* −−−> *l*) *net* ==> (*g* −−−> *l*) *net*
⟨*proof*⟩

lemma *Lim-transform-within*:
assumes *0* < *d* (∀ *x'*∈*S*. *0* < *dist x' x* ∧ *dist x' x* < *d* ⟶ *f x'* = *g x'*)
(*f* −−−> *l*) (*at x within S*)
shows (*g* −−−> *l*) (*at x within S*)
⟨*proof*⟩

lemma *Lim-transform-at*: *0* < *d* ⟹ (∀ *x'*. *0* < *dist x' x* ∧ *dist x' x* < *d* ⟶ *f x'* = *g x'*) ⟹
(*f* −−−> *l*) (*at x*) ==> (*g* −−−> *l*) (*at x*)
⟨*proof*⟩

Common case assuming being away from some crucial point like 0.

lemma *Lim-transform-away-within*:
fixes *f*:: *real* ^'*m*::*finite* ⇒ *real* ^'*n*::*finite*
assumes *a*≠*b* ∀ *x*∈ *S*. *x* ≠ *a* ∧ *x* ≠ *b* ⟶ *f x* = *g x*
and (*f* −−−> *l*) (*at a within S*)
shows (*g* −−−> *l*) (*at a within S*)
⟨*proof*⟩

lemma *Lim-transform-away-at*:
fixes *f*:: *real* ^'*m*::*finite* ⇒ *real* ^'*n*::*finite*
assumes *ab*: *a*≠*b* **and** *fg*: ∀ *x*. *x* ≠ *a* ∧ *x* ≠ *b* ⟶ *f x* = *g x*
and *fl*: (*f* −−−> *l*) (*at a*)
shows (*g* −−−> *l*) (*at a*)
⟨*proof*⟩

Alternatively, within an open set.

lemma *Lim-transform-within-open*:

fixes f:: *real ^'m::finite ⇒ real ^'n::finite*
assumes *open S* $a \in S$ $\forall x{\in}S.\ x \neq a \longrightarrow f\ x = g\ x$ $(f \longrightarrow l)\ (at\ a)$
shows $(g \longrightarrow l)\ (at\ a)$
⟨*proof*⟩

A congruence rule allowing us to transform limits assuming not at point.

lemma *Lim-cong-within*[*cong add*]:
$(\bigwedge x.\ x \neq a \Longrightarrow f\ x = g\ x) ==> ((\lambda x.\ f\ x) \longrightarrow l)\ (at\ a\ within\ S) \longleftrightarrow ((g \longrightarrow l)\ (at\ a\ within\ S))$
⟨*proof*⟩

lemma *Lim-cong-at*[*cong add*]:
$(\bigwedge x.\ x \neq a ==> f\ x = g\ x) ==> (((\lambda x.\ f\ x) \longrightarrow l)\ (at\ a) \longleftrightarrow ((g \longrightarrow l)\ (at\ a)))$
⟨*proof*⟩

Useful lemmas on closure and set of possible sequential limits.

lemma *closure-sequential*:
$l \in closure\ S \longleftrightarrow (\exists x.\ (\forall n.\ x\ n \in S) \wedge (x \longrightarrow l)\ sequentially)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *closed-sequential-limits*:
$closed\ S \longleftrightarrow (\forall x\ l.\ (\forall n.\ x\ n \in S) \wedge (x \longrightarrow l)\ sequentially \longrightarrow l \in S)$
⟨*proof*⟩

lemma *closure-approachable*: $x \in closure\ S \longleftrightarrow (\forall e{>}0.\ \exists y{\in}S.\ dist\ y\ x < e)$
⟨*proof*⟩

lemma *closed-approachable*: $closed\ S ==> (\forall e{>}0.\ \exists y{\in}S.\ dist\ y\ x < e) \longleftrightarrow x \in S$
⟨*proof*⟩

Some other lemmas about sequences.

lemma *seq-offset*: $(f \longrightarrow l)\ sequentially ==> ((\lambda i.\ f(\ i + k)) \longrightarrow l)\ sequentially$
⟨*proof*⟩

lemma *seq-offset-neg*: $(f \longrightarrow l)\ sequentially ==> ((\lambda i.\ f(i - k)) \longrightarrow l)\ sequentially$
⟨*proof*⟩

lemma *seq-offset-rev*: $((\lambda i.\ f(i + k)) \longrightarrow l)\ sequentially ==> (f \longrightarrow l)\ sequentially$
⟨*proof*⟩

lemma *seq-harmonic*: $((\lambda n.\ vec1(inverse\ (real\ n))) \longrightarrow 0)\ sequentially$
⟨*proof*⟩

More properties of closed balls.

lemma *closed-cball*: *closed*(*cball x e*)
⟨*proof*⟩

lemma *open-contains-cball*: *open S* ⟷ (∀ *x*∈*S*. ∃ *e*>*0*. *cball x e* ⊆ *S*)
⟨*proof*⟩

lemma *open-contains-cball-eq*: *open S* ==> (∀ *x*. *x* ∈ *S* ⟷ (∃ *e*>*0*. *cball x e* ⊆ *S*))
⟨*proof*⟩

lemma *mem-interior-cball*: *x* ∈ *interior S* ⟷ (∃ *e*>*0*. *cball x e* ⊆ *S*)
⟨*proof*⟩

lemma *islimpt-ball*: *y islimpt ball x e* ⟷ *0* < *e* ∧ *y* ∈ *cball x e* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *closure-ball*: *0* < *e* ==> (*closure*(*ball x e*) = *cball x e*)
⟨*proof*⟩

lemma *interior-cball*: *interior*(*cball x e*) = *ball x e*
⟨*proof*⟩

lemma *frontier-ball*: *0* < *e* ==> *frontier*(*ball a e*) = {*x*. *dist a x* = *e*}
⟨*proof*⟩

lemma *frontier-cball*: *frontier*(*cball a e*) = {*x*. *dist a x* = *e*}
⟨*proof*⟩

lemma *cball-eq-empty*: (*cball x e* = {}) ⟷ *e* < *0*
⟨*proof*⟩
lemma *cball-empty*: *e* < *0* ==> *cball x e* = {} ⟨*proof*⟩

lemma *cball-eq-sing*: (*cball x e* = {*x*}) ⟷ *e* = *0*
⟨*proof*⟩

lemma *cball-sing*: *e* = *0* ==> *cball x e* = {*x*} ⟨*proof*⟩

For points in the interior, localization of limits makes no difference.

lemma *eventually-within-interior*: **assumes** *x* ∈ *interior S*
shows *eventually P* (*at x within S*) ⟷ *eventually P* (*at x*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *lim-within-interior*: *x* ∈ *interior S* ==> ((*f* ---> *l*) (*at x within S*) ⟷ (*f* ---> *l*) (*at x*))
⟨*proof*⟩

lemma *netlimit-within-interior*: **assumes** *x* ∈ *interior S*
shows *netlimit*(*at x within S*) = *x* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

64.20 Boundedness.

definition *bounded S* $\longleftrightarrow$ $(\exists\, a.\ \forall (x{::}real\hat{}'n{::}finite) \in S.\ norm\ x <= a)$

lemma *bounded-empty*[*simp*]: *bounded* {} ⟨*proof*⟩
lemma *bounded-subset*: *bounded T* $\Longrightarrow$ $S \subseteq T$ ==> *bounded S*
⟨*proof*⟩

lemma *bounded-interior*[*intro*]: *bounded S* ==> *bounded*(*interior S*)
⟨*proof*⟩

lemma *bounded-closure*[*intro*]: **assumes** *bounded S* **shows** *bounded*(*closure S*)
⟨*proof*⟩

lemma *bounded-cball*[*simp*,*intro*]: *bounded* (*cball x e*)
⟨*proof*⟩

lemma *bounded-ball*[*simp*,*intro*]: *bounded*(*ball x e*)
⟨*proof*⟩

lemma *finite-imp-bounded*[*intro*]: **assumes** *finite S* **shows** *bounded S*
⟨*proof*⟩

lemma *bounded-Un*[*simp*]: *bounded* $(S \cup T)$ $\longleftrightarrow$ *bounded S* $\wedge$ *bounded T*
⟨*proof*⟩

lemma *bounded-Union*[*intro*]: *finite F* $\Longrightarrow$ $(\forall\, S{\in}F.\ bounded\ S) \Longrightarrow bounded(\bigcup F)$
⟨*proof*⟩

lemma *bounded-pos*: *bounded S* $\longleftrightarrow$ $(\exists\, b{>}0.\ \forall\, x{\in}\ S.\ norm\ x <= b)$
⟨*proof*⟩

lemma *bounded-Int*[*intro*]: *bounded S* $\vee$ *bounded T* $\Longrightarrow$ *bounded* $(S \cap T)$
⟨*proof*⟩

lemma *bounded-diff*[*intro*]: *bounded S* ==> *bounded* $(S - T)$
⟨*proof*⟩

lemma *bounded-insert*[*intro*]:*bounded*(*insert x S*) $\longleftrightarrow$ *bounded S*
⟨*proof*⟩

lemma *bot-bounded-UNIV*[*simp*, *intro*]: ~(*bounded* (*UNIV*:: (*real*^'*n*::*finite*) *set*))
⟨*proof*⟩

lemma *bounded-linear-image*:
fixes *f* :: *real*^'*m*::*finite* ⇒ *real*^'*n*::*finite*
assumes *bounded S linear f*
shows *bounded*(*f* ‘ *S*)
⟨*proof*⟩

lemma *bounded-scaling*: *bounded S* $\Longrightarrow$ *bounded* ((λ*x*. *c* ∗*s x*) ‘ *S*)
⟨*proof*⟩

lemma *bounded-translation*: **assumes** *bounded S* **shows** *bounded* ((λ*x*. *a* + *x*) ‘ *S*)
⟨*proof*⟩

Some theorems on sups and infs using the notion "bounded".

lemma *bounded-vec1*: *bounded*(*vec1* ‘ *S*) ⟷ (∃ *a*. ∀ *x*∈*S*. *abs x* <= *a*)
⟨*proof*⟩

lemma *bounded-has-rsup*: **assumes** *bounded*(*vec1* ‘ *S*) *S* ≠ {}
shows ∀ *x*∈*S*. *x* <= *rsup S* **and** ∀ *b*. (∀ *x*∈*S*. *x* <= *b*) ⟶ *rsup S* <= *b*
⟨*proof*⟩

lemma *rsup-insert*: **assumes** *bounded* (*vec1* ‘ *S*)
shows *rsup*(*insert x S*) = (*if S* = {} *then x else max x* (*rsup S*))
⟨*proof*⟩

lemma *sup-insert-finite*: *finite S* $\Longrightarrow$ *rsup*(*insert x S*) = (*if S* = {} *then x else max x* (*rsup S*))
⟨*proof*⟩

lemma *bounded-has-rinf*:
assumes *bounded*(*vec1* ‘ *S*) *S* ≠ {}
shows ∀ *x*∈*S*. *x* >= *rinf S* **and** ∀ *b*. (∀ *x*∈*S*. *x* >= *b*) ⟶ *rinf S* >= *b*
⟨*proof*⟩

lemma *real-isGlb-unique*: [| *isGlb R S x*; *isGlb R S y* |] ==> *x* = (*y*::*real*)
⟨*proof*⟩

lemma *rinf-insert*: **assumes** *bounded* (*vec1* ‘ *S*)
shows *rinf*(*insert x S*) = (*if S* = {} *then x else min x* (*rinf S*)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *inf-insert-finite*: *finite S* ==> *rinf*(*insert x S*) = (*if S* = {} *then x else min x* (*rinf S*))
⟨*proof*⟩

64.21 Compactness (the definition is the one based on convegent subsequences).

definition *compact S* ⟷
(∀ (*f*::*nat* ⇒ *real*^'*n*::*finite*). (∀ *n*. *f n* ∈ *S*) ⟶
(∃ *l*∈*S*. ∃ *r*. (∀ *m n*. *m* < *n* ⟶ *r m* < *r n*) ∧ ((*f o r*) −−−> *l*) *sequentially*))

lemma *monotone-bigger*: **fixes** *r*::*nat*⇒*nat*
assumes ∀ *m n*::*nat*. *m* < *n* −−> *r m* < *r n*

shows $n \leq r\ n$
⟨*proof*⟩

lemma *lim-subsequence*: $\forall\, m\ n.\ m < n \longrightarrow r\ m < r\ n \Longrightarrow (s \longrightarrow l)$ *sequentially* $\Longrightarrow ((s\ o\ r) \longrightarrow l)$ *sequentially*
⟨*proof*⟩

lemma *num-Axiom*: $EX!\ g.\ g\ 0 = e \wedge (\forall\, n.\ g\ (Suc\ n) = f\ n\ (g\ n))$
⟨*proof*⟩

lemma *convergent-bounded-increasing*: **fixes** s ::*nat⇒real*
assumes $\forall\, m\ n.\ m \leq n \longrightarrow s\ m \leq s\ n$ **and** $\forall\, n.\ abs(s\ n) \leq b$
shows $\exists\ l.\ \forall\, e$::*real*$>0.\ \exists\ N.\ \forall\, n \geq N.\ \ abs(s\ n - l) < e$
⟨*proof*⟩

lemma *convergent-bounded-monotone*: **fixes** s::*nat* ⇒ *real*
assumes $\forall\, n.\ abs(s\ n) \leq b$ **and** $(\forall\, m\ n.\ m \leq n \longrightarrow s\ m \leq s\ n) \vee (\forall\, m\ n.\ m \leq n \longrightarrow s\ n \leq s\ m)$
shows $\exists\, l.\ \forall\, e$::*real*$>0.\ \exists\, N.\ \forall\, n \geq N.\ abs(s\ n - l) < e$
⟨*proof*⟩

lemma *compact-real-lemma*:
assumes $\forall\, n$::*nat*. $abs(s\ n) \leq b$
shows $\exists\, l\ r.\ (\forall\, m\ n$::*nat*. $m < n \longrightarrow r\ m < r\ n) \wedge$
$(\forall\, e$>*0*::*real*. $\exists\, N.\ \forall\, n \geq N.\ (abs(s\ (r\ n) - l) < e))$
⟨*proof*⟩

lemma *compact-lemma*:
assumes *bounded s* **and** $\forall\, n.\ (x$::*nat* ⇒*real^'a*::*finite*$)\ n \in s$
shows $\forall\, d.$
$\exists\, l$::$(real\hat{}'a$::*finite*$).\ \exists\ r.\ (\forall\, n\ m$::*nat*. $m < n \longrightarrow r\ m < r\ n) \wedge$
$(\forall\, e>0.\ \exists\, N.\ \forall\, n \geq N.\ \forall\, i \in d.\ |x\ (r\ n)\ \$\ i - l\ \$\ i| < e)$
⟨*proof*⟩

lemma *bounded-closed-imp-compact*: **fixes** s::$(real\hat{}'a$::*finite*$)$ *set*
assumes *bounded s* **and** *closed s*
shows *compact s*
⟨*proof*⟩

64.22 Completeness.

definition *cauchy-def*:*cauchy* $s \longleftrightarrow (\forall\, e>0.\ \exists\, N.\ \forall\, m\ n.\ m \geq N \wedge n \geq N \longrightarrow dist(s\ m)(s\ n) < e)$

definition *complete-def*:*complete* $s \longleftrightarrow (\forall\, f$::$(nat$=>*real^'a*::*finite*$).\ (\forall\, n.\ f\ n \in s) \wedge$ *cauchy f*
$\longrightarrow (\exists\, l \in s.\ (f \longrightarrow l)$ *sequentially*$))$

lemma *cauchy*: *cauchy s* $\longleftrightarrow$ $(\forall\, e{>}0.\exists\ N{::}nat.\ \forall\, n{\geq}N.\ dist(s\ n)(s\ N) < e)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *convergent-imp-cauchy*:
$(s \longrightarrow l)$ *sequentially* ==> *cauchy s*
⟨*proof*⟩

lemma *cauchy-imp-bounded*: **assumes** *cauchy s* **shows** *bounded* $\{y.\ (\exists\, n{::}nat.\ y = s\ n)\}$
⟨*proof*⟩

lemma *compact-imp-complete*: **assumes** *compact s* **shows** *complete s*
⟨*proof*⟩

lemma *complete-univ*:
complete UNIV
⟨*proof*⟩

lemma *complete-eq-closed*: *complete s* $\longleftrightarrow$ *closed s* (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *convergent-eq-cauchy*: $(\exists\, l.\ (s \longrightarrow l)$ *sequentially*$)$ $\longleftrightarrow$ *cauchy s* (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *convergent-imp-bounded*: $(s \longrightarrow l)$ *sequentially* ==> *bounded* $(s\ `\ (UNIV{::}(nat\ set)))$
⟨*proof*⟩

64.23 Total boundedness.

fun *helper-1*::$((real\hat{}'n{::}finite)\ set) \Rightarrow real \Rightarrow nat \Rightarrow real\hat{}'n$ **where**
helper-1 s e n = $(SOME\ y{::}real\hat{}'n.\ y \in s \wedge (\forall\, m{<}n.\ \neg\ (dist\ (helper\text{-}1\ s\ e\ m)\ y < e)))$
declare *helper-1.simps*[*simp del*]

lemma *compact-imp-totally-bounded*:
assumes *compact s*
shows $\forall\, e{>}0.\ \exists\, k.\ finite\ k \wedge k \subseteq s \wedge s \subseteq (\bigcup((\lambda x.\ ball\ x\ e)\ `\ k))$
⟨*proof*⟩

64.24 Heine-Borel theorem (following Burkill & Burkill vol. 2)

lemma *heine-borel-lemma*: **fixes** *s*::$(real\hat{}'n{::}finite)\ set$
assumes *compact s* $s \subseteq (\bigcup\ t)$ $\forall\, b \in t.\ open\ b$
shows $\exists\, e{>}0.\ \forall\, x \in s.\ \exists\, b \in t.\ ball\ x\ e \subseteq b$
⟨*proof*⟩

lemma *compact-imp-heine-borel*: *compact s ==> (∀f. (∀t ∈ f. open t) ∧ s ⊆ (⋃ f)*
⟶ (∃f'. f' ⊆ f ∧ finite f' ∧ s ⊆ (⋃ f')))
⟨*proof*⟩

64.25 Bolzano-Weierstrass property.

lemma *heine-borel-imp-bolzano-weierstrass*:
assumes *∀f. (∀t ∈ f. open t) ∧ s ⊆ (⋃ f) −−> (∃f'. f' ⊆ f ∧ finite f' ∧ s ⊆ (⋃ f'))*
infinite t t ⊆ s
shows *∃x ∈ s. x islimpt t*
⟨*proof*⟩

64.26 Complete the chain of compactness variants.

primrec *helper-2*::(*real ⇒ real^'n*::*finite*) *⇒ nat ⇒ real ^'n* **where**
helper-2 beyond 0 = beyond 0 |
helper-2 beyond (Suc n) = beyond (norm (helper-2 beyond n) + 1)

lemma *bolzano-weierstrass-imp-bounded*: **fixes** *s*::(*real^'n*::*finite*) *set*
assumes *∀t. infinite t ∧ t ⊆ s −−> (∃x ∈ s. x islimpt t)*
shows *bounded s*
⟨*proof*⟩

lemma *sequence-infinite-lemma*:
assumes *∀n*::*nat*. (*f n ≠ l*) (*f −−−> l*) *sequentially*
shows *infinite {y*::*real^'a*::*finite*. (*∃ n*. *y = f n*)*}*
⟨*proof*⟩

lemma *sequence-unique-limpt*:
assumes *∀n*::*nat*. (*f n ≠ l*) (*f −−−> l*) *sequentially l' islimpt {y*. (*∃n*. *y = f n*)*}*
shows *l' = l*
⟨*proof*⟩

lemma *bolzano-weierstrass-imp-closed*:
assumes *∀t. infinite t ∧ t ⊆ s −−> (∃x ∈ s. x islimpt t)*
shows *closed s*
⟨*proof*⟩

Hence express everything as an equivalence.

lemma *compact-eq-heine-borel*: *compact s ⟷*
(∀f. (∀t ∈ f. open t) ∧ s ⊆ (⋃ f)
*−−> (∃f'. f' ⊆ f ∧ finite f' ∧ s ⊆ (⋃ f'))) (***is** *?lhs = ?rhs)*
⟨*proof*⟩

lemma *compact-eq-bolzano-weierstrass*:

compact s $\longleftrightarrow$ $(\forall\, t.\ \textit{infinite}\ t \wedge t \subseteq s \longrightarrow (\exists\, x \in s.\ x\ \textit{islimpt}\ t))$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *compact-eq-bounded-closed*:
compact s $\longleftrightarrow$ *bounded s* $\wedge$ *closed s* (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *compact-imp-bounded*:
compact s ==> *bounded s*
⟨*proof*⟩

lemma *compact-imp-closed*:
compact s ==> *closed s*
⟨*proof*⟩

In particular, some common special cases.

lemma *compact-empty*[*simp*]:
compact {}
⟨*proof*⟩

lemma *compact-union*[*intro*]:
compact s $\Longrightarrow$ *compact t* ==> *compact* $(s \cup t)$
⟨*proof*⟩

lemma *compact-inter*[*intro*]:
compact s $\Longrightarrow$ *compact t* ==> *compact* $(s \cap t)$
⟨*proof*⟩

lemma *compact-inter-closed*[*intro*]:
compact s $\Longrightarrow$ *closed t* ==> *compact* $(s \cap t)$
⟨*proof*⟩

lemma *closed-inter-compact*[*intro*]:
closed s $\Longrightarrow$ *compact t* ==> *compact* $(s \cap t)$
⟨*proof*⟩

lemma *finite-imp-closed*:
finite s ==> *closed s*
⟨*proof*⟩

lemma *finite-imp-compact*:
finite s ==> *compact s*
⟨*proof*⟩

lemma *compact-sing*[*simp*]:
compact {*a*}
⟨*proof*⟩

lemma *closed-sing*[*simp*]:
closed {*a*}
⟨*proof*⟩

lemma *compact-cball*[*simp*]:
compact(*cball x e*)
⟨*proof*⟩

lemma *compact-frontier-bounded*[*intro*]:
bounded s ==> *compact*(*frontier s*)
⟨*proof*⟩

lemma *compact-frontier*[*intro*]:
compact s ==> *compact* (*frontier s*)
⟨*proof*⟩

lemma *frontier-subset-compact*:
compact s ==> *frontier s* ⊆ *s*
⟨*proof*⟩

lemma *open-delete*:
open s ==> *open*(*s* − {*x*})
⟨*proof*⟩

Finite intersection property. I could make it an equivalence in fact.

lemma *compact-imp-fip*:
assumes *compact s* ∀ *t* ∈ *f*. *closed t*
∀ *f'*. *finite f'* ∧ *f'* ⊆ *f* −−> (*s* ∩ (⋂ *f'*) ≠ {})
shows *s* ∩ (⋂ *f*) ≠ {}
⟨*proof*⟩

64.27 Bounded closed nest property (proof does not use Heine-Borel).

lemma *bounded-closed-nest*:
assumes ∀ *n*. *closed*(*s n*) ∀ *n*. (*s n* ≠ {})
(∀ *m n*. *m* ≤ *n* −−> *s n* ⊆ *s m*) *bounded*(*s* 0)
shows ∃ *a*::*real*^'*a*::*finite*. ∀ *n*::*nat*. *a* ∈ *s*(*n*)
⟨*proof*⟩

Decreasing case does not even need compactness, just completeness.

lemma *decreasing-closed-nest*:
assumes ∀ *n*. *closed*(*s n*)
∀ *n*. (*s n* ≠ {})
∀ *m n*. *m* ≤ *n* −−> *s n* ⊆ *s m*
∀ *e*>0. ∃ *n*. ∀ *x* ∈ (*s n*). ∀ *y* ∈ (*s n*). *dist x y* < *e*
shows ∃ *a*::*real*^'*a*::*finite*. ∀ *n*::*nat*. *a* ∈ *s n*
⟨*proof*⟩

Strengthen it to the intersection actually being a singleton.

lemma *decreasing-closed-nest-sing*:
assumes $\forall n.\ closed(s\ n)$
$\forall n.\ s\ n \neq \{\}$
$\forall m\ n.\ m \leq n \longrightarrow s\ n \subseteq s\ m$
$\forall e{>}0.\ \exists n.\ \forall x \in (s\ n).\ \forall\ y{\in}(s\ n).\ dist\ x\ y < e$
shows $\exists a{::}real\hat{}'a{::}finite.\ \bigcap\ \{t.\ (\exists n{::}nat.\ t = s\ n)\} = \{a\}$
⟨*proof*⟩

Cauchy-type criteria for uniform convergence.

lemma *uniformly-convergent-eq-cauchy*: **fixes** $s{::}nat \Rightarrow 'b \Rightarrow real\hat{}'a{::}finite$ **shows**
$(\exists l.\ \forall e{>}0.\ \exists N.\ \forall n\ x.\ N \leq n \wedge P\ x \longrightarrow dist(s\ n\ x)(l\ x) < e) \longleftrightarrow$
$(\forall e{>}0.\ \exists N.\ \forall m\ n\ x.\ N \leq m \wedge N \leq n \wedge P\ x \longrightarrow dist\ (s\ m\ x)\ (s\ n\ x) < e)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *uniformly-cauchy-imp-uniformly-convergent*:
assumes $\forall e{>}0.\exists N.\ \forall m\ (n{::}nat)\ x.\ N \leq m \wedge N \leq n \wedge P\ x \longrightarrow dist(s\ m\ x)(s\ n\ x) < e$
$\forall x.\ P\ x \longrightarrow (\forall e{>}0.\ \exists N.\ \forall n.\ N \leq n \longrightarrow dist(s\ n\ x)(l\ x) < e)$
shows $\forall e{>}0.\ \exists N.\ \forall n\ x.\ N \leq n \wedge P\ x \longrightarrow dist(s\ n\ x)(l\ x) < e$
⟨*proof*⟩

64.28 Define continuity over a net to take in restrictions of the set.

definition *continuous net f* $\longleftrightarrow$ $(f \longrightarrow f(netlimit\ net))\ net$

lemma *continuous-trivial-limit*:
trivial-limit net $\Longrightarrow$ *continuous net f*
⟨*proof*⟩

lemma *continuous-within*: *continuous* $(at\ x\ within\ s)\ f \longleftrightarrow (f \longrightarrow f(x))\ (at\ x\ within\ s)$
⟨*proof*⟩

lemma *continuous-at*: *continuous* $(at\ x)\ f \longleftrightarrow (f \longrightarrow f(x))\ (at\ x)$ ⟨*proof*⟩

lemma *continuous-at-within*:
assumes *continuous* $(at\ x)\ f$ **shows** *continuous* $(at\ x\ within\ s)\ f$
⟨*proof*⟩

Derive the epsilon-delta forms, which we often use as "definitions"

lemma *continuous-within-eps-delta*:
continuous $(at\ x\ within\ s)\ f \longleftrightarrow (\forall e{>}0.\ \exists d{>}0.\ \forall x'{\in}\ s.\ \ dist\ x'\ x < d \longrightarrow dist\ (f\ x')\ (f\ x) < e)$
⟨*proof*⟩

lemma *continuous-at-eps-delta*: *continuous* $(at\ x)\ f \longleftrightarrow\ (\forall e{>}0.\ \exists d{>}0.$

$$\forall x'.\ dist\ x'\ x < d \longrightarrow dist(f\ x')(f\ x) < e)$$

⟨*proof*⟩

Versions in terms of open balls.

lemma *continuous-within-ball*:
continuous (*at x within s*) *f* ⟷ (∀ *e*>*0*. ∃ *d*>*0*.
f ‘ (*ball x d* ∩ *s*) ⊆ *ball* (*f x*) *e*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *continuous-at-ball*: **fixes** *f*::*real^'a*::*finite* ⇒ *real^'a*
shows *continuous* (*at x*) *f* ⟷ (∀ *e*>*0*. ∃ *d*>*0*. *f* ‘ (*ball x d*) ⊆ *ball* (*f x*) *e*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

For setwise continuity, just start from the epsilon-delta definitions.

definition *continuous-on s f* ⟷ (∀ *x* ∈ *s*. ∀ *e*>*0*. ∃ *d*::*real*>*0*. ∀ *x'* ∈ *s*. *dist x' x* < *d* −−> *dist* (*f x'*) (*f x*) < *e*)

definition *uniformly-continuous-on s f* ⟷
(∀ *e*>*0*. ∃ *d*>*0*. ∀ *x*∈*s*. ∀ *x'*∈*s*. *dist x' x* < *d*
−−> *dist* (*f x'*) (*f x*) < *e*)

Some simple consequential lemmas.

lemma *uniformly-continuous-imp-continuous*:
uniformly-continuous-on s f ==> *continuous-on s f*
⟨*proof*⟩

lemma *continuous-at-imp-continuous-within*:
continuous (*at x*) *f* ==> *continuous* (*at x within s*) *f*
⟨*proof*⟩

lemma *continuous-at-imp-continuous-on*: **assumes** (∀ *x* ∈ *s*. *continuous* (*at x*) *f*)
shows *continuous-on s f*
⟨*proof*⟩

lemma *continuous-on-eq-continuous-within*:
continuous-on s f ⟷ (∀ *x* ∈ *s*. *continuous* (*at x within s*) *f*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *continuous-on*:
continuous-on s f ⟷ (∀ *x* ∈ *s*. (*f* −−−> *f*(*x*)) (*at x within s*))
⟨*proof*⟩

lemma *continuous-on-eq-continuous-at*:
open s ==> (*continuous-on s f* ⟷ (∀ *x* ∈ *s*. *continuous* (*at x*) *f*))
⟨*proof*⟩

lemma *continuous-within-subset*:

continuous (at x within s) f $\Longrightarrow$ $t \subseteq s$
$==>$ *continuous (at x within t) f*
⟨*proof*⟩

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

lemma *continuous-on-interior*:
continuous-on s f $\Longrightarrow$ $x \in$ *interior s* $==>$ *continuous (at x) f*
⟨*proof*⟩

lemma *continuous-on-eq*:
$(\forall x \in s.\ f\ x = g\ x)$ $\Longrightarrow$ *continuous-on s f*
$==>$ *continuous-on s g*
⟨*proof*⟩

Characterization of various kinds of continuity in terms of sequences.

lemma *continuous-within-sequentially*:
continuous (at a within s) f $\longleftrightarrow$
$(\forall x.\ (\forall n{::}nat.\ x\ n \in s) \wedge (x \longrightarrow a)\ sequentially$
$\longrightarrow ((f\ o\ x) \longrightarrow f\ a)\ sequentially)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *continuous-at-sequentially*:
continuous (at a) f $\longleftrightarrow$ $(\forall x.\ (x \longrightarrow a)\ sequentially$
$\longrightarrow ((f\ o\ x) \longrightarrow f\ a)\ sequentially)$
⟨*proof*⟩

lemma *continuous-on-sequentially*:
continuous-on s f $\longleftrightarrow$ $(\forall x.\ \forall a \in s.\ (\forall n.\ x(n) \in s) \wedge (x \longrightarrow a)\ sequentially$
$\longrightarrow ((f\ o\ x) \longrightarrow f(a))\ sequentially)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *uniformly-continuous-on-sequentially*:
uniformly-continuous-on s f $\longleftrightarrow$ $(\forall x\ y.\ (\forall n.\ x\ n \in s) \wedge (\forall n.\ y\ n \in s) \wedge$
$((\lambda n.\ x\ n - y\ n) \longrightarrow 0)\ sequentially$
$\longrightarrow ((\lambda n.\ f(x\ n) - f(y\ n)) \longrightarrow 0)\ sequentially)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

The usual transformation theorems.

lemma *continuous-transform-within*:
assumes $0 < d\ x \in s\ \forall x' \in s.\ dist\ x'\ x < d \longrightarrow f\ x' = g\ x'$
continuous (at x within s) f
shows *continuous (at x within s) g*
⟨*proof*⟩

lemma *continuous-transform-at*:

assumes $0 < d$ $\forall x'.$ *dist* x' $x < d$ $--> f\,x' = g\,x'$
continuous (*at* x) f
shows *continuous* (*at* x) g
⟨*proof*⟩

Combination results for pointwise continuity.

lemma *continuous-const*: *continuous net* ($\lambda x::'a::$*zero-neq-one*. c)
⟨*proof*⟩

lemma *continuous-cmul*:
continuous net f ==> *continuous net* ($\lambda x.$ c *∗s* $f\,x$)
⟨*proof*⟩

lemma *continuous-neg*:
continuous net f ==> *continuous net* ($\lambda x.$ $-(f\,x)$)
⟨*proof*⟩

lemma *continuous-add*:
continuous net f $\Longrightarrow$ *continuous net* g
==> *continuous net* ($\lambda x.$ $f\,x + g\,x$)
⟨*proof*⟩

lemma *continuous-sub*:
continuous net f $\Longrightarrow$ *continuous net* g
==> *continuous net* ($\lambda x.$ $f(x) - g(x)$)
⟨*proof*⟩

Same thing for setwise continuity.

lemma *continuous-on-const*:
continuous-on s ($\lambda x.$ c)
⟨*proof*⟩

lemma *continuous-on-cmul*:
continuous-on s f ==> *continuous-on* s ($\lambda x.$ c *∗s* ($f\,x$))
⟨*proof*⟩

lemma *continuous-on-neg*:
continuous-on s f ==> *continuous-on* s ($\lambda x.$ $-(f\,x)$)
⟨*proof*⟩

lemma *continuous-on-add*:
continuous-on s f $\Longrightarrow$ *continuous-on* s g
==> *continuous-on* s ($\lambda x.$ $f\,x + g\,x$)
⟨*proof*⟩

lemma *continuous-on-sub*:
continuous-on s f $\Longrightarrow$ *continuous-on* s g
==> *continuous-on* s ($\lambda x.$ $f(x) - g(x)$)
⟨*proof*⟩

Same thing for uniform continuity, using sequential formulations.

lemma *uniformly-continuous-on-const*:
uniformly-continuous-on s (λ*x*. *c*)
⟨*proof*⟩

lemma *uniformly-continuous-on-cmul*:
assumes *uniformly-continuous-on s f*
shows *uniformly-continuous-on s* (λ*x*. *c* ∗*s f*(*x*))
⟨*proof*⟩

lemma *uniformly-continuous-on-neg*:
uniformly-continuous-on s f
==> *uniformly-continuous-on s* (λ*x*. −(*f x*))
⟨*proof*⟩

lemma *uniformly-continuous-on-add*:
assumes *uniformly-continuous-on s f uniformly-continuous-on s g*
shows *uniformly-continuous-on s* (λ*x*. *f*(*x*) + *g*(*x*) ::*real*^'*n*::*finite*)
⟨*proof*⟩

lemma *uniformly-continuous-on-sub*:
uniformly-continuous-on s f ⟹ *uniformly-continuous-on s g*
==> *uniformly-continuous-on s* (λ*x*. *f x* − *g x*)
⟨*proof*⟩

Identity function is continuous in every sense.

lemma *continuous-within-id*:
continuous (*at a within s*) (λ*x*. *x*)
⟨*proof*⟩

lemma *continuous-at-id*:
continuous (*at a*) (λ*x*. *x*)
⟨*proof*⟩

lemma *continuous-on-id*:
continuous-on s (λ*x*. *x*)
⟨*proof*⟩

lemma *uniformly-continuous-on-id*:
uniformly-continuous-on s (λ*x*. *x*)
⟨*proof*⟩

Continuity of all kinds is preserved under composition.

lemma *continuous-within-compose*:
assumes *continuous* (*at x within s*) *f* *continuous* (*at* (*f x*) *within f* ‘ *s*) *g*
shows *continuous* (*at x within s*) (*g o f*)
⟨*proof*⟩

lemma *continuous-at-compose*:

assumes *continuous* (*at x*) *f* *continuous* (*at* (*f x*)) *g*
shows *continuous* (*at x*) (*g o f*)
⟨*proof*⟩

lemma *continuous-on-compose*:
continuous-on s f ⟹ *continuous-on* (*f ' s*) *g* ⟹ *continuous-on s* (*g o f*)
⟨*proof*⟩

lemma *uniformly-continuous-on-compose*:
assumes *uniformly-continuous-on s f* *uniformly-continuous-on* (*f ' s*) *g*
shows *uniformly-continuous-on s* (*g o f*)
⟨*proof*⟩

Continuity in terms of open preimages.

lemma *continuous-at-open*:
continuous (*at x*) *f* ⟷ (∀ *t*. *open t* ∧ *f x* ∈ *t* −−> (∃ *s*. *open s* ∧ *x* ∈ *s* ∧ (∀ *x'* ∈ *s*. (*f x'*) ∈ *t*))) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *continuous-on-open*:
continuous-on s f ⟷
(∀ *t*. *openin* (*subtopology euclidean* (*f ' s*)) *t*
−−> *openin* (*subtopology euclidean s*) {*x* ∈ *s*. *f x* ∈ *t*}) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *continuous-on-closed*:
continuous-on s f ⟷ (∀ *t*. *closedin* (*subtopology euclidean* (*f ' s*)) *t* −−> *closedin* (*subtopology euclidean s*) {*x* ∈ *s*. *f x* ∈ *t*}) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

Half-global and completely global cases.

lemma *continuous-open-in-preimage*:
assumes *continuous-on s f* *open t*
shows *openin* (*subtopology euclidean s*) {*x* ∈ *s*. *f x* ∈ *t*}
⟨*proof*⟩

lemma *continuous-closed-in-preimage*:
assumes *continuous-on s f* *closed t*
shows *closedin* (*subtopology euclidean s*) {*x* ∈ *s*. *f x* ∈ *t*}
⟨*proof*⟩

lemma *continuous-open-preimage*:
assumes *continuous-on s f open s open t*
shows *open* {*x* ∈ *s*. *f x* ∈ *t*}
⟨*proof*⟩

lemma *continuous-closed-preimage*:
assumes *continuous-on s f closed s closed t*
shows *closed* $\{x \in s.\ f\ x \in t\}$
⟨*proof*⟩

lemma *continuous-open-preimage-univ*:
$\forall x.$ *continuous* (*at x*) *f* $\Longrightarrow$ *open s* $\Longrightarrow$ *open* $\{x.\ f\ x \in s\}$
⟨*proof*⟩

lemma *continuous-closed-preimage-univ*:
$(\forall x.$ *continuous* (*at x*) *f*$) \Longrightarrow$ *closed s* ==> *closed* $\{x.\ f\ x \in s\}$
⟨*proof*⟩

Equality of continuous functions on closure and related results.

lemma *continuous-closed-in-preimage-constant*:
continuous-on s f ==> *closedin* (*subtopology euclidean s*) $\{x \in s.\ f\ x = a\}$
⟨*proof*⟩

lemma *continuous-closed-preimage-constant*:
continuous-on s f $\Longrightarrow$ *closed s* ==> *closed* $\{x \in s.\ f\ x = a\}$
⟨*proof*⟩

lemma *continuous-constant-on-closure*:
assumes *continuous-on* (*closure s*) *f*
$\forall x \in s.\ f\ x = a$
shows $\forall x \in$ (*closure s*). $f\ x = a$
⟨*proof*⟩

lemma *image-closure-subset*:
assumes *continuous-on* (*closure s*) *f closed t* $(f\ `\ s) \subseteq t$
shows $f\ `$ (*closure s*) $\subseteq t$
⟨*proof*⟩

lemma *continuous-on-closure-norm-le*:
assumes *continuous-on* (*closure s*) *f* $\forall y \in s.$ *norm*$(f\ y) \leq b$ $x \in$ (*closure s*)
shows *norm*$(f\ x) \leq b$
⟨*proof*⟩

Making a continuous function avoid some value in a neighbourhood.

lemma *continuous-within-avoid*:
assumes *continuous* (*at x within s*) *f* $x \in s$ $f\ x \neq a$
shows $\exists e{>}0.\ \forall y \in s.$ *dist x y* $< e$ −−> $f\ y \neq a$
⟨*proof*⟩

lemma *continuous-at-avoid*:
assumes *continuous* (*at x*) *f* $f\ x \neq a$
shows $\exists e{>}0.\ \forall y.$ *dist x y* $< e \longrightarrow f\ y \neq a$
⟨*proof*⟩

lemma *continuous-on-avoid*:
assumes *continuous-on s f* $x \in s$ $f\,x \neq a$
shows $\exists e{>}0.\ \forall y \in s.\ dist\ x\ y < e \longrightarrow f\ y \neq a$
⟨*proof*⟩

lemma *continuous-on-open-avoid*:
assumes *continuous-on s f open s* $x \in s$ $f\,x \neq a$
shows $\exists e{>}0.\ \forall y.\ dist\ x\ y < e \longrightarrow f\ y \neq a$
⟨*proof*⟩

Proving a function is constant by proving open-ness of level set.

lemma *continuous-levelset-open-in-cases*:
connected s $\Longrightarrow$ *continuous-on s f* $\Longrightarrow$
openin (*subtopology euclidean s*) $\{x \in s.\ f\,x = a\}$
$==>$ $(\forall x \in s.\ f\,x \neq a) \vee (\forall x \in s.\ f\,x = a)$
⟨*proof*⟩

lemma *continuous-levelset-open-in*:
connected s $\Longrightarrow$ *continuous-on s f* $\Longrightarrow$
openin (*subtopology euclidean s*) $\{x \in s.\ f\,x = a\} \Longrightarrow$
$(\exists x \in s.\ f\,x = a)$ $==>$ $(\forall x \in s.\ f\,x = a)$
⟨*proof*⟩

lemma *continuous-levelset-open*:
assumes *connected s continuous-on s f open* $\{x \in s.\ f\,x = a\}$ $\exists x \in s.\ f\,x = a$
shows $\forall x \in s.\ f\,x = a$
⟨*proof*⟩

Some arithmetical combinations (more to prove).

lemma *open-scaling*[*intro*]:
assumes $c \neq 0$ *open s*
shows *open*$((\lambda x.\ c *s x)\ `\ s)$
⟨*proof*⟩

lemma *open-negations*:
open s $==>$ *open* $((\lambda\ x.\ -x)\ `\ s)$ ⟨*proof*⟩

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

lemma *open-affinity*:
assumes *open s* $c \neq 0$
shows *open* $((\lambda x.\ a + c *s x)\ `\ s)$
⟨*proof*⟩

lemma *interior-translation*: *interior* $((\lambda x.\ a + x)\ `\ s) = (\lambda x.\ a + x)\ `\ (interior\ s)$

⟨*proof*⟩

64.29 Preservation of compactness and connectedness under continuous function.

lemma *compact-continuous-image*:
assumes *continuous-on s f* *compact s*
shows *compact*(*f ' s*)
⟨*proof*⟩

lemma *connected-continuous-image*:
assumes *continuous-on s f* *connected s*
shows *connected*(*f ' s*)
⟨*proof*⟩

Continuity implies uniform continuity on a compact domain.

lemma *compact-uniformly-continuous*:
assumes *continuous-on s f* *compact s*
shows *uniformly-continuous-on s f*
⟨*proof*⟩

Continuity of inverse function on compact domain.

lemma *continuous-on-inverse*:
assumes *continuous-on s f* *compact s* $\forall x \in s.\ g\ (f\ x) = x$
shows *continuous-on* (*f ' s*) *g*
⟨*proof*⟩

64.30 A uniformly convergent limit of continuous functions is continuous.

lemma *continuous-uniform-limit*:
assumes $\neg$ (*trivial-limit net*) *eventually* ($\lambda n.$ *continuous-on s* (*f n*)) *net*
$\forall e{>}0.$ *eventually* ($\lambda n. \forall x \in s.$ *norm*(*f n x − g x*) $<$ *e*) *net*
shows *continuous-on s g*
⟨*proof*⟩

64.31 Topological properties of linear functions.

lemma *linear-lim-0*: **fixes** *f*::*real^'a*::*finite* $\Rightarrow$ *real^'b*::*finite*
assumes *linear f* **shows** (*f* −−−> *0*) (*at* (*0*))
⟨*proof*⟩

lemma *linear-continuous-at*:
assumes *linear f* **shows** *continuous* (*at a*) *f*
⟨*proof*⟩

lemma *linear-continuous-within*:
linear f ==> *continuous* (*at x within s*) *f*
⟨*proof*⟩

lemma *linear-continuous-on*:
linear f ==> continuous-on s f
⟨*proof*⟩

Also bilinear functions, in composition form.

lemma *bilinear-continuous-at-compose*:
continuous (at x) f ⟹ continuous (at x) g ⟹ bilinear h
==> continuous (at x) (λx. h (f x) (g x))
⟨*proof*⟩

lemma *bilinear-continuous-within-compose*:
continuous (at x within s) f ⟹ continuous (at x within s) g ⟹ bilinear h
==> continuous (at x within s) (λx. h (f x) (g x))
⟨*proof*⟩

lemma *bilinear-continuous-on-compose*:
continuous-on s f ⟹ continuous-on s g ⟹ bilinear h
==> continuous-on s (λx. h (f x) (g x))
⟨*proof*⟩

64.32 Topological stuff lifted from and dropped to R

lemma *open-vec1*:
open(vec1 ' s) ⟷
(∀ x ∈ s. ∃ e>0. ∀ x'. abs(x' − x) < e −−> x' ∈ s) (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *islimpt-approachable-vec1*:
(vec1 x) islimpt (vec1 ' s) ⟷
(∀ e>0. ∃ x'∈ s. x' ≠ x ∧ abs(x' − x) < e)
⟨*proof*⟩

lemma *closed-vec1*:
closed (vec1 ' s) ⟷
(∀ x. (∀ e>0. ∃ x' ∈ s. x' ≠ x ∧ abs(x' − x) < e)
−−> x ∈ s)
⟨*proof*⟩

lemma *continuous-at-vec1-range*:
continuous (at x) (vec1 o f) ⟷ (∀ e>0. ∃ d>0.
∀ x'. norm(x' − x) < d −−> abs(f x' − f x) < e)
⟨*proof*⟩

lemma *continuous-on-vec1-range*:
continuous-on s (vec1 o f) ⟷ (∀ x ∈ s. ∀ e>0. ∃ d>0. (∀ x' ∈ s. norm(x' − x) < d −−> abs(f x' − f x) < e))
⟨*proof*⟩

lemma *continuous-at-vec1-norm*:

$\forall x.\ continuous\ (at\ x)\ (vec1\ o\ norm)$
⟨*proof*⟩

lemma *continuous-on-vec1-norm*:
$\forall s.\ continuous\text{-}on\ s\ (vec1\ o\ norm)$
⟨*proof*⟩

lemma *continuous-at-vec1-component*:
shows *continuous* $(at\ (a{::}real\hat{}'a{::}finite))\ (\lambda\ x.\ vec1(x\$i))$
⟨*proof*⟩

lemma *continuous-on-vec1-component*:
shows *continuous-on* $s\ (\lambda\ x{::}real\hat{}'a{::}finite.\ vec1(x\$i))$
⟨*proof*⟩

lemma *continuous-at-vec1-infnorm*:
$continuous\ (at\ x)\ (vec1\ o\ infnorm)$
⟨*proof*⟩

Hence some handy theorems on distance, diameter etc. of/from a set.

lemma *compact-attains-sup*:
assumes $compact\ (vec1\ `\ s)\ \ s \neq \{\}$
shows $\exists x \in s.\ \forall y \in s.\ y \leq x$
⟨*proof*⟩

lemma *compact-attains-inf*:
assumes $compact\ (vec1\ `\ s)\ s \neq \{\}$ **shows** $\exists x \in s.\ \forall y \in s.\ x \leq y$
⟨*proof*⟩

lemma *continuous-attains-sup*:
$compact\ s \Longrightarrow s \neq \{\} \Longrightarrow continuous\text{-}on\ s\ (vec1\ o\ f)$
$\quad ==> (\exists x \in s.\ \forall y \in s.\ \ f\ y \leq f\ x)$
⟨*proof*⟩

lemma *continuous-attains-inf*:
$compact\ s \Longrightarrow s \neq \{\} \Longrightarrow continuous\text{-}on\ s\ (vec1\ o\ f)$
$\quad ==> (\exists x \in s.\ \forall y \in s.\ f\ x \leq f\ y)$
⟨*proof*⟩

lemma *distance-attains-sup*:
assumes $compact\ s\ s \neq \{\}$
shows $\exists x \in s.\ \forall y \in s.\ dist\ a\ y \leq dist\ a\ x$
⟨*proof*⟩

For *minimal* distance, we only need closure, not compactness.

lemma *distance-attains-inf*:
assumes $closed\ s\ \ s \neq \{\}$
shows $\exists x \in s.\ \forall y \in s.\ dist\ a\ x \leq dist\ a\ y$
⟨*proof*⟩

64.33 We can now extend limit compositions to consider the scalar multiplier.

lemma *Lim-mul*:
assumes $((vec1\ o\ c) \longrightarrow vec1\ d)\ net\ \ (f \longrightarrow l)\ net$
shows $((\lambda x.\ c(x) *s\ f\ x) \longrightarrow (d *s\ l))\ net$
⟨*proof*⟩

lemma *Lim-vmul*:
$((vec1\ o\ c) \longrightarrow vec1\ d)\ net \Longrightarrow ((\lambda x.\ c(x) *s\ v) \longrightarrow d *s\ v)\ net$
⟨*proof*⟩

lemma *continuous-vmul*:
$continuous\ net\ (vec1\ o\ c) \Longrightarrow continuous\ net\ (\lambda x.\ c(x) *s\ v)$
⟨*proof*⟩

lemma *continuous-mul*:
$continuous\ net\ (vec1\ o\ c) \Longrightarrow continuous\ net\ f$
$\Longrightarrow continuous\ net\ (\lambda x.\ c(x) *s\ f\ x)$
⟨*proof*⟩

lemma *continuous-on-vmul*:
$continuous\text{-}on\ s\ (vec1\ o\ c) \Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ c(x) *s\ v)$
⟨*proof*⟩

lemma *continuous-on-mul*:
$continuous\text{-}on\ s\ (vec1\ o\ c) \Longrightarrow continuous\text{-}on\ s\ f$
$\Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ c(x) *s\ f\ x)$
⟨*proof*⟩

And so we have continuity of inverse.

lemma *Lim-inv*:
assumes $((vec1\ o\ f) \longrightarrow vec1\ l)\ (net::'a\ net)\ \ l \neq 0$
shows $((vec1\ o\ inverse\ o\ f) \longrightarrow vec1(inverse\ l))\ net$
⟨*proof*⟩

lemma *continuous-inv*:
$continuous\ net\ (vec1\ o\ f) \Longrightarrow f(netlimit\ net) \neq 0$
$\Longrightarrow continuous\ net\ (vec1\ o\ inverse\ o\ f)$
⟨*proof*⟩

lemma *continuous-at-within-inv*:
assumes $continuous\ (at\ a\ within\ s)\ (vec1\ o\ f)\ f\ a \neq 0$
shows $continuous\ (at\ a\ within\ s)\ (vec1\ o\ inverse\ o\ f)$
⟨*proof*⟩

lemma *continuous-at-inv*:
$continuous\ (at\ a)\ (vec1\ o\ f) \Longrightarrow f\ a \neq 0$
$\Longrightarrow continuous\ (at\ a)\ (vec1\ o\ inverse\ o\ f)$
⟨*proof*⟩

64.34 Preservation properties for pasted sets.

lemma *bounded-pastecart*:
assumes *bounded s bounded t*
shows *bounded* { *pastecart x y* | *x y* . $(x \in s \wedge y \in t)$}
⟨*proof*⟩

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

lemma *compact-pastecart*:
compact s $\Longrightarrow$ *compact t* ==> *compact* {*pastecart x y* | *x y* . $x \in s \wedge y \in t$}
⟨*proof*⟩

Hence some useful properties follow quite easily.

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

lemma *compact-negations*:
assumes *compact s* **shows** *compact* $((\lambda x.\ -x)\ `\ s)$
⟨*proof*⟩

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

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

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

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

Hence we get the following.

lemma *compact-sup-maxdistance*:
assumes *compact s* $s \neq \{\}$
shows $\exists x{\in}s.\ \exists y{\in}s.\ \forall u{\in}s.\ \forall v{\in}s.\ norm(u - v) \leq norm(x - y)$
⟨*proof*⟩

We can state this in terms of diameter of a set.

definition *diameter s* = (*if s* = {} *then 0*::*real else rsup* {*norm*$(x - y)$ | *x y*. *x* $\in s \wedge y \in s$})

lemma *diameter-bounded*:
assumes *bounded s*
shows $\forall x{\in}s.\ \forall y{\in}s.\ norm(x - y) \leq diameter\ s$
$\forall d{>}0.\ d < diameter\ s \longrightarrow (\exists x{\in}s.\ \exists y{\in}s.\ norm(x - y) > d)$
⟨*proof*⟩

lemma *diameter-bounded-bound*:
$bounded\ s \Longrightarrow x \in s \Longrightarrow y \in s \Longrightarrow norm(x - y) \leq diameter\ s$
⟨*proof*⟩

lemma *diameter-compact-attained*:
assumes *compact s* $s \neq \{\}$
shows $\exists x{\in}s.\ \exists y{\in}s.\ (norm(x - y) = diameter\ s)$
⟨*proof*⟩

Related results with closure as the conclusion.

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

lemma *closed-negations*:
assumes *closed s* **shows** *closed* $((\lambda x.\ -x)\ `\ s)$
⟨*proof*⟩

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

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

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

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

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

lemma *translation-UNIV*:

range (λx::*real*^'*a*. *a* + *x*) = *UNIV*
⟨*proof*⟩

lemma *translation-diff*: (λx::*real*^'*a*. *a* + *x*) ‘ (*s* − *t*) = ((λ. *a* + *x*) ‘ *s*) − ((λ. *a* + *x*) ‘ *t*) ⟨*proof*⟩

lemma *closure-translation*:
closure ((λx. *a* + *x*) ‘ *s*) = (λx. *a* + *x*) ‘ (*closure s*)
⟨*proof*⟩

lemma *frontier-translation*:
frontier((λx. *a* + *x*) ‘ *s*) = (λx. *a* + *x*) ‘ (*frontier s*)
⟨*proof*⟩

64.35 Separation between points and sets.

lemma *separate-point-closed*:
closed s ⟹ *a* ∉ *s* ==> (∃ *d*>*0*. ∀ *x*∈*s*. *d* ≤ *dist a x*)
⟨*proof*⟩

lemma *separate-compact-closed*:
assumes *compact s* **and** *closed t* **and** *s* ∩ *t* = {}
shows ∃ *d*>*0*. ∀ *x*∈*s*. ∀ *y*∈*t*. *d* ≤ *dist x y*
⟨*proof*⟩

lemma *separate-closed-compact*:
assumes *closed s* **and** *compact t* **and** *s* ∩ *t* = {}
shows ∃ *d*>*0*. ∀ *x*∈*s*. ∀ *y*∈*t*. *d* ≤ *dist x y*
⟨*proof*⟩

lemma *interval*: **fixes** *a* :: '*a*::*ord*^'*n*::*finite* **shows**
{*a* <..< *b*} = {*x*::'*a*^'*n*. ∀ *i*. *a*$*i* < *x*$*i* ∧ *x*$*i* < *b*$*i*} **and**
{*a* .. *b*} = {*x*::'*a*^'*n*. ∀ *i*. *a*$*i* ≤ *x*$*i* ∧ *x*$*i* ≤ *b*$*i*}
⟨*proof*⟩

lemma *mem-interval*: **fixes** *a* :: '*a*::*ord*^'*n*::*finite* **shows**
x ∈ {*a*<..<*b*} ⟷ (∀ *i*. *a*$*i* < *x*$*i* ∧ *x*$*i* < *b*$*i*)
x ∈ {*a* .. *b*} ⟷ (∀ *i*. *a*$*i* ≤ *x*$*i* ∧ *x*$*i* ≤ *b*$*i*)
⟨*proof*⟩

lemma *interval-eq-empty*: **fixes** *a* :: *real*^'*n*::*finite* **shows**
({*a* <..< *b*} = {} ⟷ (∃ *i*. *b*$*i* ≤ *a*$*i*)) (**is** *?th1*) **and**
({*a* .. *b*} = {} ⟷ (∃ *i*. *b*$*i* < *a*$*i*)) (**is** *?th2*)
⟨*proof*⟩

lemma *interval-ne-empty*: **fixes** *a* :: *real*^'*n*::*finite* **shows**
{*a* .. *b*} ≠ {} ⟷ (∀ *i*. *a*$*i* ≤ *b*$*i*) **and**

$\{a <..< b\} \neq \{\} \longleftrightarrow (\forall\, i.\ a\$i < b\$i)$
⟨*proof*⟩

lemma *subset-interval-imp*: **fixes** a :: *real^'n::finite* **shows**
$(\forall\, i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i) \Longrightarrow \{c\ ..\ d\} \subseteq \{a\ ..\ b\}$ **and**
$(\forall\, i.\ a\$i < c\$i \wedge d\$i < b\$i) \Longrightarrow \{c\ ..\ d\} \subseteq \{a<..<b\}$ **and**
$(\forall\, i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i) \Longrightarrow \{c<..<d\} \subseteq \{a\ ..\ b\}$ **and**
$(\forall\, i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i) \Longrightarrow \{c<..<d\} \subseteq \{a<..<b\}$
⟨*proof*⟩

lemma *interval-sing*: **fixes** a :: *'a::linorder^'n::finite* **shows**
$\{a\ ..\ a\} = \{a\} \wedge \{a<..<a\} = \{\}$
⟨*proof*⟩

lemma *interval-open-subset-closed*: **fixes** a :: *'a::preorder^'n::finite* **shows**
$\{a<..<b\} \subseteq \{a\ ..\ b\}$
⟨*proof*⟩

lemma *subset-interval*: **fixes** a :: *real^'n::finite* **shows**
$\{c\ ..\ d\} \subseteq \{a\ ..\ b\} \longleftrightarrow (\forall\, i.\ c\$i \leq d\$i) \longrightarrow (\forall\, i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i)$ (**is** *?th1*) **and**
$\{c\ ..\ d\} \subseteq \{a<..<b\} \longleftrightarrow (\forall\, i.\ c\$i \leq d\$i) \longrightarrow (\forall\, i.\ a\$i < c\$i \wedge d\$i < b\$i)$ (**is** *?th2*) **and**
$\{c<..<d\} \subseteq \{a\ ..\ b\} \longleftrightarrow (\forall\, i.\ c\$i < d\$i) \longrightarrow (\forall\, i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i)$ (**is** *?th3*) **and**
$\{c<..<d\} \subseteq \{a<..<b\} \longleftrightarrow (\forall\, i.\ c\$i < d\$i) \longrightarrow (\forall\, i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i)$ (**is** *?th4*)
⟨*proof*⟩

lemma *disjoint-interval*: **fixes** *a::real^'n::finite* **shows**
$\{a\ ..\ b\} \cap \{c\ ..\ d\} = \{\} \longleftrightarrow (\exists\, i.\ (b\$i < a\$i \vee d\$i < c\$i \vee b\$i < c\$i \vee d\$i < a\$i))$ (**is** *?th1*) **and**
$\{a\ ..\ b\} \cap \{c<..<d\} = \{\} \longleftrightarrow (\exists\, i.\ (b\$i < a\$i \vee d\$i \leq c\$i \vee b\$i \leq c\$i \vee d\$i \leq a\$i))$ (**is** *?th2*) **and**
$\{a<..<b\} \cap \{c\ ..\ d\} = \{\} \longleftrightarrow (\exists\, i.\ (b\$i \leq a\$i \vee d\$i < c\$i \vee b\$i \leq c\$i \vee d\$i \leq a\$i))$ (**is** *?th3*) **and**
$\{a<..<b\} \cap \{c<..<d\} = \{\} \longleftrightarrow (\exists\, i.\ (b\$i \leq a\$i \vee d\$i \leq c\$i \vee b\$i \leq c\$i \vee d\$i \leq a\$i))$ (**is** *?th4*)
⟨*proof*⟩

lemma *inter-interval*: **fixes** a :: *'a::linorder^'n::finite* **shows**
$\{a\ ..\ b\} \cap \{c\ ..\ d\} = \{(\chi\ i.\ max\ (a\$i)\ (c\$i))\ ..\ (\chi\ i.\ min\ (b\$i)\ (d\$i))\}$
⟨*proof*⟩

lemma *open-interval-lemma*: **fixes** x :: *real* **shows**
$a < x \Longrightarrow x < b \Longrightarrow (\exists\, d{>}0.\ \forall\, x'.\ abs(x' - x) < d \longrightarrow a < x' \wedge x' < b)$

⟨*proof*⟩

lemma *open-interval*: **fixes** *a* :: *real^'n*::*finite* **shows** *open* {*a*<..<*b*}
⟨*proof*⟩

lemma *closed-interval*: **fixes** *a* :: *real^'n*::*finite* **shows** *closed* {*a* .. *b*}
⟨*proof*⟩

lemma *interior-closed-interval*: **fixes** *a* :: *real^'n*::*finite* **shows**
interior {*a* .. *b*} = {*a*<..<*b*} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *bounded-closed-interval*: **fixes** *a* :: *real^'n*::*finite* **shows**
bounded {*a* .. *b*}
⟨*proof*⟩

lemma *bounded-interval*: **fixes** *a* :: *real^'n*::*finite* **shows**
bounded {*a* .. *b*} ∧ *bounded* {*a*<..<*b*}
⟨*proof*⟩

lemma *not-interval-univ*: **fixes** *a* :: *real^'n*::*finite* **shows**
({*a* .. *b*} ≠ *UNIV*) ∧ ({*a*<..<*b*} ≠ *UNIV*)
⟨*proof*⟩

lemma *compact-interval*: **fixes** *a* :: *real^'n*::*finite* **shows**
compact {*a* .. *b*}
⟨*proof*⟩

lemma *open-interval-midpoint*: **fixes** *a* :: *real^'n*::*finite*
assumes {*a*<..<*b*} ≠ {} **shows** ((*1*/*2*) *s* (*a* + *b*)) ∈ {*a*<..<*b*}
⟨*proof*⟩

lemma *open-closed-interval-convex*: **fixes** *x* :: *real^'n*::*finite*
assumes *x*:*x* ∈ {*a*<..<*b*} **and** *y*:*y* ∈ {*a* .. *b*} **and** *e*:*0* < *e* *e* ≤ *1*
shows (*e* *s* *x* + (*1* − *e*) *s* *y*) ∈ {*a*<..<*b*}
⟨*proof*⟩

lemma *closure-open-interval*: **fixes** *a* :: *real^'n*::*finite*
assumes {*a*<..<*b*} ≠ {}
shows *closure* {*a*<..<*b*} = {*a* .. *b*}
⟨*proof*⟩

lemma *bounded-subset-open-interval-symmetric*: **fixes** *s*::(*real^'n*::*finite*) *set*
assumes *bounded s* **shows** ∃ *a*. *s* ⊆ {−*a*<..<*a*}
⟨*proof*⟩

lemma *bounded-subset-open-interval*:
bounded s ==> (∃ *a b*. *s* ⊆ {*a*<..<*b*})
⟨*proof*⟩

lemma *bounded-subset-closed-interval-symmetric*:
assumes *bounded s* **shows** $\exists\, a.\ s \subseteq \{-a\ ..\ a\}$
⟨*proof*⟩

lemma *bounded-subset-closed-interval*:
bounded s ==> $(\exists\, a\ b.\ s \subseteq \{a\ ..\ b\})$
⟨*proof*⟩

lemma *frontier-closed-interval*:
frontier $\{a\ ..\ b\} = \{a\ ..\ b\} - \{a<..<b\}$
⟨*proof*⟩

lemma *frontier-open-interval*:
frontier $\{a<..<b\} = (\mathit{if}\ \{a<..<b\} = \{\}\ \mathit{then}\ \{\}\ \mathit{else}\ \{a\ ..\ b\} - \{a<..<b\})$
⟨*proof*⟩

lemma *inter-interval-mixed-eq-empty*: **fixes** *a* :: *real^'n::finite*
assumes $\{c<..<d\} \neq \{\}$ **shows** $\{a<..<b\} \cap \{c\ ..\ d\} = \{\} \longleftrightarrow \{a<..<b\} \cap \{c<..<d\} = \{\}$
⟨*proof*⟩

lemma *all-1*: $(\forall\, x::1.\ P\ x) \longleftrightarrow P\ 1$
⟨*proof*⟩

lemma *ex-1*: $(\exists\, x::1.\ P\ x) \longleftrightarrow P\ 1$
⟨*proof*⟩

lemma *interval-cases-1*: **fixes** *x* :: *real^1* **shows**
$x \in \{a\ ..\ b\}$ ==> $x \in \{a<..<b\} \vee (x = a) \vee (x = b)$
⟨*proof*⟩

lemma *in-interval-1*: **fixes** *x* :: *real^1* **shows**
$(x \in \{a\ ..\ b\} \longleftrightarrow \mathit{dest\text{-}vec1}\ a \leq \mathit{dest\text{-}vec1}\ x \wedge \mathit{dest\text{-}vec1}\ x \leq \mathit{dest\text{-}vec1}\ b) \wedge$
$(x \in \{a<..<b\} \longleftrightarrow \mathit{dest\text{-}vec1}\ a < \mathit{dest\text{-}vec1}\ x \wedge \mathit{dest\text{-}vec1}\ x < \mathit{dest\text{-}vec1}\ b)$
⟨*proof*⟩

lemma *interval-eq-empty-1*: **fixes** *a* :: *real^1* **shows**
$\{a\ ..\ b\} = \{\} \longleftrightarrow \mathit{dest\text{-}vec1}\ b < \mathit{dest\text{-}vec1}\ a$
$\{a<..<b\} = \{\} \longleftrightarrow \mathit{dest\text{-}vec1}\ b \leq \mathit{dest\text{-}vec1}\ a$
⟨*proof*⟩

lemma *subset-interval-1*: **fixes** *a* :: *real^1* **shows**
$(\{a\ ..\ b\} \subseteq \{c\ ..\ d\} \longleftrightarrow\ \mathit{dest\text{-}vec1}\ b < \mathit{dest\text{-}vec1}\ a \vee$
$\mathit{dest\text{-}vec1}\ c \leq \mathit{dest\text{-}vec1}\ a \wedge \mathit{dest\text{-}vec1}\ a \leq \mathit{dest\text{-}vec1}\ b \wedge \mathit{dest\text{-}vec1}\ b \leq \mathit{dest\text{-}vec1}\ d)$

$(\{a\ ..\ b\} \subseteq \{c<..<d\} \longleftrightarrow$ *dest-vec1* $b <$ *dest-vec1* $a \vee$
dest-vec1 $c <$ *dest-vec1* $a \wedge$ *dest-vec1* $a \leq$ *dest-vec1* $b \wedge$ *dest-vec1* $b <$ *dest-vec1* $d)$
$(\{a<..<b\} \subseteq \{c\ ..\ d\} \longleftrightarrow$ *dest-vec1* $b \leq$ *dest-vec1* $a \vee$
dest-vec1 $c \leq$ *dest-vec1* $a \wedge$ *dest-vec1* $a <$ *dest-vec1* $b \wedge$ *dest-vec1* $b \leq$ *dest-vec1* $d)$
$(\{a<..<b\} \subseteq \{c<..<d\} \longleftrightarrow$ *dest-vec1* $b \leq$ *dest-vec1* $a \vee$
dest-vec1 $c \leq$ *dest-vec1* $a \wedge$ *dest-vec1* $a <$ *dest-vec1* $b \wedge$ *dest-vec1* $b \leq$ *dest-vec1* $d)$
⟨*proof*⟩

lemma *eq-interval-1*: **fixes** a :: *real^1* **shows**
$\{a\ ..\ b\} = \{c\ ..\ d\} \longleftrightarrow$
dest-vec1 $b <$ *dest-vec1* $a \wedge$ *dest-vec1* $d <$ *dest-vec1* $c \vee$
dest-vec1 $a =$ *dest-vec1* $c \wedge$ *dest-vec1* $b =$ *dest-vec1* d
⟨*proof*⟩

lemma *disjoint-interval-1*: **fixes** a :: *real^1* **shows**
$\{a\ ..\ b\} \cap \{c\ ..\ d\} = \{\} \longleftrightarrow$ *dest-vec1* $b <$ *dest-vec1* $a \vee$ *dest-vec1* $d <$ *dest-vec1* $c \vee$ *dest-vec1* $b <$ *dest-vec1* $c \vee$ *dest-vec1* $d <$ *dest-vec1* a
$\{a\ ..\ b\} \cap \{c<..<d\} = \{\} \longleftrightarrow$ *dest-vec1* $b <$ *dest-vec1* $a \vee$ *dest-vec1* $d \leq$ *dest-vec1* $c \vee$ *dest-vec1* $b \leq$ *dest-vec1* $c \vee$ *dest-vec1* $d \leq$ *dest-vec1* a
$\{a<..<b\} \cap \{c\ ..\ d\} = \{\} \longleftrightarrow$ *dest-vec1* $b \leq$ *dest-vec1* $a \vee$ *dest-vec1* $d <$ *dest-vec1* $c \vee$ *dest-vec1* $b \leq$ *dest-vec1* $c \vee$ *dest-vec1* $d \leq$ *dest-vec1* a
$\{a<..<b\} \cap \{c<..<d\} = \{\} \longleftrightarrow$ *dest-vec1* $b \leq$ *dest-vec1* $a \vee$ *dest-vec1* $d \leq$ *dest-vec1* $c \vee$ *dest-vec1* $b \leq$ *dest-vec1* $c \vee$ *dest-vec1* $d \leq$ *dest-vec1* a
⟨*proof*⟩

lemma *open-closed-interval-1*: **fixes** a :: *real^1* **shows**
$\{a<..<b\} = \{a\ ..\ b\} - \{a,\ b\}$
⟨*proof*⟩

lemma *closed-open-interval-1*: *dest-vec1* $(a$::*real^1*$) \leq$ *dest-vec1* b ==> $\{a\ ..\ b\} = \{a<..<b\} \cup \{a,b\}$
⟨*proof*⟩

lemma *closed-interval-left*: **fixes** b::*real^'n*::*finite*
shows *closed* $\{x$::*real^'n*$.\ \forall i.\ x\$i \leq b\$i\}$
⟨*proof*⟩

lemma *closed-interval-right*: **fixes** a::*real^'n*::*finite*
shows *closed* $\{x$::*real^'n*$.\ \forall i.\ a\$i \leq x\$i\}$
⟨*proof*⟩

64.36 Intervals in general, including infinite and mixtures of open and closed.

definition *is-interval* $s \longleftrightarrow (\forall\, a{\in}s.\ \forall\, b{\in}s.\ \forall\, x.\ a \le x \wedge x \le b \longrightarrow x \in s)$

lemma *is-interval-interval*: **fixes** *a*::*real^'n*::*finite* **shows**
is-interval $\{a<..<b\}$ *is-interval* $\{a\ ..\ b\}$
⟨*proof*⟩

lemma *is-interval-empty*:
is-interval $\{\}$
⟨*proof*⟩

lemma *is-interval-univ*:
is-interval UNIV
⟨*proof*⟩

64.37 Closure of halfspaces and hyperplanes.

lemma *Lim-vec1-dot*: **fixes** *f* :: *real^'m* ⇒ *real^'n*::*finite*
assumes (*f* −−−> *l*) *net* **shows** ((*vec1 o* (λ*y*. *a* · (*f y*))) −−−> *vec1*(*a* · *l*)) *net*
⟨*proof*⟩

lemma *continuous-at-vec1-dot*:
continuous (*at x*) (*vec1 o* (λ*y*. *a* · *y*))
⟨*proof*⟩

lemma *continuous-on-vec1-dot*:
continuous-on s (*vec1 o* (λ*y*. *a* · *y*))
⟨*proof*⟩

lemma *closed-halfspace-le*: **fixes** *a*::*real^'n*::*finite*
shows *closed* $\{x.\ a \cdot x \le b\}$
⟨*proof*⟩

lemma *closed-halfspace-ge*: *closed* $\{x.\ a \cdot x \ge b\}$
⟨*proof*⟩

lemma *closed-hyperplane*: *closed* $\{x.\ a \cdot x = b\}$
⟨*proof*⟩

lemma *closed-halfspace-component-le*:
shows *closed* $\{x$::*real^'n*::*finite*. $x\$i \le a\}$
⟨*proof*⟩

lemma *closed-halfspace-component-ge*:
shows *closed* $\{x$::*real^'n*::*finite*. $x\$i \ge a\}$
⟨*proof*⟩

Openness of halfspaces.

lemma *open-halfspace-lt*: *open* $\{x.\ a \cdot x < b\}$
⟨*proof*⟩

lemma *open-halfspace-gt*: *open* $\{x.\ a \cdot x > b\}$
⟨*proof*⟩

lemma *open-halfspace-component-lt*:
shows *open* $\{x::real\hat{}'n::finite.\ x\$i < a\}$
⟨*proof*⟩

lemma *open-halfspace-component-gt*:
shows *open* $\{x::real\hat{}'n::finite.\ x\$i > a\}$
⟨*proof*⟩

This gives a simple derivation of limit component bounds.

lemma *Lim-component-le*: **fixes** $f :: 'a \Rightarrow real\hat{}'n::finite$
assumes $(f \longrightarrow l)$ *net* $\neg$ (*trivial-limit net*) *eventually* $(\lambda x.\ f(x)\$i \leq b)$ *net*
shows $l\$i \leq b$
⟨*proof*⟩

lemma *Lim-component-ge*: **fixes** $f :: 'a \Rightarrow real\hat{}'n::finite$
assumes $(f \longrightarrow l)$ *net* $\neg$ (*trivial-limit net*) *eventually* $(\lambda x.\ b \leq (f\ x)\$i)$ *net*
shows $b \leq l\$i$
⟨*proof*⟩

lemma *Lim-component-eq*: **fixes** $f :: 'a \Rightarrow real\hat{}'n::finite$
assumes *net*:$(f \longrightarrow l)$ *net* ~(*trivial-limit net*) **and** *ev*:*eventually* $(\lambda x.\ f(x)\$i = b)$ *net*
shows $l\$i = b$
⟨*proof*⟩

lemma *Lim-drop-le*: **fixes** $f :: 'a \Rightarrow real\hat{}1$ **shows**
$(f \longrightarrow l)$ *net* $\Longrightarrow$ ~(*trivial-limit net*) $\Longrightarrow$ *eventually* $(\lambda x.\ dest\text{-}vec1\ (f\ x) \leq b)$ *net* ==> $dest\text{-}vec1\ l \leq b$
⟨*proof*⟩

lemma *Lim-drop-ge*: **fixes** $f :: 'a \Rightarrow real\hat{}1$ **shows**
$(f \longrightarrow l)$ *net* $\Longrightarrow$ ~(*trivial-limit net*) $\Longrightarrow$ *eventually* $(\lambda x.\ b \leq dest\text{-}vec1\ (f\ x))$ *net* ==> $b \leq dest\text{-}vec1\ l$
⟨*proof*⟩

Limits relative to a union.

lemma *Lim-within-union*:
$(f \longrightarrow l)\ (at\ x\ within\ (s \cup t)) \longleftrightarrow$
$(f \longrightarrow l)\ (at\ x\ within\ s) \wedge (f \longrightarrow l)\ (at\ x\ within\ t)$
⟨*proof*⟩

lemma *continuous-on-union*:

assumes *closed s closed t continuous-on s f continuous-on t f*
shows *continuous-on* $(s \cup t)$ *f*
⟨*proof*⟩

lemma *continuous-on-cases*: **fixes** *g* :: *real^'m::finite* ⇒ *real ^'n::finite*
assumes *closed s closed t continuous-on s f continuous-on t g*
$\forall x.\ (x{\in}s \wedge \neg\ P\ x) \vee (x \in t \wedge P\ x) \longrightarrow f\ x = g\ x$
shows *continuous-on* $(s \cup t)$ $(\lambda x.$ *if P x then f x else g x*$)$
⟨*proof*⟩

Some more convenient intermediate-value theorem formulations.

lemma *connected-ivt-hyperplane*: **fixes** *y* :: *real^'n::finite*
assumes *connected s x* ∈ *s y* ∈ *s a* · *x* ≤ *b b* ≤ *a* · *y*
shows ∃ *z* ∈ *s*. *a* · *z* = *b*
⟨*proof*⟩

lemma *connected-ivt-component*: **fixes** *x::real^'n::finite* **shows**
connected s ⟹ *x* ∈ *s* ⟹ *y* ∈ *s* ⟹ *x*$*k* ≤ *a* ⟹ *a* ≤ *y*$*k* ⟹ (∃ *z*∈*s*. *z*$*k* = *a*)
⟨*proof*⟩

Also more convenient formulations of monotone convergence.

lemma *bounded-increasing-convergent*: **fixes** *s::nat* ⇒ *real^1*
assumes *bounded* {*s n*| *n::nat. True*} ∀ *n*. *dest-vec1*(*s n*) ≤ *dest-vec1*(*s*(*Suc n*))
shows ∃ *l*. (*s* −−−> *l*) *sequentially*
⟨*proof*⟩

64.38 Basic homeomorphism definitions.

definition *homeomorphism s t f g* ≡
(∀ *x*∈*s*. (*g*(*f x*) = *x*)) ∧ (*f* ‘ *s* = *t*) ∧ *continuous-on s f* ∧
(∀ *y*∈*t*. (*f*(*g y*) = *y*)) ∧ (*g* ‘ *t* = *s*) ∧ *continuous-on t g*

definition *homeomorphic* :: ((*real^'a::finite*) *set*) ⇒ ((*real^'b::finite*) *set*) ⇒ *bool* (**infixr** *homeomorphic 60*) **where**
homeomorphic-def: *s homeomorphic t* ≡ (∃ *f g*. *homeomorphism s t f g*)

lemma *homeomorphic-refl*: *s homeomorphic s*
⟨*proof*⟩

lemma *homeomorphic-sym*:
s homeomorphic t ⟷ *t homeomorphic s*
⟨*proof*⟩

lemma *homeomorphic-trans*:
assumes *s homeomorphic t t homeomorphic u* **shows** *s homeomorphic u*
⟨*proof*⟩

lemma *homeomorphic-minimal*:
s homeomorphic t ⟷

$(\exists f\ g.\ (\forall x{\in}s.\ f(x) \in t \wedge (g(f(x)) = x)) \wedge$
$(\forall y{\in}t.\ g(y) \in s \wedge (f(g(y)) = y)) \wedge$
continuous-on s f $\wedge$ *continuous-on t g*)
⟨*proof*⟩

64.39 Relatively weak hypotheses if a set is compact.

definition *inv-on f s* $= (\lambda x.\ SOME\ y.\ y{\in}s \wedge f\ y = x)$

lemma assumes *inj-on f s x*$\in$*s*
shows *inv-on f s* (*f x*) $= x$
⟨*proof*⟩

lemma *homeomorphism-compact*:
assumes *compact s continuous-on s f* $f\ `\ s = t$ *inj-on f s*
shows $\exists g.$ *homeomorphism s t f g*
⟨*proof*⟩

lemma *homeomorphic-compact*:
compact s $\Longrightarrow$ *continuous-on s f* $\Longrightarrow$ $(f\ `\ s = t) \Longrightarrow$ *inj-on f s*
$\Longrightarrow$ *s homeomorphic t*
⟨*proof*⟩

Preservation of topological properties.

lemma *homeomorphic-compactness*:
s homeomorphic t ==> (*compact s* $\longleftrightarrow$ *compact t*)
⟨*proof*⟩

Results on translation, scaling etc.

lemma *homeomorphic-scaling*:
assumes $c \neq 0$ **shows** *s homeomorphic* $((\lambda x.\ c *s x)\ `\ s)$
⟨*proof*⟩

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

lemma *homeomorphic-affinity*:
assumes $c \neq 0$ **shows** *s homeomorphic* $((\lambda x.\ a + c *s x)\ `\ s)$
⟨*proof*⟩

lemma *homeomorphic-balls*: **fixes** *a b* ::*real^'a*::*finite*
assumes $0 < d$ $0 < e$
shows (*ball a d*) *homeomorphic* (*ball b e*) (**is** *?th*)
(*cball a d*) *homeomorphic* (*cball b e*) (**is** *?cth*)
⟨*proof*⟩

"Isometry" (up to constant bounds) of injective linear map etc.

lemma *cauchy-isometric*:

assumes *e*:*0 < e* **and** *s*:*subspace s* **and** *f*:*linear f* **and** *normf*:$\forall\, x{\in}s.\ norm(f\ x) \geq e * norm(x)$ **and** *xs*:$\forall\, n{::}nat.\ x\ n \in s$ **and** *cf*:*cauchy*(*f o x*)
shows *cauchy x*
⟨*proof*⟩

lemma *complete-isometric-image*:
assumes *0 < e* **and** *s*:*subspace s* **and** *f*:*linear f* **and** *normf*:$\forall\, x{\in}s.\ norm(f\ x) \geq e * norm(x)$ **and** *cs*:*complete s*
shows *complete*(*f ' s*)
⟨*proof*⟩

lemma *dist-0-norm*:*dist 0 x = norm x* ⟨*proof*⟩

lemma *injective-imp-isometric*: **fixes** *f*::*real^'m*::*finite* ⇒ *real^'n*::*finite*
assumes *s*:*closed s subspace s* **and** *f*:*linear f* $\forall\, x{\in}s.\ (f\ x = 0) \longrightarrow (x = 0)$
shows $\exists\, e{>}0.\ \forall\, x{\in}s.\ norm\ (f\ x) \geq e * norm(x)$
⟨*proof*⟩

lemma *closed-injective-image-subspace*:
assumes *subspace s linear f* $\forall\, x{\in}s.\ f\ x = 0$ *--> x = 0 closed s*
shows *closed*(*f ' s*)
⟨*proof*⟩

64.40 Some properties of a canonical subspace.

lemma *subspace-substandard*:
subspace $\{x{::}real\hat{}'n.\ (\forall\, i.\ P\ i \longrightarrow x\$i = 0)\}$
⟨*proof*⟩

lemma *closed-substandard*:
closed $\{x{::}real\hat{}'n{::}finite.\ \forall\, i.\ P\ i$ *--> x$i = 0*$\}$ (**is** *closed ?A*)
⟨*proof*⟩

lemma *dim-substandard*:
shows *dim* $\{x{::}real\hat{}'n{::}finite.\ \forall\, i.\ i \notin d \longrightarrow x\$i = 0\} = card\ d$ (**is** *dim ?A = -*)
⟨*proof*⟩

Hence closure and completeness of all subspaces.

lemma *closed-subspace-lemma*: $n \leq card\ (UNIV{::}'n{::}finite\ set) \Longrightarrow \exists\, A{::}'n\ set.\ card\ A = n$
⟨*proof*⟩

lemma *closed-subspace*: **fixes** *s*::(*real^'n*::*finite*) *set*
assumes *subspace s* **shows** *closed s*
⟨*proof*⟩

lemma *complete-subspace*:
subspace s ==> complete s
⟨*proof*⟩

lemma *dim-closure*:
dim(*closure s*) = *dim s* (**is** *?dc* = *?d*)
⟨*proof*⟩

Affine transformations of intervals.

lemma *affinity-inverses*:
assumes *m0*: *m* ≠ (*0*::'*a*::*field*)
shows (λ*x*. *m* ∗*s x* + *c*) *o* (λ*x*. *inverse*(*m*) ∗*s x* + (−(*inverse*(*m*) ∗*s c*))) = *id*
(λ*x*. *inverse*(*m*) ∗*s x* + (−(*inverse*(*m*) ∗*s c*))) *o* (λ*x*. *m* ∗*s x* + *c*) = *id*
⟨*proof*⟩

lemma *real-affinity-le*:
0 < (*m*::'*a*::*ordered-field*) ==> (*m* ∗ *x* + *c* ≤ *y* ⟷ *x* ≤ *inverse*(*m*) ∗ *y* + −(*c* / *m*))
⟨*proof*⟩

lemma *real-le-affinity*:
0 < (*m*::'*a*::*ordered-field*) ==> (*y* ≤ *m* ∗ *x* + *c* ⟷ *inverse*(*m*) ∗ *y* + −(*c* / *m*) ≤ *x*)
⟨*proof*⟩

lemma *real-affinity-lt*:
0 < (*m*::'*a*::*ordered-field*) ==> (*m* ∗ *x* + *c* < *y* ⟷ *x* < *inverse*(*m*) ∗ *y* + −(*c* / *m*))
⟨*proof*⟩

lemma *real-lt-affinity*:
0 < (*m*::'*a*::*ordered-field*) ==> (*y* < *m* ∗ *x* + *c* ⟷ *inverse*(*m*) ∗ *y* + −(*c* / *m*) < *x*)
⟨*proof*⟩

lemma *real-affinity-eq*:
(*m*::'*a*::*ordered-field*) ≠ *0* ==> (*m* ∗ *x* + *c* = *y* ⟷ *x* = *inverse*(*m*) ∗ *y* + −(*c* / *m*))
⟨*proof*⟩

lemma *real-eq-affinity*:
(*m*::'*a*::*ordered-field*) ≠ *0* ==> (*y* = *m* ∗ *x* + *c* ⟷ *inverse*(*m*) ∗ *y* + −(*c* / *m*) = *x*)
⟨*proof*⟩

lemma *vector-affinity-eq*:
assumes *m0*: (*m*::'*a*::*field*) ≠ *0*
shows *m* ∗*s x* + *c* = *y* ⟷ *x* = *inverse m* ∗*s y* + −(*inverse m* ∗*s c*)
⟨*proof*⟩

lemma *vector-eq-affinity*:
(*m*::'*a*::*field*) ≠ *0* ==> (*y* = *m* ∗*s x* + *c* ⟷ *inverse*(*m*) ∗*s y* + −(*inverse*(*m*) ∗*s c*) = *x*)

⟨*proof*⟩

lemma *image-affinity-interval*: **fixes** *m*::*real*
fixes *a b c* :: *real^'n*::*finite*
shows (λ*x*. *m* ∗*s x* + *c*) ' {*a* .. *b*} =
(*if* {*a* .. *b*} = {} *then* {}
else (*if* 0 ≤ *m then* {*m* ∗*s a* + *c* .. *m* ∗*s b* + *c*}
else {*m* ∗*s b* + *c* .. *m* ∗*s a* + *c*}))
⟨*proof*⟩

64.41 Banach fixed point theorem (not really topological...)

lemma *banach-fix*:
assumes *s*:*complete s s* ≠ {} **and** *c*:*0* ≤ *c c* < *1* **and** *f*:(*f* ' *s*) ⊆ *s* **and**
lipschitz:∀ *x*∈*s*. ∀ *y*∈*s*. *dist* (*f x*) (*f y*) ≤ *c* ∗ *dist x y*
shows ∃! *x*∈*s*. (*f x* = *x*)
⟨*proof*⟩

64.42 Edelstein fixed point theorem.

lemma *edelstein-fix*:
assumes *s*:*compact s s* ≠ {} **and** *gs*:(*g* ' *s*) ⊆ *s*
and *dist*:∀ *x*∈*s*. ∀ *y*∈*s*. *x* ≠ *y* ⟶ *dist* (*g x*) (*g y*) < *dist x y*
shows ∃! *x*::*real^'a*::*finite*∈*s*. *g x* = *x*
⟨*proof*⟩

end

65 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*

65.1 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 *recpower-semiring* = *semiring* + *recpower*
class *recpower-semiring-1* = *semiring-1* + *recpower*
class *recpower-semiring-0* = *semiring-0* + *recpower*
class *recpower-ring* = *ring* + *recpower*
class *recpower-ring-1* = *ring-1* + *recpower*
subclass (**in** *recpower-ring-1*) *recpower-ring* ⟨*proof*⟩
class *recpower-comm-semiring-1* = *recpower* + *comm-semiring-1*
class *recpower-comm-ring-1* = *recpower* + *comm-ring-1*
subclass (**in** *recpower-comm-ring-1*) *recpower-comm-semiring-1* ⟨*proof*⟩
class *recpower-idom* = *recpower* + *idom*
subclass (**in** *recpower-idom*) *recpower-comm-ring-1* ⟨*proof*⟩
class *idom-char-0* = *idom* + *ring-char-0*
class *recpower-idom-char-0* = *recpower* + *idom-char-0*
subclass (**in** *recpower-idom-char-0*) *recpower-idom* ⟨*proof*⟩

lemma (**in** *recpower-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*⟩

65.2 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*⟩

65.3 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* = []) −−> (*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*: $(f = g) = (\forall x.\ f\ x = g\ x)$
⟨*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*: $(poly\ (p *\!*\!* q) = poly\ (p *\!*\!* r))$ $= (poly\ p = poly\ [] \mid poly\ q = poly\ r)$
⟨*proof*⟩

lemma (**in** *recpower-idom*) *poly-exp-eq-zero*[*simp*]:
$(poly\ (p\ \%\hat{}\ n) = poly\ []) = (poly\ p = poly\ [] \ \&\ n \neq 0)$
⟨*proof*⟩

lemma (**in** *semiring-1*) *one-neq-zero*[*simp*]: $1 \neq 0$ ⟨*proof*⟩
lemma (**in** *comm-ring-1*) *poly-prime-eq-zero*[*simp*]: $poly\ [a,1] \neq poly\ []$
⟨*proof*⟩

lemma (**in** *recpower-idom*) *poly-exp-prime-eq-zero*: $(poly\ ([a,\ 1]\ \%\hat{}\ n) \neq poly\ [])$
⟨*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*: $(poly\ p = poly\ []) = list\text{-}all\ (\%c.\ c = 0)\ p$
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-0*: $list\text{-}all\ (\lambda c.\ c = 0)\ p \Longrightarrow poly\ p\ x = 0$
⟨*proof*⟩

Basics of divisibility.

lemma (**in** *idom*) *poly-primes*: $([a,\ 1]\ divides\ (p *\!*\!* q)) = ([a,\ 1]\ divides\ p \mid [a,\ 1]\ divides\ q)$
⟨*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** *recpower-comm-semiring-1*) *poly-divides-exp*: $m \leq n$ ==> $(p \,\%\hat{}\, m)$ *divides* $(p \,\%\hat{}\, n)$
⟨*proof*⟩

lemma (**in** *recpower-comm-semiring-1*) *poly-exp-divides*: [| $(p \,\%\hat{}\, n)$ *divides* q; $m \leq n$ |] ==> $(p \,\%\hat{}\, m)$ *divides* q
⟨*proof*⟩

lemma (**in** *comm-semiring-0*) *poly-divides-add*:
[| p *divides* q; p *divides* r |] ==> p *divides* $(q +++ r)$
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-divides-diff*:
[| p *divides* q; p *divides* $(q +++ r)$ |] ==> p *divides* r
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-divides-diff2*: [| p *divides* r; p *divides* $(q +++ r)$ |] ==> p *divides* q
⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-divides-zero*: *poly* p = *poly* [] ==> 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** *recpower-comm-semiring-1*) *poly-mulexp*: *poly* (*mulexp* $n\ p\ q$) $x = (\textit{poly}\ p\ x)\ \hat{}\ n *$ *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** *recpower-semiring-1*)

zero-power-iff: $0 \hat{} n = (\mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ 0)$
⟨*proof*⟩

lemma (**in** *recpower-idom-char-0*) *poly-order-exists*:
assumes *lp*: *length p* = *d* **and** *p0*: *poly p* ≠ *poly* []
shows ∃ *n*. ([−*a*, *1*] % ^ *n*) *divides p* & ~(([−*a*, *1*] % ^ (*Suc n*)) *divides p*)
⟨*proof*⟩

lemma (**in** *semiring-1*) *poly-one-divides*[*simp*]: [*1*] *divides p*
⟨*proof*⟩

lemma (**in** *recpower-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** *recpower-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** *recpower-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** *recpower-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** *recpower-idom-char-0*) *order-unique-lemma*: (*poly p* ≠ *poly* [] & ([−*a*, *1*] % ^ *n*) *divides p* &
~(([−*a*, *1*] % ^ (*Suc n*)) *divides p*))
==> (*n* = *order a p*)
⟨*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] %^ n divides p & ~ [− a, 1] %^ (Suc n) divides p
⟹ poly p a = 0
⟨*proof*⟩

lemma (**in** *recpower-idom-char-0*) *order-root*: *(poly p a = 0) = ((poly p = poly []) | order a p ≠ 0)*
⟨*proof*⟩

lemma (**in** *recpower-idom-char-0*) *order-divides*: *(([−a, 1] %^ n) divides p) = ((poly p = poly []) | n ≤ order a p)*
⟨*proof*⟩

lemma (**in** *recpower-idom-char-0*) *order-decomp*:
poly p ≠ poly []
*==> ∃ q. (poly p = poly (([−a, 1] %^ (order a p)) *** q)) &*
~([−a, 1] divides q)
⟨*proof*⟩

Important composition properties of orders.

lemma *order-mult*: *poly (p *** q) ≠ poly []*
*==> order a (p *** q) = order a p + order (a::'a::{recpower-idom-char-0}) q*
⟨*proof*⟩

lemma (**in** *recpower-idom-char-0*) *order-mult*:
assumes *pq0*: *poly (p *** q) ≠ poly []*
shows *order a (p *** q) = order a p + order a q*
⟨*proof*⟩

lemma (**in** *recpower-idom-char-0*) *order-root2*: *poly p ≠ poly [] ==> (poly p a = 0) = (order a p ≠ 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** *recpower-idom-char-0*) *rsquarefree-decomp*:
[| rsquarefree p; poly p a = 0 |]
*==> ∃ q. (poly p = poly ([−a, 1] *** q)) & poly q a ≠ 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* ⟷ *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** *recpower-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*::{*ordered-idom*})) ≤ *poly* (*map abs p*) *k*
⟨*proof*⟩

lemma (**in** *semiring-0*) *poly-Sing*: *poly* [*c*] *x* = *c* ⟨*proof*⟩

end

66 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

67 Word: Binary Words

theory *Word*
imports ~~/*src*/*HOL*/*Main*
begin

67.1 Auxilary Lemmas

lemma *max-le* [*intro*!]: [| $x \leq z$; $y \leq z$ |] ==> *max* x $y \leq z$
⟨*proof*⟩

lemma *max-mono*:
fixes x :: 'a::*linorder*
assumes *mf*: *mono* f
shows *max* (f x) (f y) $\leq$ f (*max* x y)
⟨*proof*⟩

declare *zero-le-power* [*intro*]
and *zero-less-power* [*intro*]

lemma *int-nat-two-exp*: *2* ^ k = *int* (*2* ^ k)
⟨*proof*⟩

67.2 Bits

datatype *bit* =
Zero (**0**)
| *One* (**1**)

primrec
bitval :: *bit* => *nat*
where
bitval **0** = *0*
| *bitval* **1** = *1*

consts
bitnot :: *bit* => *bit*
bitand :: *bit* => *bit* => *bit* (**infixr** *bitand* *35*)
bitor :: *bit* => *bit* => *bit* (**infixr** *bitor* *30*)
bitxor :: *bit* => *bit* => *bit* (**infixr** *bitxor* *30*)

notation (*xsymbols*)
bitnot ($\neg_b$ - [*40*] *40*) **and**
bitand (**infixr** $\wedge_b$ *35*) **and**
bitor (**infixr** $\vee_b$ *30*) **and**
bitxor (**infixr** $\oplus_b$ *30*)

notation (*HTML* **output**)
bitnot ($\neg_b$ - [*40*] *40*) **and**
bitand (**infixr** $\wedge_b$ *35*) **and**

bitor (**infixr** $\vee_b$ *30*) **and**
bitxor (**infixr** $\oplus_b$ *30*)

primrec
bitnot-zero: (*bitnot* **0**) = **1**
bitnot-one : (*bitnot* **1**) = **0**

primrec
bitand-zero: (**0** *bitand* *y*) = **0**
bitand-one: (**1** *bitand* *y*) = *y*

primrec
bitor-zero: (**0** *bitor* *y*) = *y*
bitor-one: (**1** *bitor* *y*) = **1**

primrec
bitxor-zero: (**0** *bitxor* *y*) = *y*
bitxor-one: (**1** *bitxor* *y*) = (*bitnot* *y*)

lemma *bitnot-bitnot* [*simp*]: (*bitnot* (*bitnot* *b*)) = *b*
⟨*proof*⟩

lemma *bitand-cancel* [*simp*]: (*b* *bitand* *b*) = *b*
⟨*proof*⟩

lemma *bitor-cancel* [*simp*]: (*b* *bitor* *b*) = *b*
⟨*proof*⟩

lemma *bitxor-cancel* [*simp*]: (*b* *bitxor* *b*) = **0**
⟨*proof*⟩

67.3 Bit Vectors

First, a couple of theorems expressing case analysis and induction principles for bit vectors.

lemma *bit-list-cases*:
assumes *empty*: *w* = [] ==> *P* *w*
and *zero*: !!*bs*. *w* = **0** # *bs* ==> *P* *w*
and *one*: !!*bs*. *w* = **1** # *bs* ==> *P* *w*
shows *P* *w*
⟨*proof*⟩

lemma *bit-list-induct*:
assumes *empty*: *P* []
and *zero*: !!*bs*. *P* *bs* ==> *P* (**0**#*bs*)
and *one*: !!*bs*. *P* *bs* ==> *P* (**1**#*bs*)
shows *P* *w*
⟨*proof*⟩

definition
bv-msb :: *bit list* => *bit* **where**
bv-msb w = (*if w* = [] *then* **0** *else hd w*)

definition
bv-extend :: [*nat*,*bit*,*bit list*]=>*bit list* **where**
bv-extend i b w = (*replicate* (*i* − *length w*) *b*) @ *w*

definition
bv-not :: *bit list* => *bit list* **where**
bv-not w = *map bitnot w*

lemma *bv-length-extend* [*simp*]: *length w* ≤ *i* ==> *length* (*bv-extend i b w*) = *i*
⟨*proof*⟩

lemma *bv-not-Nil* [*simp*]: *bv-not* [] = []
⟨*proof*⟩

lemma *bv-not-Cons* [*simp*]: *bv-not* (*b*#*bs*) = (*bitnot b*) # *bv-not bs*
⟨*proof*⟩

lemma *bv-not-bv-not* [*simp*]: *bv-not* (*bv-not w*) = *w*
⟨*proof*⟩

lemma *bv-msb-Nil* [*simp*]: *bv-msb* [] = **0**
⟨*proof*⟩

lemma *bv-msb-Cons* [*simp*]: *bv-msb* (*b*#*bs*) = *b*
⟨*proof*⟩

lemma *bv-msb-bv-not* [*simp*]: *0* < *length w* ==> *bv-msb* (*bv-not w*) = (*bitnot* (*bv-msb w*))
⟨*proof*⟩

lemma *bv-msb-one-length* [*simp*,*intro*]: *bv-msb w* = **1** ==> *0* < *length w*
⟨*proof*⟩

lemma *length-bv-not* [*simp*]: *length* (*bv-not w*) = *length w*
⟨*proof*⟩

definition
bv-to-nat :: *bit list* => *nat* **where**
bv-to-nat = *foldl* (%*bn b*. *2* ∗ *bn* + *bitval b*) *0*

lemma *bv-to-nat-Nil* [*simp*]: *bv-to-nat* [] = *0*
⟨*proof*⟩

lemma *bv-to-nat-helper* [*simp*]: *bv-to-nat* (*b* # *bs*) = *bitval b* ∗ *2* ^ *length bs* + *bv-to-nat bs*

⟨*proof*⟩

lemma *bv-to-nat0* [*simp*]: *bv-to-nat* (**0**#*bs*) = *bv-to-nat bs*
⟨*proof*⟩

lemma *bv-to-nat1* [*simp*]: *bv-to-nat* (**1**#*bs*) = *2* ^ *length bs* + *bv-to-nat bs*
⟨*proof*⟩

lemma *bv-to-nat-upper-range*: *bv-to-nat w* < *2* ^ *length w*
⟨*proof*⟩

lemma *bv-extend-longer* [*simp*]:
assumes *wn*: *n* ≤ *length w*
shows *bv-extend n b w* = *w*
⟨*proof*⟩

lemma *bv-extend-shorter* [*simp*]:
assumes *wn*: *length w* < *n*
shows *bv-extend n b w* = *bv-extend n b* (*b*#*w*)
⟨*proof*⟩

consts
rem-initial :: *bit* => *bit list* => *bit list*
primrec
rem-initial b [] = []
rem-initial b (*x*#*xs*) = (*if b* = *x then rem-initial b xs else x*#*xs*)

lemma *rem-initial-length*: *length* (*rem-initial b w*) ≤ *length w*
⟨*proof*⟩

lemma *rem-initial-equal*:
assumes *p*: *length* (*rem-initial b w*) = *length w*
shows *rem-initial b w* = *w*
⟨*proof*⟩

lemma *bv-extend-rem-initial*: *bv-extend* (*length w*) *b* (*rem-initial b w*) = *w*
⟨*proof*⟩

lemma *rem-initial-append1*:
assumes *rem-initial b xs* ~= []
shows *rem-initial b* (*xs* @ *ys*) = *rem-initial b xs* @ *ys*
⟨*proof*⟩

lemma *rem-initial-append2*:
assumes *rem-initial b xs* = []
shows *rem-initial b* (*xs* @ *ys*) = *rem-initial b ys*
⟨*proof*⟩

definition

norm-unsigned :: *bit list* => *bit list* **where**
norm-unsigned = *rem-initial* **0**

lemma *norm-unsigned-Nil* [*simp*]: *norm-unsigned* [] = []
⟨*proof*⟩

lemma *norm-unsigned-Cons0* [*simp*]: *norm-unsigned* (**0**#*bs*) = *norm-unsigned bs*
⟨*proof*⟩

lemma *norm-unsigned-Cons1* [*simp*]: *norm-unsigned* (**1**#*bs*) = **1**#*bs*
⟨*proof*⟩

lemma *norm-unsigned-idem* [*simp*]: *norm-unsigned* (*norm-unsigned w*) = *norm-unsigned w*
⟨*proof*⟩

consts
nat-to-bv-helper :: *nat* => *bit list* => *bit list*
recdef *nat-to-bv-helper measure* (λ*n*. *n*)
nat-to-bv-helper n = (%*bs*. (*if n* = *0 then bs*
else nat-to-bv-helper (*n div 2*) ((*if n mod 2* = *0 then* **0** *else* **1**)#*bs*)))

definition
nat-to-bv :: *nat* => *bit list* **where**
nat-to-bv n = *nat-to-bv-helper n* []

lemma *nat-to-bv0* [*simp*]: *nat-to-bv 0* = []
⟨*proof*⟩

lemmas [*simp del*] = *nat-to-bv-helper.simps*

lemma *n-div-2-cases*:
assumes *zero*: (*n*::*nat*) = *0* ==> *R*
and *div* : [| *n div 2* < *n* ; *0* < *n* |] ==> *R*
shows *R*
⟨*proof*⟩

lemma *int-wf-ge-induct*:
assumes *ind* : !!*i*::*int*. (!!*j*. [| *k* ≤ *j* ; *j* < *i* |] ==> *P j*) ==> *P i*
shows *P i*
⟨*proof*⟩

lemma *unfold-nat-to-bv-helper*:
nat-to-bv-helper b l = *nat-to-bv-helper b* [] @ *l*
⟨*proof*⟩

lemma *nat-to-bv-non0* [*simp*]: *n*≠*0* ==> *nat-to-bv n* = *nat-to-bv* (*n div 2*) @ [*if n mod 2* = *0 then* **0** *else* **1**]

⟨*proof*⟩

lemma *bv-to-nat-dist-append*:
bv-to-nat (*l1* @ *l2*) = *bv-to-nat l1* ∗ *2* ^ *length l2* + *bv-to-nat l2*
⟨*proof*⟩

lemma *bv-nat-bv* [*simp*]: *bv-to-nat* (*nat-to-bv n*) = *n*
⟨*proof*⟩

lemma *bv-to-nat-type* [*simp*]: *bv-to-nat* (*norm-unsigned w*) = *bv-to-nat w*
⟨*proof*⟩

lemma *length-norm-unsigned-le* [*simp*]: *length* (*norm-unsigned w*) ≤ *length w*
⟨*proof*⟩

lemma *bv-to-nat-rew-msb*: *bv-msb w* = **1** ==> *bv-to-nat w* = *2* ^ (*length w* − *1*)
+ *bv-to-nat* (*tl w*)
⟨*proof*⟩

lemma *norm-unsigned-result*: *norm-unsigned xs* = [] ∨ *bv-msb* (*norm-unsigned xs*)
= **1**
⟨*proof*⟩

lemma *norm-empty-bv-to-nat-zero*:
assumes *nw*: *norm-unsigned w* = []
shows *bv-to-nat w* = *0*
⟨*proof*⟩

lemma *bv-to-nat-lower-limit*:
assumes *w0*: *0* < *bv-to-nat w*
shows *2* ^ (*length* (*norm-unsigned w*) − *1*) ≤ *bv-to-nat w*
⟨*proof*⟩

lemmas [*simp del*] = *nat-to-bv-non0*

lemma *norm-unsigned-length* [*intro*!]: *length* (*norm-unsigned w*) ≤ *length w*
⟨*proof*⟩

lemma *norm-unsigned-equal*:
length (*norm-unsigned w*) = *length w* ==> *norm-unsigned w* = *w*
⟨*proof*⟩

lemma *bv-extend-norm-unsigned*: *bv-extend* (*length w*) **0** (*norm-unsigned w*) = *w*
⟨*proof*⟩

lemma *norm-unsigned-append1* [*simp*]:
norm-unsigned xs ≠ [] ==> *norm-unsigned* (*xs* @ *ys*) = *norm-unsigned xs* @ *ys*
⟨*proof*⟩

lemma *norm-unsigned-append2* [*simp*]:
norm-unsigned xs = [] ==> norm-unsigned (xs @ ys) = norm-unsigned ys
⟨*proof*⟩

lemma *bv-to-nat-zero-imp-empty*:
bv-to-nat w = 0 ⟹ norm-unsigned w = []
⟨*proof*⟩

lemma *bv-to-nat-nzero-imp-nempty*:
bv-to-nat w ≠ 0 ⟹ norm-unsigned w ≠ []
⟨*proof*⟩

lemma *nat-helper1*:
assumes *ass*: *nat-to-bv (bv-to-nat w) = norm-unsigned w*
shows *nat-to-bv (2 * bv-to-nat w + bitval x) = norm-unsigned (w @ [x])*
⟨*proof*⟩

lemma *nat-helper2*: *nat-to-bv (2 ^ length xs + bv-to-nat xs) = 1 # xs*
⟨*proof*⟩

lemma *nat-bv-nat* [*simp*]: *nat-to-bv (bv-to-nat w) = norm-unsigned w*
⟨*proof*⟩

lemma *bv-to-nat-qinj*:
assumes *one*: *bv-to-nat xs = bv-to-nat ys*
and *len*: *length xs = length ys*
shows *xs = ys*
⟨*proof*⟩

lemma *norm-unsigned-nat-to-bv* [*simp*]:
norm-unsigned (nat-to-bv n) = nat-to-bv n
⟨*proof*⟩

lemma *length-nat-to-bv-upper-limit*:
assumes *nk*: *n ≤ 2 ^ k - 1*
shows *length (nat-to-bv n) ≤ k*
⟨*proof*⟩

lemma *length-nat-to-bv-lower-limit*:
assumes *nk*: *2 ^ k ≤ n*
shows *k < length (nat-to-bv n)*
⟨*proof*⟩

67.4 Unsigned Arithmetic Operations

definition
bv-add :: [bit list, bit list] => bit list **where**
bv-add w1 w2 = nat-to-bv (bv-to-nat w1 + bv-to-nat w2)

lemma *bv-add-type1* [*simp*]: *bv-add* (*norm-unsigned w1*) *w2* = *bv-add w1 w2*
⟨*proof*⟩

lemma *bv-add-type2* [*simp*]: *bv-add w1* (*norm-unsigned w2*) = *bv-add w1 w2*
⟨*proof*⟩

lemma *bv-add-returntype* [*simp*]: *norm-unsigned* (*bv-add w1 w2*) = *bv-add w1 w2*
⟨*proof*⟩

lemma *bv-add-length*: *length* (*bv-add w1 w2*) ≤ *Suc* (*max* (*length w1*) (*length w2*))
⟨*proof*⟩

definition
bv-mult :: [*bit list*, *bit list*] => *bit list* **where**
bv-mult w1 w2 = *nat-to-bv* (*bv-to-nat w1* ∗ *bv-to-nat w2*)

lemma *bv-mult-type1* [*simp*]: *bv-mult* (*norm-unsigned w1*) *w2* = *bv-mult w1 w2*
⟨*proof*⟩

lemma *bv-mult-type2* [*simp*]: *bv-mult w1* (*norm-unsigned w2*) = *bv-mult w1 w2*
⟨*proof*⟩

lemma *bv-mult-returntype* [*simp*]: *norm-unsigned* (*bv-mult w1 w2*) = *bv-mult w1 w2*
⟨*proof*⟩

lemma *bv-mult-length*: *length* (*bv-mult w1 w2*) ≤ *length w1* + *length w2*
⟨*proof*⟩

67.5 Signed Vectors

consts
norm-signed :: *bit list* => *bit list*
primrec
norm-signed-Nil: *norm-signed* [] = []
norm-signed-Cons: *norm-signed* (*b*#*bs*) =
(*case b of*
0 => *if norm-unsigned bs* = [] *then* [] *else b*#*norm-unsigned bs*
| **1** => *b*#*rem-initial b bs*)

lemma *norm-signed0* [*simp*]: *norm-signed* [**0**] = []
⟨*proof*⟩

lemma *norm-signed1* [*simp*]: *norm-signed* [**1**] = [**1**]
⟨*proof*⟩

lemma *norm-signed01* [*simp*]: *norm-signed* (**0**#**1**#*xs*) = **0**#**1**#*xs*
⟨*proof*⟩

lemma *norm-signed00* [*simp*]: *norm-signed* (**0**#**0**#*xs*) = *norm-signed* (**0**#*xs*)
⟨*proof*⟩

lemma *norm-signed10* [*simp*]: *norm-signed* (**1**#**0**#*xs*) = **1**#**0**#*xs*
⟨*proof*⟩

lemma *norm-signed11* [*simp*]: *norm-signed* (**1**#**1**#*xs*) = *norm-signed* (**1**#*xs*)
⟨*proof*⟩

lemmas [*simp del*] = *norm-signed-Cons*

definition
int-to-bv :: *int* => *bit list* **where**
int-to-bv n = (*if* 0 ≤ *n*
then norm-signed (**0**#*nat-to-bv* (*nat n*))
else norm-signed (*bv-not* (**0**#*nat-to-bv* (*nat* (−*n*− 1)))))

lemma *int-to-bv-ge0* [*simp*]: 0 ≤ *n* ==> *int-to-bv n* = *norm-signed* (**0** # *nat-to-bv* (*nat n*))
⟨*proof*⟩

lemma *int-to-bv-lt0* [*simp*]:
n < 0 ==> *int-to-bv n* = *norm-signed* (*bv-not* (**0**#*nat-to-bv* (*nat* (−*n*− 1))))
⟨*proof*⟩

lemma *norm-signed-idem* [*simp*]: *norm-signed* (*norm-signed w*) = *norm-signed w*
⟨*proof*⟩

definition
bv-to-int :: *bit list* => *int* **where**
bv-to-int w =
(*case bv-msb w of* **0** => *int* (*bv-to-nat w*)
| **1** => − *int* (*bv-to-nat* (*bv-not w*) + 1))

lemma *bv-to-int-Nil* [*simp*]: *bv-to-int* [] = 0
⟨*proof*⟩

lemma *bv-to-int-Cons0* [*simp*]: *bv-to-int* (**0**#*bs*) = *int* (*bv-to-nat bs*)
⟨*proof*⟩

lemma *bv-to-int-Cons1* [*simp*]: *bv-to-int* (**1**#*bs*) = − *int* (*bv-to-nat* (*bv-not bs*) + 1)
⟨*proof*⟩

lemma *bv-to-int-type* [*simp*]: *bv-to-int* (*norm-signed w*) = *bv-to-int w*
⟨*proof*⟩

lemma *bv-to-int-upper-range*: *bv-to-int w* < 2 ^ (*length w* − 1)
⟨*proof*⟩

lemma *bv-to-int-lower-range*: $-(2\ \hat{}\ (length\ w - 1)) \leq bv\text{-}to\text{-}int\ w$
⟨*proof*⟩

lemma *int-bv-int* [*simp*]: $int\text{-}to\text{-}bv\ (bv\text{-}to\text{-}int\ w) = norm\text{-}signed\ w$
⟨*proof*⟩

lemma *bv-int-bv* [*simp*]: $bv\text{-}to\text{-}int\ (int\text{-}to\text{-}bv\ i) = i$
⟨*proof*⟩

lemma *bv-msb-norm* [*simp*]: $bv\text{-}msb\ (norm\text{-}signed\ w) = bv\text{-}msb\ w$
⟨*proof*⟩

lemma *norm-signed-length*: $length\ (norm\text{-}signed\ w) \leq length\ w$
⟨*proof*⟩

lemma *norm-signed-equal*: $length\ (norm\text{-}signed\ w) = length\ w \Longrightarrow norm\text{-}signed\ w = w$
⟨*proof*⟩

lemma *bv-extend-norm-signed*: $bv\text{-}msb\ w = b \Longrightarrow bv\text{-}extend\ (length\ w)\ b\ (norm\text{-}signed\ w) = w$
⟨*proof*⟩

lemma *bv-to-int-qinj*:
assumes *one*: $bv\text{-}to\text{-}int\ xs = bv\text{-}to\text{-}int\ ys$
and *len*: $length\ xs = length\ ys$
shows $xs = ys$
⟨*proof*⟩

lemma *int-to-bv-returntype* [*simp*]: $norm\text{-}signed\ (int\text{-}to\text{-}bv\ w) = int\text{-}to\text{-}bv\ w$
⟨*proof*⟩

lemma *bv-to-int-msb0*: $0 \leq bv\text{-}to\text{-}int\ w1 \Longrightarrow bv\text{-}msb\ w1 = \mathbf{0}$
⟨*proof*⟩

lemma *bv-to-int-msb1*: $bv\text{-}to\text{-}int\ w1 < 0 \Longrightarrow bv\text{-}msb\ w1 = \mathbf{1}$
⟨*proof*⟩

lemma *bv-to-int-lower-limit-gt0*:
assumes *w0*: $0 < bv\text{-}to\text{-}int\ w$
shows $2\ \hat{}\ (length\ (norm\text{-}signed\ w) - 2) \leq bv\text{-}to\text{-}int\ w$
⟨*proof*⟩

lemma *norm-signed-result*: $norm\text{-}signed\ w = [] \vee norm\text{-}signed\ w = [\mathbf{1}] \vee bv\text{-}msb\ (norm\text{-}signed\ w) \neq bv\text{-}msb\ (tl\ (norm\text{-}signed\ w))$
⟨*proof*⟩

lemma *bv-to-int-upper-limit-lem1*:

assumes *w0*: *bv-to-int w* < −*1*
shows *bv-to-int w* < − (*2* ^ (*length* (*norm-signed w*) − *2*))
⟨*proof*⟩

lemma *length-int-to-bv-upper-limit-gt0*:
assumes *w0*: *0* < *i*
and *wk*: *i* ≤ *2* ^ (*k* − *1*) − *1*
shows *length* (*int-to-bv i*) ≤ *k*
⟨*proof*⟩

lemma *pos-length-pos*:
assumes *i0*: *0* < *bv-to-int w*
shows *0* < *length w*
⟨*proof*⟩

lemma *neg-length-pos*:
assumes *i0*: *bv-to-int w* < −*1*
shows *0* < *length w*
⟨*proof*⟩

lemma *length-int-to-bv-lower-limit-gt0*:
assumes *wk*: *2* ^ (*k* − *1*) ≤ *i*
shows *k* < *length* (*int-to-bv i*)
⟨*proof*⟩

lemma *length-int-to-bv-upper-limit-lem1*:
assumes *w1*: *i* < −*1*
and *wk*: − (*2* ^ (*k* − *1*)) ≤ *i*
shows *length* (*int-to-bv i*) ≤ *k*
⟨*proof*⟩

lemma *length-int-to-bv-lower-limit-lem1*:
assumes *wk*: *i* < −(*2* ^ (*k* − *1*))
shows *k* < *length* (*int-to-bv i*)
⟨*proof*⟩

67.6 Signed Arithmetic Operations

67.6.1 Conversion from unsigned to signed

definition
utos :: *bit list* => *bit list* **where**
utos w = *norm-signed* (**0** # *w*)

lemma *utos-type* [*simp*]: *utos* (*norm-unsigned w*) = *utos w*
⟨*proof*⟩

lemma *utos-returntype* [*simp*]: *norm-signed* (*utos w*) = *utos w*
⟨*proof*⟩

lemma *utos-length*: *length* (*utos w*) ≤ *Suc* (*length w*)
⟨*proof*⟩

lemma *bv-to-int-utos*: *bv-to-int* (*utos w*) = *int* (*bv-to-nat w*)
⟨*proof*⟩

67.6.2 Unary minus

definition
bv-uminus :: *bit list* => *bit list* **where**
bv-uminus w = *int-to-bv* (− *bv-to-int w*)

lemma *bv-uminus-type* [*simp*]: *bv-uminus* (*norm-signed w*) = *bv-uminus w*
⟨*proof*⟩

lemma *bv-uminus-returntype* [*simp*]: *norm-signed* (*bv-uminus w*) = *bv-uminus w*
⟨*proof*⟩

lemma *bv-uminus-length*: *length* (*bv-uminus w*) ≤ *Suc* (*length w*)
⟨*proof*⟩

lemma *bv-uminus-length-utos*: *length* (*bv-uminus* (*utos w*)) ≤ *Suc* (*length w*)
⟨*proof*⟩

definition
bv-sadd :: [*bit list*, *bit list*] => *bit list* **where**
bv-sadd w1 w2 = *int-to-bv* (*bv-to-int w1* + *bv-to-int w2*)

lemma *bv-sadd-type1* [*simp*]: *bv-sadd* (*norm-signed w1*) *w2* = *bv-sadd w1 w2*
⟨*proof*⟩

lemma *bv-sadd-type2* [*simp*]: *bv-sadd w1* (*norm-signed w2*) = *bv-sadd w1 w2*
⟨*proof*⟩

lemma *bv-sadd-returntype* [*simp*]: *norm-signed* (*bv-sadd w1 w2*) = *bv-sadd w1 w2*
⟨*proof*⟩

lemma *adder-helper*:
assumes *lw*: *0* < *max* (*length w1*) (*length w2*)
shows ((*2*::*int*) ^ (*length w1* − *1*)) + (*2* ^ (*length w2* − *1*)) ≤ *2* ^ *max* (*length w1*) (*length w2*)
⟨*proof*⟩

lemma *bv-sadd-length*: *length* (*bv-sadd w1 w2*) ≤ *Suc* (*max* (*length w1*) (*length w2*))
⟨*proof*⟩

definition
bv-sub :: [*bit list*, *bit list*] => *bit list* **where**

bv-sub w1 w2 = bv-sadd w1 (bv-uminus w2)

lemma *bv-sub-type1* [*simp*]: *bv-sub (norm-signed w1) w2 = bv-sub w1 w2*
⟨*proof*⟩

lemma *bv-sub-type2* [*simp*]: *bv-sub w1 (norm-signed w2) = bv-sub w1 w2*
⟨*proof*⟩

lemma *bv-sub-returntype* [*simp*]: *norm-signed (bv-sub w1 w2) = bv-sub w1 w2*
⟨*proof*⟩

lemma *bv-sub-length*: *length (bv-sub w1 w2) ≤ Suc (max (length w1) (length w2))*
⟨*proof*⟩

definition
bv-smult :: [*bit list*, *bit list*] => *bit list* **where**
*bv-smult w1 w2 = int-to-bv (bv-to-int w1 * bv-to-int w2)*

lemma *bv-smult-type1* [*simp*]: *bv-smult (norm-signed w1) w2 = bv-smult w1 w2*
⟨*proof*⟩

lemma *bv-smult-type2* [*simp*]: *bv-smult w1 (norm-signed w2) = bv-smult w1 w2*
⟨*proof*⟩

lemma *bv-smult-returntype* [*simp*]: *norm-signed (bv-smult w1 w2) = bv-smult w1 w2*
⟨*proof*⟩

lemma *bv-smult-length*: *length (bv-smult w1 w2) ≤ length w1 + length w2*
⟨*proof*⟩

lemma *bv-msb-one*: *bv-msb w = 1 ==> bv-to-nat w ≠ 0*
⟨*proof*⟩

lemma *bv-smult-length-utos*: *length (bv-smult (utos w1) w2) ≤ length w1 + length w2*
⟨*proof*⟩

lemma *bv-smult-sym*: *bv-smult w1 w2 = bv-smult w2 w1*
⟨*proof*⟩

67.7 Structural operations

definition
bv-select :: [*bit list*,*nat*] => *bit* **where**
bv-select w i = w ! (length w − 1 − i)

definition
bv-chop :: [*bit list*,*nat*] => *bit list * bit list* **where**

bv-chop w i = (*let len* = *length w in* (*take* (*len* − *i*) *w*,*drop* (*len* − *i*) *w*))

definition
bv-slice :: [*bit list*,*nat*∗*nat*] => *bit list* **where**
bv-slice w = (λ(*b*,*e*). *fst* (*bv-chop* (*snd* (*bv-chop w* (*b*+*1*))) *e*))

lemma *bv-select-rev*:
assumes *notnull*: *n* < *length w*
shows *bv-select w n* = *rev w* ! *n*
⟨*proof*⟩

lemma *bv-chop-append*: *bv-chop* (*w1* @ *w2*) (*length w2*) = (*w1*,*w2*)
⟨*proof*⟩

lemma *append-bv-chop-id*: *fst* (*bv-chop w l*) @ *snd* (*bv-chop w l*) = *w*
⟨*proof*⟩

lemma *bv-chop-length-fst* [*simp*]: *length* (*fst* (*bv-chop w i*)) = *length w* − *i*
⟨*proof*⟩

lemma *bv-chop-length-snd* [*simp*]: *length* (*snd* (*bv-chop w i*)) = *min i* (*length w*)
⟨*proof*⟩

lemma *bv-slice-length* [*simp*]: [| *j* ≤ *i*; *i* < *length w* |] ==> *length* (*bv-slice w* (*i*,*j*)) = *i* − *j* + *1*
⟨*proof*⟩

definition
length-nat :: *nat* => *nat* **where**
[*code del*]: *length-nat x* = (*LEAST n*. *x* < *2* ^ *n*)

lemma *length-nat*: *length* (*nat-to-bv n*) = *length-nat n*
⟨*proof*⟩

lemma *length-nat-0* [*simp*]: *length-nat 0* = *0*
⟨*proof*⟩

lemma *length-nat-non0*:
assumes *n0*: *n* ≠ *0*
shows *length-nat n* = *Suc* (*length-nat* (*n div 2*))
⟨*proof*⟩

definition
length-int :: *int* => *nat* **where**
length-int x =
(*if 0* < *x then Suc* (*length-nat* (*nat x*))
else if x = *0 then 0*
else Suc (*length-nat* (*nat* (−*x* − *1*))))

lemma *length-int*: *length* (*int-to-bv i*) = *length-int i*
⟨*proof*⟩

lemma *length-int-0* [*simp*]: *length-int 0* = *0*
⟨*proof*⟩

lemma *length-int-gt0*: *0* < *i* ==> *length-int i* = *Suc* (*length-nat* (*nat i*))
⟨*proof*⟩

lemma *length-int-lt0*: *i* < *0* ==> *length-int i* = *Suc* (*length-nat* (*nat* (− *i*) − *1*))
⟨*proof*⟩

lemma *bv-chopI*: [| *w* = *w1* @ *w2* ; *i* = *length w2* |] ==> *bv-chop w i* = (*w1*,*w2*)
⟨*proof*⟩

lemma *bv-sliceI*: [| *j* ≤ *i* ; *i* < *length w* ; *w* = *w1* @ *w2* @ *w3* ; *Suc i* = *length w2* + *j* ; *j* = *length w3* |] ==> *bv-slice w* (*i*,*j*) = *w2*
⟨*proof*⟩

lemma *bv-slice-bv-slice*:
assumes *ki*: *k* ≤ *i*
and *ij*: *i* ≤ *j*
and *jl*: *j* ≤ *l*
and *lw*: *l* < *length w*
shows *bv-slice w* (*j*,*i*) = *bv-slice* (*bv-slice w* (*l*,*k*)) (*j*−*k*,*i*−*k*)
⟨*proof*⟩

lemma *bv-to-nat-extend* [*simp*]: *bv-to-nat* (*bv-extend n* **0** *w*) = *bv-to-nat w*
⟨*proof*⟩

lemma *bv-msb-extend-same* [*simp*]: *bv-msb w* = *b* ==> *bv-msb* (*bv-extend n b w*) = *b*
⟨*proof*⟩

lemma *bv-to-int-extend* [*simp*]:
assumes *a*: *bv-msb w* = *b*
shows *bv-to-int* (*bv-extend n b w*) = *bv-to-int w*
⟨*proof*⟩

lemma *length-nat-mono* [*simp*]: *x* ≤ *y* ==> *length-nat x* ≤ *length-nat y*
⟨*proof*⟩

lemma *length-nat-mono-int* [*simp*]: *x* ≤ *y* ==> *length-nat x* ≤ *length-nat y*
⟨*proof*⟩

lemma *length-nat-pos* [*simp*,*intro*!]: *0* < *x* ==> *0* < *length-nat x*
⟨*proof*⟩

lemma *length-int-mono-gt0*: [| *0* ≤ *x* ; *x* ≤ *y* |] ==> *length-int x* ≤ *length-int y*

⟨*proof*⟩

lemma *length-int-mono-lt0*: [| $x \leq y$; $y \leq 0$ |] ==> *length-int* $y \leq$ *length-int* x
⟨*proof*⟩

lemmas [*simp*] = *length-nat-non0*

lemma *nat-to-bv* (*number-of Int.Pls*) = []
⟨*proof*⟩

consts
fast-bv-to-nat-helper :: [*bit list*, *int*] => *int*
primrec
fast-bv-to-nat-Nil: *fast-bv-to-nat-helper* [] *k* = *k*
fast-bv-to-nat-Cons: *fast-bv-to-nat-helper* (*b*#*bs*) *k* =
fast-bv-to-nat-helper bs ((*bit-case Int.Bit0 Int.Bit1 b*) *k*)

declare *fast-bv-to-nat-helper.simps* [*code del*]

lemma *fast-bv-to-nat-Cons0*: *fast-bv-to-nat-helper* (**0**#*bs*) *bin* =
fast-bv-to-nat-helper bs (*Int.Bit0 bin*)
⟨*proof*⟩

lemma *fast-bv-to-nat-Cons1*: *fast-bv-to-nat-helper* (**1**#*bs*) *bin* =
fast-bv-to-nat-helper bs (*Int.Bit1 bin*)
⟨*proof*⟩

lemma *fast-bv-to-nat-def*:
bv-to-nat bs == *number-of* (*fast-bv-to-nat-helper bs Int.Pls*)
⟨*proof*⟩

declare *fast-bv-to-nat-Cons* [*simp del*]
declare *fast-bv-to-nat-Cons0* [*simp*]
declare *fast-bv-to-nat-Cons1* [*simp*]

⟨*ML*⟩

declare *bv-to-nat1* [*simp del*]
declare *bv-to-nat-helper* [*simp del*]

definition
bv-mapzip :: [*bit* => *bit* => *bit*,*bit list*, *bit list*] => *bit list* **where**
bv-mapzip f w1 w2 =
(*let g* = *bv-extend* (*max* (*length w1*) (*length w2*)) **0**
in map (*split f*) (*zip* (*g w1*) (*g w2*)))

lemma *bv-length-bv-mapzip* [*simp*]:
length (*bv-mapzip f w1 w2*) = *max* (*length w1*) (*length w2*)
⟨*proof*⟩

lemma *bv-mapzip-Nil* [*simp*]: *bv-mapzip f* [] [] = []
⟨*proof*⟩

lemma *bv-mapzip-Cons* [*simp*]: *length w1* = *length w2* ==>
bv-mapzip f (*x*#*w1*) (*y*#*w2*) = *f x y* # *bv-mapzip f w1 w2*
⟨*proof*⟩

end

68 Order-Relation: Orders as Relations

theory *Order-Relation*
imports *Main*
begin

68.1 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*⟩

68.2 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*⟩

68.3 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

69 Zorn: Zorn's Lemma

theory *Zorn*
imports *Order-Relation Main*
begin

definition *chain-subset* :: *'a set set* ⇒ *bool* (*chain*$_{\subseteq}$) **where**
chain$_{\subseteq}$ *C* ≡ ∀ *A*∈*C*.∀ *B*∈*C*. *A* ⊆ *B* ∨ *B* ⊆ *A*

The lemma and section numbers refer to an unpublished article [1].

definition
chain :: *'a set set* => *'a set set set* **where**
chain S = {*F*. *F* ⊆ *S* & *chain*$_{\subseteq}$ *F*}

definition
super :: ['*a set set*,'*a set set*] => *'a set set set* **where**
super S c = {*d*. *d* ∈ *chain S* & *c* ⊂ *d*}

definition
maxchain :: *'a set set* => *'a set set set* **where**
maxchain S = {*c*. *c* ∈ *chain S* & *super S c* = {}}

definition
succ :: ['*a set set*,'*a set set*] => *'a set set* **where**
succ S c =
(*if c* ∉ *chain S* | *c* ∈ *maxchain S*
then c else SOME c'. *c'* ∈ *super S c*)

inductive-set
TFin :: *'a set set* => *'a set set set*
for *S* :: *'a set set*
where
succI: *x* ∈ *TFin S* ==> *succ S x* ∈ *TFin S*
| *Pow-UnionI*: *Y* ∈ *Pow*(*TFin S*) ==> *Union*(*Y*) ∈ *TFin S*

69.1 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*: $x \subseteq succ\ S\ x$
⟨*proof*⟩

lemmas *TFin-UnionI* = *TFin.Pow-UnionI* [*OF PowI*]

lemma *TFin-induct*:
assumes *H*: $n \in TFin\ S$
and *I*: $!!x.\ x \in TFin\ S ==> P\ x ==> P\ (succ\ S\ x)$
$!!Y.\ Y \subseteq TFin\ S ==> Ball\ Y\ P ==> P(Union\ Y)$
shows $P\ n$ ⟨*proof*⟩

lemma *succ-trans*: $x \subseteq y ==> x \subseteq succ\ S\ y$
⟨*proof*⟩

Lemma 1 of section 3.1

lemma *TFin-linear-lemma1*:
$[|\ n \in TFin\ S;\ m \in TFin\ S;$
$\forall x \in TFin\ S.\ x \subseteq m --> x = m \mid succ\ S\ x \subseteq m$
$|] ==> n \subseteq m \mid succ\ S\ m \subseteq n$
⟨*proof*⟩

Lemma 2 of section 3.2

lemma *TFin-linear-lemma2*:
$m \in TFin\ S ==> \forall n \in TFin\ S.\ n \subseteq m --> n=m \mid succ\ S\ n \subseteq m$
⟨*proof*⟩

Re-ordering the premises of Lemma 2

lemma *TFin-subsetD*:
$[|\ n \subseteq m;\ m \in TFin\ S;\ n \in TFin\ S\ |] ==> n=m \mid succ\ S\ n \subseteq m$
⟨*proof*⟩

Consequences from section 3.3 – Property 3.2, the ordering is total

lemma *TFin-subset-linear*: $[|\ m \in TFin\ S;\ n \in TFin\ S|] ==> n \subseteq m \mid m \subseteq n$
⟨*proof*⟩

Lemma 3 of section 3.3

lemma *eq-succ-upper*: $[|\ n \in TFin\ S;\ m \in TFin\ S;\ m = succ\ S\ m\ |] ==> n \subseteq m$
⟨*proof*⟩

Property 3.3 of section 3.3

lemma *equal-succ-Union*: $m \in TFin\ S ==> (m = succ\ S\ m) = (m = Union(TFin\ S))$
⟨*proof*⟩

69.2 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*:
$c \in chain\ S - maxchain\ S \Longrightarrow (\epsilon\, c'.\ c' \text{: } super\ S\ c) \text{: } super\ S\ c$
⟨*proof*⟩

lemma *select-not-equals*:
$c \in chain\ S - maxchain\ S \Longrightarrow (\epsilon\, c'.\ c' \text{: } super\ S\ c) \neq c$
⟨*proof*⟩

lemma *succI3*: $c \in chain\ S - maxchain\ S \Longrightarrow succ\ S\ c = (\epsilon\, c'.\ c' \text{: } super\ S\ c)$
⟨*proof*⟩

lemma *succ-not-equals*: $c \in chain\ S - maxchain\ S \Longrightarrow succ\ S\ c \neq c$
⟨*proof*⟩

lemma *TFin-chain-lemma4*: $c \in TFin\ S \Longrightarrow (c :: 'a\ set\ set) \text{: } chain\ S$
⟨*proof*⟩

theorem *Hausdorff*: $\exists\, c.\ (c :: 'a\ set\ set) \text{: } maxchain\ S$
⟨*proof*⟩

69.3 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*:
$\forall c \in chain\ S.\ Union(c)\colon S ==> \exists y \in S.\ \forall z \in S.\ y \subseteq z --> y = z$
⟨*proof*⟩

69.4 Alternative version of Zorn's Lemma

lemma *Zorn-Lemma2*:
$\forall c \in chain\ S.\ \exists y \in S.\ \forall x \in c.\ x \subseteq y$
$==> \exists y \in S.\ \forall x \in S.\ (y :: 'a\ set) \subseteq x --> y = x$
⟨*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** $u\colon \forall C{\in}Chain\ r.\ \exists u{\in}Field\ r.\ \forall a{\in}C.\ (a,u){:}r$
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-seg-of* $\Longrightarrow r{\in}R \Longrightarrow r$ *initial-segment-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-seg-of* **and** $\forall\, r{\in}R.\ wf\ r$ **shows** $wf(\bigcup R)$
⟨*proof*⟩

lemma *initial-segment-of-Diff*:
p *initial-segment-of* $q \Longrightarrow p - s$ *initial-segment-of* $q - s$
⟨*proof*⟩

lemma *Chain-inits-DiffI*:
$R \in$ *Chain init-seg-of* $\Longrightarrow \{r - s \mid r.\ r \in R\} \in$ *Chain init-seg-of*
⟨*proof*⟩

theorem *well-ordering*: $\exists\, r{::}('a{*}'a)set.$ *Well-order* $r \wedge$ *Field* $r =$ *UNIV*
⟨*proof*⟩

corollary *well-order-on*: $\exists\, r{::}('a{*}'a)set.$ *well-order-on* $A\ r$
⟨*proof*⟩

end

70 List-Prefix: List prefixes and postfixes

theory *List-Prefix*
imports *List Main*
begin

70.1 Prefix order on lists

instantiation *list* :: (*type*) *order*
begin

definition
prefix-def [*code del*]: $xs \le ys = (\exists zs.\ ys = xs\ @\ zs)$

definition
strict-prefix-def [*code del*]: $xs < ys = (xs \le ys \wedge xs \neq (ys::'a\ list))$

instance
⟨*proof*⟩

end

lemma *prefixI* [*intro?*]: $ys = xs\ @\ zs \Longrightarrow xs \le ys$
⟨*proof*⟩

lemma *prefixE* [*elim?*]:
assumes $xs \le ys$
obtains *zs* **where** $ys = xs\ @\ zs$
⟨*proof*⟩

lemma *strict-prefixI'* [*intro?*]: $ys = xs\ @\ z\ \#\ zs \Longrightarrow xs < ys$
⟨*proof*⟩

lemma *strict-prefixE'* [*elim?*]:
assumes $xs < ys$
obtains *z zs* **where** $ys = xs\ @\ z\ \#\ zs$
⟨*proof*⟩

lemma *strict-prefixI* [*intro?*]: $xs \le ys \Longrightarrow xs \neq ys \Longrightarrow xs < (ys::'a\ list)$
⟨*proof*⟩

lemma *strict-prefixE* [*elim?*]:
fixes *xs ys* :: *'a list*
assumes $xs < ys$
obtains $xs \le ys$ **and** $xs \neq ys$
⟨*proof*⟩

70.2 Basic properties of prefixes

theorem *Nil-prefix* [*iff*]: $[] \le xs$
⟨*proof*⟩

theorem *prefix-Nil* [*simp*]: $(xs \le []) = (xs = [])$
⟨*proof*⟩

lemma *prefix-snoc* [*simp*]: $(xs \le ys\ @\ [y]) = (xs = ys\ @\ [y] \vee xs \le ys)$
⟨*proof*⟩

lemma *Cons-prefix-Cons* [*simp*]: $(x\ \#\ xs \le y\ \#\ ys) = (x = y \wedge xs \le ys)$

⟨*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*⟩

70.3 Parallel lists

definition
parallel :: *'a list => 'a list => bool* (**infixl** ∥ *50*) **where**
$(xs \parallel ys) = (\neg\ xs \leq ys \wedge \neg\ ys \leq xs)$

lemma *parallelI* [*intro*]: $\neg\ xs \leq ys ==> \neg\ ys \leq xs ==> xs \parallel ys$
⟨*proof*⟩

lemma *parallelE* [*elim*]:
assumes $xs \parallel ys$
obtains $\neg\ xs \leq ys \wedge \neg\ ys \leq xs$
⟨*proof*⟩

theorem *prefix-cases*:
obtains $xs \leq ys \mid ys < xs \mid xs \parallel ys$
⟨*proof*⟩

theorem *parallel-decomp*:
$xs \parallel ys ==> \exists\ as\ b\ bs\ c\ cs.\ b \neq c \wedge xs = as\ @\ b\ \#\ bs \wedge ys = as\ @\ c\ \#\ cs$
⟨*proof*⟩

lemma *parallel-append*: $a \parallel b \Longrightarrow a\ @\ c \parallel b\ @\ d$
⟨*proof*⟩

lemma *parallel-appendI*: $xs \parallel ys \Longrightarrow x = xs\ @\ xs' \Longrightarrow y = ys\ @\ ys' \Longrightarrow x \parallel y$
⟨*proof*⟩

lemma *parallel-commute*: $a \parallel b \longleftrightarrow b \parallel a$
⟨*proof*⟩

70.4 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*: $xs >>= ys \Longrightarrow xs = take\ (length\ xs - length\ ys)\ xs\ @\ ys$
⟨*proof*⟩

lemma *parallelD1*: $x \parallel y \Longrightarrow \neg\ x \leq y$
⟨*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*⟩

70.5 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

71 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*]: $x \leq [] \longleftrightarrow x = []$
⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
⟨*proof*⟩

lemma *Cons-le-Cons* [*simp*]: $a \# x \leq b \# y \longleftrightarrow a < b \vee a = b \wedge x \leq y$
⟨*proof*⟩

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

72 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*.

72.1 Definitions and basic lemmas

instantiation *list* :: (*type*) *order*
begin

inductive *less-eq-list* **where**
empty [*simp*, *intro*!]: $[] \leq xs$
| *drop*: $ys \leq xs \Longrightarrow ys \leq x \# xs$
| *take*: $ys \leq xs \Longrightarrow x \# ys \leq x \# xs$

lemmas *ileq-empty* = *empty*
lemmas *ileq-drop* = *drop*
lemmas *ileq-take* = *take*

lemma *ileq-cases* [*cases set*, *case-names empty drop take*]:
assumes $xs \leq ys$

and $xs = [] \Longrightarrow P$
and $\bigwedge z\ zs.\ ys = z \# zs \Longrightarrow xs \leq zs \Longrightarrow P$
and $\bigwedge x\ zs\ ws.\ xs = x \# zs \Longrightarrow ys = x \# ws \Longrightarrow zs \leq ws \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *ileq-induct* [*induct set*, *case-names empty drop take*]:
assumes $xs \leq ys$
and $\bigwedge zs.\ P\ []\ zs$
and $\bigwedge z\ zs\ ws.\ ws \leq zs \Longrightarrow\ P\ ws\ zs \Longrightarrow P\ ws\ (z \# zs)$
and $\bigwedge z\ zs\ ws.\ ws \leq zs \Longrightarrow P\ ws\ zs \Longrightarrow P\ (z \# ws)\ (z \# zs)$
shows $P\ xs\ ys$
⟨*proof*⟩

definition
[*code del*]: $(xs :: 'a\ list) < ys \longleftrightarrow xs \leq ys \wedge \neg\ ys \leq xs$

lemma *ileq-length*: $xs \leq ys \Longrightarrow length\ xs \leq length\ ys$
⟨*proof*⟩
lemma *ileq-below-empty* [*simp*]: $xs \leq [] \longleftrightarrow xs = []$
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemmas *ileq-intros* = *ileq-empty ileq-drop ileq-take*

lemma *ileq-drop-many*: $xs \leq ys \Longrightarrow xs \leq zs\ @\ ys$
⟨*proof*⟩
lemma *ileq-take-many*: $xs \leq ys \Longrightarrow zs\ @\ xs \leq zs\ @\ ys$
⟨*proof*⟩

lemma *ileq-same-length*: $xs \leq ys \Longrightarrow length\ xs = length\ ys \Longrightarrow xs = ys$
⟨*proof*⟩
lemma *ileq-same-append* [*simp*]: $x \# xs \leq xs \longleftrightarrow False$
⟨*proof*⟩

lemma *ilt-length* [*intro*]:
assumes $xs < ys$
shows $length\ xs < length\ ys$
⟨*proof*⟩

lemma *ilt-empty* [*simp*]: $[] < xs \longleftrightarrow xs \neq []$
⟨*proof*⟩
lemma *ilt-emptyI*: $xs \neq [] \Longrightarrow [] < xs$
⟨*proof*⟩
lemma *ilt-emptyD*: $[] < xs \Longrightarrow xs \neq []$
⟨*proof*⟩

lemma *ilt-below-empty*[*simp*]: $xs < [] \Longrightarrow False$
⟨*proof*⟩
lemma *ilt-drop*: $xs < ys \Longrightarrow xs < x \# ys$
⟨*proof*⟩
lemma *ilt-take*: $xs < ys \Longrightarrow x \# xs < x \# ys$
⟨*proof*⟩
lemma *ilt-drop-many*: $xs < ys \Longrightarrow xs < zs @ ys$
⟨*proof*⟩
lemma *ilt-take-many*: $xs < ys \Longrightarrow zs @ xs < zs @ ys$
⟨*proof*⟩

72.2 Appending elements

lemma *ileq-rev-take*: $xs \leq ys \Longrightarrow xs @ [x] \leq ys @ [x]$
⟨*proof*⟩
lemma *ilt-rev-take*: $xs < ys \Longrightarrow xs @ [x] < ys @ [x]$
⟨*proof*⟩
lemma *ileq-rev-drop*: $xs \leq ys \Longrightarrow xs \leq ys @ [x]$
⟨*proof*⟩
lemma *ileq-rev-drop-many*: $xs \leq ys \Longrightarrow xs \leq ys @ zs$
⟨*proof*⟩

72.3 Relation to standard list operations

lemma *ileq-map*: $xs \leq ys \Longrightarrow map\ f\ xs \leq map\ f\ ys$
⟨*proof*⟩
lemma *ileq-filter-left*[*simp*]: $filter\ f\ xs \leq xs$
⟨*proof*⟩
lemma *ileq-filter*: $xs \leq ys \Longrightarrow filter\ f\ xs \leq filter\ f\ ys$
⟨*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. Oberschelp. *Rekursionstheorie*. BI-Wissenschafts-Verlag, 1993.

[4] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.