The Supplemental Isabelle/HOL Library

June 8, 2008

Contents

1 GCD: The Greatest Common Divisor

theory *GCD*
imports *ATP-Linkup*
begin

See [3].

1.1 Specification of GCD on nats

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

Uniqueness

lemma *is-gcd-unique*: *is-gcd m a b* ⟹ *is-gcd n a b* ⟹ *m* = *n*
⟨*proof*⟩

Connection to divides relation

lemma *is-gcd-dvd*: *is-gcd m a b* ⟹ *k dvd a* ⟹ *k dvd b* ⟹ *k dvd m*
⟨*proof*⟩

Commutativity

lemma *is-gcd-commute*: *is-gcd k m n* = *is-gcd k n m*
⟨*proof*⟩

1.2 GCD on nat by Euclid's algorithm

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

lemma *gcd-induct*:
fixes *m n* :: *nat*
assumes ⋀*m*. *P m 0*
and ⋀*m n*. *0* < *n* ⟹ *P n* (*m mod n*) ⟹ *P m n*
shows *P m n*
⟨*proof*⟩

lemma *gcd-0* [*simp*]: *gcd* (*m*, *0*) = *m*
⟨*proof*⟩

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

lemma *gcd-non-0*: *n* > *0* ⟹ *gcd* (*m*, *n*) = *gcd* (*n*, *m mod n*)
⟨*proof*⟩

lemma *gcd-1* [*simp*]: *gcd* (*m*, *Suc 0*) = *1*
⟨*proof*⟩

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

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

lemma *gcd-dvd1* [*iff*]: *gcd* (*m*, *n*) *dvd m*
and *gcd-dvd2* [*iff*]: *gcd* (*m*, *n*) *dvd n*
⟨*proof*⟩

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

lemma *gcd-greatest*: *k dvd m* ⟹ *k dvd n* ⟹ *k dvd gcd* (*m*, *n*)
⟨*proof*⟩

Function gcd yields the Greatest Common Divisor.

lemma *is-gcd*: *is-gcd* (*gcd* (*m*, *n*)) *m n*
⟨*proof*⟩

1.3 Derived laws for GCD

lemma *gcd-greatest-iff* [*iff*]: *k dvd gcd* (*m*, *n*) ⟷ *k dvd m* ∧ *k dvd n*
⟨*proof*⟩

lemma *gcd-zero*: *gcd* (*m*, *n*) = *0* ⟷ *m* = *0* ∧ *n* = *0*
⟨*proof*⟩

lemma *gcd-commute*: *gcd* (*m*, *n*) = *gcd* (*n*, *m*)
⟨*proof*⟩

lemma *gcd-assoc*: *gcd* (*gcd* (*k*, *m*), *n*) = *gcd* (*k*, *gcd* (*m*, *n*))
⟨*proof*⟩

lemma *gcd-1-left* [*simp*]: *gcd* (*Suc 0*, *m*) = *1*
⟨*proof*⟩

Multiplication laws

lemma *gcd-mult-distrib2*: *k* ∗ *gcd* (*m*, *n*) = *gcd* (*k* ∗ *m*, *k* ∗ *n*)
— [3, page 27]
⟨*proof*⟩

lemma *gcd-mult* [*simp*]: *gcd* (*k*, *k* ∗ *n*) = *k*
⟨*proof*⟩

lemma *gcd-self* [*simp*]: *gcd* (*k*, *k*) = *k*

⟨*proof*⟩

lemma *relprime-dvd-mult*: *gcd* (*k*, *n*) = *1* ==> *k dvd m* * *n* ==> *k dvd m*
⟨*proof*⟩

lemma *relprime-dvd-mult-iff*: *gcd* (*k*, *n*) = *1* ==> (*k dvd m* * *n*) = (*k dvd m*)
⟨*proof*⟩

lemma *gcd-mult-cancel*: *gcd* (*k*, *n*) = *1* ==> *gcd* (*k* * *m*, *n*) = *gcd* (*m*, *n*)
⟨*proof*⟩

Addition laws

lemma *gcd-add1* [*simp*]: *gcd* (*m* + *n*, *n*) = *gcd* (*m*, *n*)
⟨*proof*⟩

lemma *gcd-add2* [*simp*]: *gcd* (*m*, *m* + *n*) = *gcd* (*m*, *n*)
⟨*proof*⟩

lemma *gcd-add2'* [*simp*]: *gcd* (*m*, *n* + *m*) = *gcd* (*m*, *n*)
⟨*proof*⟩

lemma *gcd-add-mult*: *gcd* (*m*, *k* * *m* + *n*) = *gcd* (*m*, *n*)
⟨*proof*⟩

lemma *gcd-dvd-prod*: *gcd* (*m*, *n*) *dvd m* * *n*
⟨*proof*⟩

Division by gcd yields rrelatively primes.

lemma *div-gcd-relprime*:
assumes *nz*: *a* ≠ *0* ∨ *b* ≠ *0*
shows *gcd* (*a div gcd*(*a*,*b*), *b div gcd*(*a*,*b*)) = *1*
⟨*proof*⟩

1.4 LCM defined by GCD

definition
lcm :: *nat* × *nat* ⇒ *nat*
where
lcm-prim-def: *lcm* = (λ(*m*, *n*). *m* * *n div gcd* (*m*, *n*))

lemma *lcm-def*:
lcm (*m*, *n*) = *m* * *n div gcd* (*m*, *n*)
⟨*proof*⟩

lemma *prod-gcd-lcm*:
m * *n* = *gcd* (*m*, *n*) * *lcm* (*m*, *n*)
⟨*proof*⟩

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

⟨*proof*⟩

lemma *lcm-1* [*simp*]: *lcm* (*m*, *1*) = *m*
⟨*proof*⟩

lemma *lcm-0-left* [*simp*]: *lcm* (*0*, *n*) = *0*
⟨*proof*⟩

lemma *lcm-1-left* [*simp*]: *lcm* (*1*, *m*) = *m*
⟨*proof*⟩

lemma *dvd-pos*:
fixes *n m* :: *nat*
assumes *n* > *0* **and** *m dvd n*
shows *m* > *0*
⟨*proof*⟩

lemma *lcm-least*:
assumes *m dvd k* **and** *n dvd k*
shows *lcm* (*m*, *n*) *dvd k*
⟨*proof*⟩

lemma *lcm-dvd1* [*iff*]:
m dvd lcm (*m*, *n*)
⟨*proof*⟩

lemma *lcm-dvd2* [*iff*]:
n dvd lcm (*m*, *n*)
⟨*proof*⟩

1.5 GCD and LCM on integers

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

lemma *igcd-dvd1* [*simp*]: *igcd i j dvd i*
⟨*proof*⟩

lemma *igcd-dvd2* [*simp*]: *igcd i j dvd j*
⟨*proof*⟩

lemma *igcd-pos*: *igcd i j* ≥ *0*
⟨*proof*⟩

lemma *igcd0* [*simp*]: (*igcd i j* = *0*) = (*i* = *0* ∧ *j* = *0*)
⟨*proof*⟩

lemma *igcd-commute*: *igcd i j* = *igcd j i*

⟨*proof*⟩

lemma *igcd-neg1* [*simp*]: *igcd* (− *i*) *j* = *igcd i j*
⟨*proof*⟩

lemma *igcd-neg2* [*simp*]: *igcd i* (− *j*) = *igcd i j*
⟨*proof*⟩

lemma *zrelprime-dvd-mult*: *igcd i j* = *1* ⟹ *i dvd k* ∗ *j* ⟹ *i dvd k*
⟨*proof*⟩

lemma *int-nat-abs*: *int* (*nat* (*abs x*)) = *abs x* ⟨*proof*⟩

lemma *igcd-greatest*:
assumes *k dvd m* **and** *k dvd n*
shows *k dvd igcd m n*
⟨*proof*⟩

lemma *div-igcd-relprime*:
assumes *nz*: *a* ≠ *0* ∨ *b* ≠ *0*
shows *igcd* (*a div* (*igcd a b*)) (*b div* (*igcd a b*)) = *1*
⟨*proof*⟩

definition *ilcm* = (λ*i j*. *int* (*lcm*(*nat*(*abs i*),*nat*(*abs j*))))

lemma *dvd-ilcm-self1*[*simp*]: *i dvd ilcm i j*
⟨*proof*⟩

lemma *dvd-ilcm-self2*[*simp*]: *j dvd ilcm i j*
⟨*proof*⟩

lemma *dvd-imp-dvd-ilcm1*:
assumes *k dvd i* **shows** *k dvd* (*ilcm i j*)
⟨*proof*⟩

lemma *dvd-imp-dvd-ilcm2*:
assumes *k dvd j* **shows** *k dvd* (*ilcm i j*)
⟨*proof*⟩

lemma *zdvd-self-abs1*: (*d*::*int*) *dvd* (*abs d*)
⟨*proof*⟩

lemma *zdvd-self-abs2*: (*abs* (*d*::*int*)) *dvd d*
⟨*proof*⟩

lemma *lcm-pos*:
  assumes *mpos*: $m > 0$
  and *npos*: $n>0$
  shows $lcm\ (m,n) > 0$
⟨*proof*⟩

lemma *ilcm-pos*:
  assumes *anz*: $a \neq 0$
  and *bnz*: $b \neq 0$
  shows $0 < ilcm\ a\ b$
⟨*proof*⟩

end

2 Abstract-Rat: Abstract rational numbers

theory *Abstract-Rat*
imports *GCD*
begin

types $Num = int \times int$

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

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

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

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

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

Arithmetic over Num

definition
  Nadd :: $Num \Rightarrow Num \Rightarrow Num$ (**infixl** $+_N$ *60*)

where
Nadd = (λ(*a*,*b*) (*a*′,*b*′). *if a* = *0* ∨ *b* = *0 then normNum*(*a*′,*b*′)
else if a′=*0* ∨ *b*′ = *0 then normNum*(*a*,*b*)
else normNum(*a*∗*b*′ + *b*∗*a*′, *b*∗*b*′))

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

definition
Nneg :: *Num* ⇒ *Num* (~$_N$)
where
Nneg ≡ (λ(*a*,*b*). (−*a*,*b*))

definition
Nsub :: *Num* ⇒ *Num* ⇒ *Num* (**infixl** −$_N$ *60*)
where
Nsub = (λ*a b*. *a* +$_N$ ~$_N$ *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** ÷$_N$ *60*)
where
Ndiv ≡ λ*a b*. *a* ∗$_N$ *Ninv b*

lemma *Nneg-normN*[*simp*]: *isnormNum x* ⟹ *isnormNum* (~$_N$ *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* (~$_N$ *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* ÷$_N$ *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*}) ≤ *0*) = *0*≥$_N$ *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*}) ≥ *0*) = *0*≤$_N$ *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*})≤ *INum y*) = (*x* ≤$_N$ *y*)
⟨*proof*⟩

lemma *Nadd-commute*: *x* +$_N$ *y* = *y* +$_N$ *x*
⟨*proof*⟩

lemma[*simp*]: (*0*, *b*) +$_N$ *y* = *normNum y* (*a*, *0*) +$_N$ *y* = *normNum y*
x +$_N$ (*0*, *b*) = *normNum x x* +$_N$ (*a*, *0*) = *normNum x*
⟨*proof*⟩

lemma *normNum-nilpotent-aux*[*simp*]: **assumes** *nx*: *isnormNum x*
shows *normNum x* = *x*
⟨*proof*⟩

lemma *normNum-nilpotent*[*simp*]: *normNum* (*normNum x*) = *normNum x*
⟨*proof*⟩
lemma *normNum0*[*simp*]: *normNum* (*0*,*b*) = *0*$_N$ *normNum* (*a*,*0*) = *0*$_N$
⟨*proof*⟩
lemma *normNum-Nadd*: *normNum* (*x* +$_N$ *y*) = *x* +$_N$ *y* ⟨*proof*⟩
lemma *Nadd-normNum1*[*simp*]: *normNum x* +$_N$ *y* = *x* +$_N$ *y*
⟨*proof*⟩
lemma *Nadd-normNum2*[*simp*]: *x* +$_N$ *normNum y* = *x* +$_N$ *y*
⟨*proof*⟩

lemma *Nadd-assoc*: *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** *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** *x*: *isnormNum x* **and** *y*:*isnormNum y* **shows** (*x* −$_N$ *y* = *0*$_N$) = (*x* = *y*)

⟨*proof*⟩

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

lemma *Nmul-eq0*[*simp*]: **assumes** *nx*:*isnormNum x* **and** *ny*: *isnormNum y*
shows $(x *_N y = 0_N) = (x = 0_N \vee y = 0_N)$
⟨*proof*⟩
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

3 AssocList: Map operations implemented on association lists

theory *AssocList*
imports *Map*
begin

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

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

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

primrec
$updates :: 'key\ list \Rightarrow 'val\ list \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
$updates\ []\ vs\ ps = ps$
$| \ updates\ (k\#ks)\ vs\ ps = (case\ vs$
$of\ [] \Rightarrow ps$
$| \ (v\#vs') \Rightarrow updates\ ks\ vs'\ (update\ k\ v\ ps))$

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*

3.1 *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*) $k' = ((map\text{-}of\ al)(k := None))\ k'$
⟨*proof*⟩

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

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

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

lemma *delete-notin-dom*: $k \notin fst\ `\ set\ (delete\ k\ al)$
⟨*proof*⟩

lemma *dom-delete-subset*: $fst\ `\ set\ (delete\ k\ al) \subseteq fst\ `\ set\ al$
⟨*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*⟩

3.2 *clearjunk*

lemma *insert-fst-filter*:
insert a$(fst\ `\ \{x \in set\ ps.\ fst\ x \neq a\}) = insert\ a\ (fst\ `\ set\ ps)$
⟨*proof*⟩

lemma *dom-clearjunk*: $fst\ `\ set\ (clearjunk\ al) = fst\ `\ set\ al$
⟨*proof*⟩

lemma *notin-filter-fst*: $a \notin fst\ `\ \{x \in set\ ps.\ fst\ x \neq a\}$
⟨*proof*⟩

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

lemma *map-of-filter*: $k \neq a \Longrightarrow map\text{-}of\ [q \leftarrow ps\ .\ fst\ q \neq a]\ k = map\text{-}of\ ps\ k$
⟨*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*⟩

3.3 *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*⟩

3.4 *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*:

$a{\neq}k \Longrightarrow update\ k\ v\ [q{\leftarrow}ps\ .\ fst\ q \neq a] = [q{\leftarrow}update\ k\ v\ ps\ .\ fst\ q \neq a]$
⟨*proof*⟩

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

lemma *update-triv*: $map\text{-}of\ al\ k = Some\ v \Longrightarrow update\ k\ v\ al = al$
⟨*proof*⟩

lemma *update-nonempty* [*simp*]: $update\ k\ v\ al \neq []$
⟨*proof*⟩

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

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

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

3.5 *updates*

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

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

3.6 *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*⟩

3.7 *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*⟩

3.8 *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*⟩

3.9 *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* ∈ *A* ⟹ *map-of* (*restrict A al*) *x* = *map-of al x*
⟨*proof*⟩

lemma *restr-out* [*simp*]: *x* ∉ *A* ⟹ *map-of* (*restrict A al*) *x* = *None*
⟨*proof*⟩

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

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

al
⟨*proof*⟩

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

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

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

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

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

lemma *restr-updates* [*simp*]:
⟦ *length xs* = *length ys*; *set xs* ⊆ *D* ⟧
⟹ *map-of* (*restrict D* (*updates xs ys al*)) =
map-of (*updates xs ys* (*restrict* (*D* − *set xs*) *al*))
⟨*proof*⟩

lemma *restr-delete-twist*: (*restrict A* (*delete a ps*)) = *delete a* (*restrict A ps*)
⟨*proof*⟩

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

end

4 SetsAndFunctions: Operations on sets and functions

theory *SetsAndFunctions*
imports *ATP-Linkup*
begin

This library lifts operations like addition and muliplication to sets and

functions of appropriate types. It was designed to support asymptotic calculations. See the comments at the top of theory *BigO*.

4.1 Basic definitions

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

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

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

instance ⟨*proof*⟩

end

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

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

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

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

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

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

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

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

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

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

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

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

interpretation *set-comm-monoid-add*: *comm-monoid-add* [{*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*⟩

4.2 Basic properties

lemma *set-plus-intro* [*intro*]: $a : C \Longrightarrow b : D \Longrightarrow a + b : C \oplus D$
⟨*proof*⟩

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

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

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

lemma *set-plus-rearrange3*: $((a::'a::semigroup\text{-}add) +o B) \oplus C = a +o (B \oplus C)$
⟨*proof*⟩

theorem *set-plus-rearrange4*: $C \oplus ((a::'a::comm\text{-}monoid\text{-}add) +o D) = a +o (C \oplus D)$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *set-times-mono* [*intro*]: $C <= D ==> a *o C <= a *o D$
⟨*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-1*) : (− *1*) ∗*o* *C* ==>
− *a* : *C*
⟨*proof*⟩

lemma *set-neg-intro2*: (*a*::'*a*::*ring-1*) : *C* ==>
− *a* : (− *1*) ∗*o* *C*
⟨*proof*⟩

end

5 BigO: Big O notation

theory *BigO*
imports *Main SetsAndFunctions*
begin

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

The main changes in this version are as follows:

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

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

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

5.1 Definitions

definition

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

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

lemma *bigo-mult* [*intro*]: *O(f)⊗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 ~= 0 ==>*
*O(f * g) <= (f::'a => ('b::ordered-field)) *o O(g)*
⟨*proof*⟩

lemma *bigo-mult6*: *ALL x. f x ~= 0 ==>*
*O(f * g) = (f::'a => ('b::ordered-field)) *o O(g)*
⟨*proof*⟩

lemma *bigo-mult7*: *ALL x. f x ~= 0 ==>*
*O(f * g) <= O(f::'a => ('b::ordered-field)) ⊗ 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*: $(c::'a::ordered\text{-}field)\ \sim=\ 0 ==> (\%x.\ 1) : O(\%x.\ c)$
⟨*proof*⟩

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

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

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

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

5.2 Setsum

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

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

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

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

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

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

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

5.3 Misc useful stuff

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

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

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

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

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

5.4 Less than or equal to

definition

lesso :: ('*a* => '*b*::*ordered-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-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*⟩

end

6 Binomial: Binomial Coefficients

theory *Binomial*
imports *ATP-Linkup*
begin

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

consts
binomial :: *nat* ⇒ *nat* ⇒ *nat* (**infixl** *choose 65*)
primrec
binomial-0: (*0 choose k*) = (*if k = 0 then 1 else 0*)
binomial-Suc: (*Suc n choose k*) =
(*if k = 0 then 1 else* (*n choose* (*k − 1*)) + (*n choose k*))

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

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

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

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

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

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

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

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

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

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

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

lemma *Suc-times-binomial-eq*:
!!k. $k \leq n$ ==> *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*⟩

6.1 Theorems about *choose*

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

lemma *card-s-0-eq-empty*:
finite A ==> *card* $\{B.\ B \subseteq A$ & *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*⟩

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

lemma *constr-bij*:
[|*finite* A; $x \notin A$|] ==>
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*: $(a+b::nat)\hat{}n = (\sum k{=}0..n.\ (n$ *choose* $k) * a\hat{}k * b\hat{}(n-k))$

⟨*proof*⟩

end

7 Boolean-Algebra: Boolean Algebras

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

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

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

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

lemma *dual*: *boolean disj conj compl one zero*
⟨*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*: $\sim x \oplus y = \sim (x \oplus y)$
⟨*proof*⟩

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

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

lemma *xor-cancel-left* [*simp*]: $\sim x \oplus x = \mathbf{1}$
⟨*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 *ATP-Linkup*
begin

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

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

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

instance ⟨*proof*⟩

end

lemma [*code*, *code func del*]:
$(x1, y1) \leq (x2, y2) \longleftrightarrow x1 < x2 \vee x1 = x2 \wedge y1 \leq y2$
$(x1, y1) < (x2, y2) \longleftrightarrow x1 < x2 \vee x1 = x2 \wedge y1 < y2$
⟨*proof*⟩

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

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

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

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

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

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

instance
⟨*proof*⟩

end

end

9 Char-nat: Mapping between characters and natural numbers

theory *Char-nat*
imports *List*
begin

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

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

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

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

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

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

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

lemma *nibble-of-nat-of-nibble*: *nibble-of-nat* (*nat-of-nibble n*) = *n*
⟨*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 func*]:
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*
begin

instantiation *nibble* :: *linorder*
begin

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

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

instance ⟨*proof*⟩

end

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

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

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

instance ⟨*proof*⟩

end

instantiation *char* :: *linorder*
begin

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

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

instance
⟨*proof*⟩

end

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

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

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

instance ⟨*proof*⟩

end

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

end

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

theory *Code-Char*
imports *List*
begin

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

lemma [*code func*]:
size (*c*::*char*) = *0*
⟨*proof*⟩

lemma [*code func*]:
char-size (*c*::*char*) = *0*
⟨*proof*⟩

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

end

12 Code-Integer: Pretty integer literals for code generation

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

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

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

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

⟨*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 *op* = :: *int* ⇒ *int* ⇒ *bool*
(*SML* !((- : *IntInf.int*) = -))
(*OCaml Big′-int.eq′-big′-int*)
(*Haskell* **infixl** *4* ==)

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

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

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

end

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

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

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

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

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

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

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

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

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

lemma [*code func*]:
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 *ATP-Linkup*

begin

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

14.1 Datatype of indices

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

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

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

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

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

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

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

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

instantiation *index* :: *zero*
begin

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

instance ⟨*proof*⟩

end

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

lemma *index-induct*: *P 0* ⟹ (⋀*k*. *P k* ⟹ *P* (*Suc-index k*)) ⟹ *P k*
⟨*proof*⟩

lemma *Suc-not-Zero-index*: *Suc-index k* ≠ *0*
⟨*proof*⟩

lemma *Zero-not-Suc-index*: *0* ≠ *Suc-index k*
⟨*proof*⟩

lemma *Suc-Suc-index-eq*: *Suc-index k* = *Suc-index l* ⟷ *k* = *l*
⟨*proof*⟩

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

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

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

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

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

lemma [*code func*]:
k = *l* ⟷ *nat-of-index k* = *nat-of-index l*
⟨*proof*⟩

14.2 Indices as datatype of ints

instantiation *index* :: *number*
begin

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

instance ⟨*proof*⟩

end

lemma *nat-of-index-number* [*simp*]:
nat-of-index (*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

lemma *zero-index-code* [*code inline*, *code func*]:
(*0*::*index*) = *Numeral0*
⟨*proof*⟩
lemma [*code post*]: *Numeral0* = (*0*::*index*)
⟨*proof*⟩

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

lemma *one-index-code* [*code inline*, *code func*]:
(*1*::*index*) = *Numeral1*
⟨*proof*⟩
lemma [*code post*]: *Numeral1* = (*1*::*index*)
⟨*proof*⟩

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

lemma *plus-index-code* [*code func*]:
index-of-nat n + *index-of-nat m* = *index-of-nat* (*n* + *m*)
⟨*proof*⟩

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

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

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

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

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

lemma *div-index-code* [*code func*]:
index-of-nat n div index-of-nat m = index-of-nat (n div m)
⟨*proof*⟩

lemma *mod-index-code* [*code func*]:
index-of-nat n mod index-of-nat m = index-of-nat (n mod m)
⟨*proof*⟩

definition [*simp*, *code func del*]:
$n \leq m \longleftrightarrow$ *nat-of-index n* $\leq$ *nat-of-index m*

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

lemma *less-eq-index-code* [*code func*]:
index-of-nat n $\leq$ *index-of-nat m* $\longleftrightarrow$ $n \leq m$
⟨*proof*⟩

lemma *less-index-code* [*code func*]:
index-of-nat n $<$ *index-of-nat m* $\longleftrightarrow$ $n < m$
⟨*proof*⟩

instance ⟨*proof*⟩

end

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

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

lemma *index-not-eq-zero*: *i* $\neq$ *index-of-nat 0* $\longleftrightarrow$ *i* $\geq$ *1*
⟨*proof*⟩

definition
nat-of-index-aux :: *index* $\Rightarrow$ *nat* $\Rightarrow$ *nat*
where
nat-of-index-aux i n = nat-of-index i + n

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

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

14.4 ML interface

⟨*ML*⟩

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

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

lemma [*code func*]: *op −* = *minus-index-aux*
  ⟨*proof*⟩

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

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

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

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

14.6 Code serialization

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

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

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

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

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

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

end

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

theory *Code-Message*
imports *List*
begin

15.1 Datatype of messages

datatype *message-string* = *STR string*

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

lemma [*code func*]: *size* (*s*::*message-string*) = *0*
⟨*proof*⟩

lemma [*code func*]: *message-string-size* (*s*::*message-string*) = *0*
⟨*proof*⟩

15.2 ML interface

⟨*ML*⟩

15.3 Code serialization

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

⟨*ML*⟩

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

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

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

end

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

theory *Coinductive-List*
imports *List*
begin

16.1 List constructors over the datatype universe

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

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

lemma *CONS-mono*: $M \subseteq M' \Longrightarrow N \subseteq N' \Longrightarrow CONS\ M\ N \subseteq CONS\ M'\ N'$
⟨*proof*⟩

lemma *CONS-UN1*: $CONS\ M\ (\bigcup x.\ f\ x) = (\bigcup x.\ CONS\ M\ (f\ x))$
— 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*⟩

16.2 Corecursive lists

coinductive-set *LList* **for** *A*
where *NIL* [*intro*]: *NIL* ∈ *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* ⇒ (′*a* ⇒ (′*b Datatype.item* × ′*a*) *option*) ⇒
′*a* ⇒ ′*b Datatype.item*
primrec
LList-corec-aux 0 f x = {}
LList-corec-aux (*Suc k*) *f x* =
(*case f x of*
None ⇒ *NIL*
| *Some* (*z*, *w*) ⇒ *CONS z* (*LList-corec-aux k f w*))

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

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-corec a f =
(*case f a of None* ⇒ *NIL* | *Some* (*z*, *w*) ⇒ *CONS z* (*LList-corec w f*))
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

16.3 Abstract type definition

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

lemma *NIL-type*: *NIL ∈ llist*
⟨*proof*⟩

lemma *CONS-type*: *a ∈ range Datatype.Leaf ⟹*
M ∈ llist ⟹ CONS a M ∈ llist
⟨*proof*⟩

lemma *llistI*: *x ∈ LList (range Datatype.Leaf) ⟹ x ∈ llist*
⟨*proof*⟩

lemma *llistD*: *x ∈ llist ⟹ x ∈ 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))*

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
llist-case c d l =
List-case c (λx 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*]: *llist-case c d LNil* = *c*
⟨*proof*⟩

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

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

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

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

16.4 Equality as greatest fixed-point – the bisimulation principle

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

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

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

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

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

lemma *EqLList-diag-iff* [*iff*]: (*p* ∈ *EqLList* (*diag A*)) = (*p* ∈ *diag* (*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*) ∈ *r*
and *step*: ⋀*M N*. (*M*, *N*) ∈ *r* ⟹
M = *NIL* ∧ *N* = *NIL* ∨
(∃ *a b M′ N′*.
M = *CONS a M′* ∧ *N* = *CONS b N′* ∧ (*a*, *b*) ∈ *diag A* ∧
((*M′*, *N′*) ∈ *r* ∨ (*M′*, *N′*) ∈ *EqLList* (*diag 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* ∈ *LList A*
and *fun-NIL*: *g NIL* ∈ *LList A* *f NIL* = *g NIL*
and *fun-CONS*: ⋀*x l*. *x* ∈ *A* ⟹ *l* ∈ *LList A* ⟹
(*f* (*CONS x l*), *g* (*CONS x l*)) = (*NIL*, *NIL*) ∨
(∃ *M N a b*.
(*f* (*CONS x l*), *g* (*CONS x l*)) = (*CONS a M*, *CONS b N*) ∧
(*a*, *b*) ∈ *diag A* ∧
(*M*, *N*) ∈ {(*f u*, *g u*) | *u*. *u* ∈ *LList A*} ∪ *diag* (*LList A*))
(**is** ⋀*x l*. - ⟹ - ⟹ *?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*: ⋀*x*. *h x* =
(*case f x of None* ⇒ *NIL* | *Some* (*z*, *w*) ⇒ *CONS z* (*h w*))
shows *h x* = (λ*x*. *LList-corec x f*) *x*
⟨*proof*⟩

lemma *llist-equalityI*
[*consumes 1*, *case-names Eqllist*, *case-conclusion Eqllist EqLNil EqLCons*]:
assumes *r*: (*l1*, *l2*) ∈ *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*: $\bigwedge x\ l.$
$(f\ (LCons\ x\ l),\ g\ (LCons\ x\ l)) = (LNil,\ LNil) \vee$
$(\exists\, l1\ l2\ a\ b.$
$(f\ (LCons\ x\ l),\ g\ (LCons\ x\ l)) = (LCons\ a\ l1,\ LCons\ b\ l2) \wedge$
$a = b \wedge ((l1,\ l2) \in \{(f\ u,\ g\ u) \mid u.\ True\} \vee l1 = l2))$
(**is** $\bigwedge x\ l.$ *?fun-LCons x l*)
shows $f\ l = g\ l$
⟨*proof*⟩

16.5 Derived operations – both on the set and abstract type

16.5.1 *Lconst*

definition $Lconst\ M \equiv lfp\ (\lambda N.\ CONS\ M\ N)$

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

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

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

lemma *Lconst-eq-LList-corec*: $Lconst\ M = LList\text{-}corec\ M\ (\lambda x.\ Some\ (x,\ x))$
⟨*proof*⟩

lemma *gfp-Lconst-eq-LList-corec*:
$gfp\ (\lambda N.\ CONS\ M\ N) = LList\text{-}corec\ M\ (\lambda x.\ Some(x,\ x))$
⟨*proof*⟩

16.5.2 *Lmap* **and** *lmap*

definition
$Lmap\ f\ M = LList\text{-}corec\ M\ (List\text{-}case\ None\ (\lambda x\ M'.\ Some\ (f\ x,\ M')))$
definition

lmap f l = llist-corec l
(λz.
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 ∈ LList A*
and *f*: *⋀x. x ∈ A ⟹ f x ∈ B*
shows *Lmap f M ∈ LList B*
⟨*proof*⟩

lemma *Lmap-compose*:
assumes *M*: *M ∈ 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 ∈ LList A*
shows *Lmap (λx. 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 (λx. x) l = l*
⟨*proof*⟩

16.5.3 *Lappend*

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

lemma *Lappend-NIL-NIL* [*simp*]:

Lappend NIL NIL = NIL
and *Lappend-NIL-CONS* [*simp*]:
Lappend NIL (CONS N N′) = CONS N (Lappend NIL N′)
and *Lappend-CONS* [*simp*]:
Lappend (CONS M M′) N = CONS M (Lappend M′ N)
⟨*proof*⟩

lemma *Lappend-NIL* [*simp*]: *M ∈ LList A ⟹ Lappend NIL M = M*
⟨*proof*⟩

lemma *Lappend-NIL2*: *M ∈ LList A ⟹ Lappend M NIL = M*
⟨*proof*⟩

lemma *Lappend-type*:
assumes *M*: *M ∈ LList A* **and** *N*: *N ∈ LList A*
shows *Lappend M N ∈ 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*⟩

16.6 iterates

llist-fun-equalityI cannot be used here!

definition
iterates :: *('a ⇒ 'a) ⇒ 'a ⇒ 'a llist* **where**
iterates f a = llist-corec a (λx. Some (x, f x))

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

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

lemma *funpow-lmap*:
fixes *f* :: *'a ⇒ 'a*
shows *(lmap f ^ n) (LCons b l) = LCons ((f ^ n) b) ((lmap f ^ n) l)*
⟨*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

17 Parity: Even and Odd for int and nat

theory *Parity*
imports *ATP-Linkup*
begin

class *even-odd = type +*
fixes *even* :: *'a ⇒ bool*

abbreviation
odd :: *'a::even-odd ⇒ bool* **where**
odd x ≡ ¬ even x

instantiation *nat* **and** *int* :: *even-odd*
begin

definition
even-def [*presburger*]: *even x ⟷ (x::int) mod 2 = 0*

definition
even-nat-def [*presburger*]: *even x ⟷ even (int x)*

instance ⟨*proof*⟩

end

17.1 Even and odd are mutually exclusive

lemma *int-pos-lt-two-imp-zero-or-one*:
$0 <= x ==> (x::int) < 2 ==> x = 0 \mid x = 1$
⟨*proof*⟩

lemma *neq-one-mod-two* [*simp*, *presburger*]:
$((x::int)\ mod\ 2\ \tilde{}= 0) = (x\ mod\ 2 = 1)$ ⟨*proof*⟩

17.2 Behavior under integer arithmetic operations

lemma *even-times-anything*: $even\ (x::int) ==> even\ (x * y)$
⟨*proof*⟩

lemma *anything-times-even*: $even\ (y::int) ==> even\ (x * y)$
⟨*proof*⟩

lemma *odd-times-odd*: $odd\ (x::int) ==> odd\ y ==> odd\ (x * y)$
⟨*proof*⟩

lemma *even-product*[*presburger*]: $even((x::int) * y) = (even\ x \mid even\ y)$
⟨*proof*⟩

lemma *even-plus-even*: $even\ (x::int) ==> even\ y ==> even\ (x + y)$
⟨*proof*⟩

lemma *even-plus-odd*: $even\ (x::int) ==> odd\ y ==> odd\ (x + y)$
⟨*proof*⟩

lemma *odd-plus-even*: $odd\ (x::int) ==> even\ y ==> odd\ (x + y)$
⟨*proof*⟩

lemma *odd-plus-odd*: $odd\ (x::int) ==> odd\ y ==> even\ (x + y)$ ⟨*proof*⟩

lemma *even-sum*[*presburger*]: $even\ ((x::int) + y) = ((even\ x\ \&\ even\ y) \mid (odd\ x\ \&\ odd\ y))$
⟨*proof*⟩

lemma *even-neg*[*presburger*]: $even\ (-(x::int)) = even\ x$ ⟨*proof*⟩

lemma *even-difference*:
$even\ ((x::int) - y) = ((even\ x\ \&\ even\ y) \mid (odd\ x\ \&\ odd\ y))$ ⟨*proof*⟩

lemma *even-pow-gt-zero*:
$even\ (x::int) ==> 0 < n ==> even\ (x\hat{}n)$
⟨*proof*⟩

lemma *odd-pow-iff*[*presburger*]: $odd\ ((x::int)\ \hat{}\ n) \longleftrightarrow (n = 0 \vee odd\ x)$
⟨*proof*⟩

lemma *odd-pow*: *odd x ==> odd((x::int) ^n)* ⟨*proof*⟩

lemma *even-power*[*presburger*]: *even ((x::int) ^n) = (even x & 0 < n)* ⟨*proof*⟩

lemma *even-zero*[*presburger*]: *even (0::int)* ⟨*proof*⟩

lemma *odd-one*[*presburger*]: *odd (1::int)* ⟨*proof*⟩

lemmas *even-odd-simps* [*simp*] = *even-def*[*of number-of v,standard*] *even-zero odd-one even-product even-sum even-neg even-difference even-power*

17.3 Equivalent definitions

lemma *two-times-even-div-two*: *even (x::int) ==> 2 * (x div 2) = x* ⟨*proof*⟩

lemma *two-times-odd-div-two-plus-one*: *odd (x::int) ==> 2 * (x div 2) + 1 = x* ⟨*proof*⟩

lemma *even-equiv-def*: *even (x::int) = (EX y. x = 2 * y)* ⟨*proof*⟩

lemma *odd-equiv-def*: *odd (x::int) = (EX y. x = 2 * y + 1)* ⟨*proof*⟩

17.4 even and odd for nats

lemma *pos-int-even-equiv-nat-even*: *0 ≤ x ==> even x = even (nat x)* ⟨*proof*⟩

lemma *even-nat-product*[*presburger*]: *even((x::nat) * y) = (even x | even y)* ⟨*proof*⟩

lemma *even-nat-sum*[*presburger*]: *even ((x::nat) + y) = ((even x & even y) | (odd x & odd y))* ⟨*proof*⟩

lemma *even-nat-difference*[*presburger*]: *even ((x::nat) - y) = (x < y | (even x & even y) | (odd x & odd y))* ⟨*proof*⟩

lemma *even-nat-Suc*[*presburger*]: *even (Suc x) = odd x* ⟨*proof*⟩

lemma *even-nat-power*[*presburger*]: *even ((x::nat) ^y) = (even x & 0 < y)* ⟨*proof*⟩

lemma *even-nat-zero*[*presburger*]: *even (0::nat)* ⟨*proof*⟩

lemmas *even-odd-nat-simps* [*simp*] = *even-nat-def*[*of number-of v,standard*] *even-nat-zero even-nat-Suc even-nat-product even-nat-sum even-nat-power*

17.5 Equivalent definitions

lemma *nat-lt-two-imp-zero-or-one*: $(x::nat) < Suc\ (Suc\ 0) ==>$
$x = 0 \mid x = Suc\ 0$ ⟨*proof*⟩

lemma *even-nat-mod-two-eq-zero*: $even\ (x::nat) ==> x\ mod\ (Suc\ (Suc\ 0)) = 0$
⟨*proof*⟩

lemma *odd-nat-mod-two-eq-one*: $odd\ (x::nat) ==> x\ mod\ (Suc\ (Suc\ 0)) = Suc\ 0$
⟨*proof*⟩

lemma *even-nat-equiv-def*: $even\ (x::nat) = (x\ mod\ Suc\ (Suc\ 0) = 0)$
⟨*proof*⟩

lemma *odd-nat-equiv-def*: $odd\ (x::nat) = (x\ mod\ Suc\ (Suc\ 0) = Suc\ 0)$
⟨*proof*⟩

lemma *even-nat-div-two-times-two*: $even\ (x::nat) ==>$
$Suc\ (Suc\ 0) * (x\ div\ Suc\ (Suc\ 0)) = x$ ⟨*proof*⟩

lemma *odd-nat-div-two-times-two-plus-one*: $odd\ (x::nat) ==>$
$Suc(\ Suc\ (Suc\ 0) * (x\ div\ Suc\ (Suc\ 0))) = x$ ⟨*proof*⟩

lemma *even-nat-equiv-def2*: $even\ (x::nat) = (EX\ y.\ x = Suc\ (Suc\ 0) * y)$
⟨*proof*⟩

lemma *odd-nat-equiv-def2*: $odd\ (x::nat) = (EX\ y.\ x = Suc(Suc\ (Suc\ 0) * y))$
⟨*proof*⟩

17.6 Parity and powers

lemma *minus-one-even-odd-power*:
$(even\ x --> (-\ 1::'a::\{comm\text{-}ring\text{-}1,recpower\})\ \hat{}\ x = 1)\ \&$
$(odd\ x --> (-\ 1::'a)\ \hat{}\ x = -\ 1)$
⟨*proof*⟩

lemma *minus-one-even-power* [*simp*]:
$even\ x ==> (-\ 1::'a::\{comm\text{-}ring\text{-}1,recpower\})\ \hat{}\ x = 1$
⟨*proof*⟩

lemma *minus-one-odd-power* [*simp*]:
$odd\ x ==> (-\ 1::'a::\{comm\text{-}ring\text{-}1,recpower\})\ \hat{}\ x = -\ 1$
⟨*proof*⟩

lemma *neg-one-even-odd-power*:
$(even\ x --> (-1::'a::\{number\text{-}ring,recpower\})\ \hat{}\ x = 1)\ \&$
$(odd\ x --> (-1::'a)\ \hat{}\ x = -1)$
⟨*proof*⟩

lemma *neg-one-even-power* [*simp*]:

even x ==> (−1::'a::{number-ring,recpower}) ^x = 1
⟨*proof*⟩

lemma *neg-one-odd-power* [*simp*]:
odd x ==> (−1::'a::{number-ring,recpower}) ^x = −1
⟨*proof*⟩

lemma *neg-power-if*:
(−x::'a::{comm-ring-1,recpower}) ^ n =
(if even n then (x ^ n) else −(x ^ n))
⟨*proof*⟩

lemma *zero-le-even-power*: *even n ==>*
0 <= (x::'a::{recpower,ordered-ring-strict}) ^ n
⟨*proof*⟩

lemma *zero-le-odd-power*: *odd n ==>*
(0 <= (x::'a::{recpower,ordered-idom}) ^ n) = (0 <= x)
⟨*proof*⟩

lemma *zero-le-power-eq*[*presburger*]: *(0 <= (x::'a::{recpower,ordered-idom}) ^ n)*
=
(even n | (odd n & 0 <= x))
⟨*proof*⟩

lemma *zero-less-power-eq*[*presburger*]: *(0 < (x::'a::{recpower,ordered-idom}) ^ n)*
=
(n = 0 | (even n & x ~= 0) | (odd n & 0 < x))
⟨*proof*⟩

lemma *power-less-zero-eq*[*presburger*]: *((x::'a::{recpower,ordered-idom}) ^ n < 0)*
=
(odd n & x < 0)
⟨*proof*⟩

lemma *power-le-zero-eq*[*presburger*]: *((x::'a::{recpower,ordered-idom}) ^ n <= 0)*
=
(n ~= 0 & ((odd n & x <= 0) | (even n & x = 0)))
⟨*proof*⟩

lemma *power-even-abs*: *even n ==>*
(abs (x::'a::{recpower,ordered-idom})) ^n = x^n
⟨*proof*⟩

lemma *zero-less-power-nat-eq*[*presburger*]: *(0 < (x::nat) ^ n) = (n = 0 | 0 < x)*
⟨*proof*⟩

lemma *power-minus-even* [*simp*]: *even n ==>*
(− x) ^n = (x^n::'a::{recpower,comm-ring-1})

⟨*proof*⟩

lemma *power-minus-odd* [*simp*]: *odd n* ==>
(− *x*) ^ *n* = − (*x*^*n*::'*a*::{*recpower*,*comm-ring-1*})
⟨*proof*⟩

17.7 General Lemmas About Division

lemma *Suc-times-mod-eq*: *1*<*k* ==> *Suc* (*k* ∗ *m*) *mod k* = *1*
⟨*proof*⟩

declare *Suc-times-mod-eq* [*of number-of w*, *standard*, *simp*]

lemma [*simp*]: *n div k* ≤ (*Suc n*) *div k*
⟨*proof*⟩

lemma *Suc-n-div-2-gt-zero* [*simp*]: (*0*::*nat*) < *n* ==> *0* < (*n* + *1*) *div 2*
⟨*proof*⟩

lemma *div-2-gt-zero* [*simp*]: (*1*::*nat*) < *n* ==> *0* < *n div 2*
⟨*proof*⟩

lemma *mod-mult-self3* [*simp*]: (*k*∗*n* + *m*) *mod n* = *m mod* (*n*::*nat*)
⟨*proof*⟩

lemma *mod-mult-self4* [*simp*]: *Suc* (*k*∗*n* + *m*) *mod n* = *Suc m mod n*
⟨*proof*⟩

lemma *mod-Suc-eq-Suc-mod*: *Suc m mod n* = *Suc* (*m mod n*) *mod n*
⟨*proof*⟩

17.8 More Even/Odd Results

lemma *even-mult-two-ex*: *even*(*n*) = (∃ *m*::*nat*. *n* = *2*∗*m*)
⟨*proof*⟩

lemma *odd-Suc-mult-two-ex*: *odd*(*n*) = (∃ *m*. *n* = *Suc* (*2*∗*m*))
⟨*proof*⟩

lemma *even-add* [*simp*]: *even*(*m* + *n*::*nat*) = (*even m* = *even n*)
⟨*proof*⟩

lemma *odd-add* [*simp*]: *odd*(*m* + *n*::*nat*) = (*odd m* ≠ *odd n*)
⟨*proof*⟩

lemma *div-Suc*: *Suc a div c* = *a div c* + *Suc 0 div c* +
(*a mod c* + *Suc 0 mod c*) *div c*
⟨*proof*⟩

lemma *lemma-even-div2* [*simp*]: *even* (*n*::*nat*) ==> (*n* + *1*) *div 2* = *n div 2*

⟨*proof*⟩

lemma *lemma-not-even-div2* [*simp*]: ~*even n* ==> (*n* + *1*) *div 2* = *Suc* (*n div 2*)
⟨*proof*⟩

lemma *even-num-iff*: *0* < *n* ==> *even n* = (~ *even*(*n* − *1* :: *nat*))
⟨*proof*⟩

lemma *even-even-mod-4-iff*: *even* (*n*::*nat*) = *even* (*n mod 4*)
⟨*proof*⟩

lemma *lemma-odd-mod-4-div-2*: *n mod 4* = (*3*::*nat*) ==> *odd*((*n* − *1*) *div 2*)
⟨*proof*⟩

lemma *lemma-even-mod-4-div-2*: *n mod 4* = (*1*::*nat*) ==> *even* ((*n* − *1*) *div 2*)
⟨*proof*⟩

Simplify, when the exponent is a numeral

lemmas *power-0-left-number-of* = *power-0-left* [*of number-of w*, *standard*]
declare *power-0-left-number-of* [*simp*]

lemmas *zero-le-power-eq-number-of* [*simp*] =
zero-le-power-eq [*of* - *number-of w*, *standard*]

lemmas *zero-less-power-eq-number-of* [*simp*] =
zero-less-power-eq [*of* - *number-of w*, *standard*]

lemmas *power-le-zero-eq-number-of* [*simp*] =
power-le-zero-eq [*of* - *number-of w*, *standard*]

lemmas *power-less-zero-eq-number-of* [*simp*] =
power-less-zero-eq [*of* - *number-of w*, *standard*]

lemmas *zero-less-power-nat-eq-number-of* [*simp*] =
zero-less-power-nat-eq [*of* - *number-of w*, *standard*]

lemmas *power-eq-0-iff-number-of* [*simp*] = *power-eq-0-iff* [*of* - *number-of w*, *standard*]

lemmas *power-even-abs-number-of* [*simp*] = *power-even-abs* [*of number-of w* -, *standard*]

17.9 An Equivalence for *0* ≤ *a*^*n*

lemma *even-power-le-0-imp-0*:
a ^ (*2*∗*k*) ≤ (*0*::*'a*::{*ordered-idom*,*recpower*}) ==> *a*=*0*
⟨*proof*⟩

lemma *zero-le-power-iff*[*presburger*]:

$(0 \leq a\hat{}n) = (0 \leq (a::'a::\{ordered\text{-}idom,recpower\}) \mid even\ n)$
⟨*proof*⟩

17.10 Miscellaneous

lemma *odd-pos*: $odd\ (n::nat) \Longrightarrow 0 < n$
⟨*proof*⟩

lemma [*presburger*]:$(x + 1)\ div\ 2 = x\ div\ 2 \longleftrightarrow even\ (x::int)$ ⟨*proof*⟩
lemma [*presburger*]: $(x + 1)\ div\ 2 = x\ div\ 2 + 1 \longleftrightarrow odd\ (x::int)$ ⟨*proof*⟩
lemma *even-plus-one-div-two*: $even\ (x::int) ==> (x + 1)\ div\ 2 = x\ div\ 2$ ⟨*proof*⟩
lemma *odd-plus-one-div-two*: $odd\ (x::int) ==> (x + 1)\ div\ 2 = x\ div\ 2 + 1$
⟨*proof*⟩

lemma [*presburger*]: $(Suc\ x)\ div\ Suc\ (Suc\ 0) = x\ div\ Suc\ (Suc\ 0) \longleftrightarrow even\ x$
⟨*proof*⟩
lemma [*presburger*]: $(Suc\ x)\ div\ Suc\ (Suc\ 0) = x\ div\ Suc\ (Suc\ 0) \longleftrightarrow even\ x$
⟨*proof*⟩
lemma *even-nat-plus-one-div-two*: $even\ (x::nat) ==>$
$(Suc\ x)\ div\ Suc\ (Suc\ 0) = x\ div\ Suc\ (Suc\ 0)$ ⟨*proof*⟩

lemma *odd-nat-plus-one-div-two*: $odd\ (x::nat) ==>$
$(Suc\ x)\ div\ Suc\ (Suc\ 0) = Suc\ (x\ div\ Suc\ (Suc\ 0))$ ⟨*proof*⟩

end

18 Commutative-Ring: Proving equalities in commutative rings

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

Syntax of multivariate polynomials (pol) and polynomial expressions.

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

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

Interpretation functions for the shadow syntax.

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

fun
Ipolex :: *'a*::{*comm-ring*,*recpower*} *list* ⇒ *'a polex* ⇒ *'a*
where
Ipolex l (*Pol P*) = *Ipol l P*
| *Ipolex l* (*Add P Q*) = *Ipolex l P* + *Ipolex l Q*
| *Ipolex l* (*Sub P Q*) = *Ipolex l P* − *Ipolex l Q*
| *Ipolex l* (*Mul P Q*) = *Ipolex l P* ∗ *Ipolex l Q*
| *Ipolex l* (*Pow p n*) = *Ipolex l p* ^ *n*
| *Ipolex l* (*Neg P*) = − *Ipolex l P*

Create polynomial normalized polynomials given normalized inputs.

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

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

Defining the basic ring operations on normalized polynomials

function
add :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol* ⇒ *'a pol* (**infixl** ⊕ *65*)
where
Pc a ⊕ *Pc b* = *Pc* (*a* + *b*)
| *Pc c* ⊕ *Pinj i P* = *Pinj i* (*P* ⊕ *Pc c*)
| *Pinj i P* ⊕ *Pc c* = *Pinj i* (*P* ⊕ *Pc c*)
| *Pc c* ⊕ *PX P i Q* = *PX P i* (*Q* ⊕ *Pc c*)
| *PX P i Q* ⊕ *Pc c* = *PX P i* (*Q* ⊕ *Pc c*)
| *Pinj x P* ⊕ *Pinj y Q* =
(*if x* = *y then mkPinj x* (*P* ⊕ *Q*)
else (*if x* > *y then mkPinj y* (*Pinj* (*x* − *y*) *P* ⊕ *Q*)
else mkPinj x (*Pinj* (*y* − *x*) *Q* ⊕ *P*)))
| *Pinj x P* ⊕ *PX Q y R* =
(*if x* = *0 then P* ⊕ *PX Q y R*

```
      else (if x = 1 then PX Q y (R ⊕ P)
        else PX Q y (R ⊕ Pinj (x − 1) P)))
  | PX P x R ⊕ Pinj y Q =
     (if y = 0 then PX P x R ⊕ Q
      else (if y = 1 then PX P x (R ⊕ Q)
        else PX P x (R ⊕ Pinj (y − 1) Q)))
  | PX P1 x P2 ⊕ PX Q1 y Q2 =
     (if x = y then mkPX (P1 ⊕ Q1) x (P2 ⊕ Q2)
      else (if x > y then mkPX (PX P1 (x − y) (Pc 0) ⊕ Q1) y (P2 ⊕ Q2)
        else mkPX (PX Q1 (y−x) (Pc 0) ⊕ P1) x (P2 ⊕ Q2)))
⟨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

19 Continuity: Continuity and iterations (of set transformers)

theory *Continuity*
imports *ATP-Linkup*
begin

19.1 Continuity for complete lattices

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

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

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

19.2 Chains

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

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

19.3 Continuity

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

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

19.4 Iteration

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

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

20 Countable: Encoding (almost) everything into natural numbers

theory *Countable*
imports *Finite-Set List Hilbert-Choice*
begin

20.1 The class of countable types

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

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

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 *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*⟩

end

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

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

⟨*ML*⟩

context *linorder*
begin

lemma *less-not-permute*: ¬ (*x* < *y* ∧ *y* < *x*) ⟨*proof*⟩

lemma *gather-simps*:
shows

$(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge x < u \wedge P\ x) \longleftrightarrow (\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in (\mathit{insert}\ u\ U).\ x < y) \wedge P\ x)$
and $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge l < x \wedge P\ x) \longleftrightarrow (\exists x.\ (\forall y \in (\mathit{insert}\ l\ L).\ y < x) \wedge (\forall y \in U.\ x < y) \wedge P\ x)$
$(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge x < u) \longleftrightarrow (\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in (\mathit{insert}\ u\ U).\ x < y))$
and $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge l < x) \longleftrightarrow (\exists x.\ (\forall y \in (\mathit{insert}\ l\ L).\ y < x) \wedge (\forall y \in U.\ x < y))$ ⟨*proof*⟩

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

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

lemma *minf-lt*: $\exists z\ .\ \forall x.\ x < z \longrightarrow (x < t \longleftrightarrow \mathit{True})$ ⟨*proof*⟩
lemma *minf-gt*: $\exists z\ .\ \forall x.\ x < z \longrightarrow (t < x \longleftrightarrow \mathit{False})$
⟨*proof*⟩

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

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

lemma *pinf-gt*: $\exists z\ .\ \forall x.\ z < x \longrightarrow (t < x \longleftrightarrow \mathit{True})$ ⟨*proof*⟩
lemma *pinf-lt*: $\exists z\ .\ \forall x.\ z < x \longrightarrow (x < t \longleftrightarrow \mathit{False})$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

lemma *finite-set-intervals2*:
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 *dlo-qe-bnds*:
assumes *ne*: $L \neq \{\}$ **and** *neU*: $U \neq \{\}$ **and** *fL*: *finite L* **and** *fU*: *finite U*
shows $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y)) \equiv (\forall\ l \in L.\ \forall u \in U.\ l < u)$
⟨*proof*⟩

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

lemma *dlo-qe-nolb*:
assumes *ne*: $U \neq \{\}$ **and** *fU*: *finite U*

shows $(\exists x.\ (\forall y \in \{\}.\ y < x) \wedge (\forall y \in U.\ x < y)) \equiv True$
⟨*proof*⟩

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

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

lemma *axiom*: *dense-linear-order* $(op \leq)$ $(op <)$ ⟨*proof*⟩
lemma *atoms*:
includes *meta-term-syntax*
shows *TERM* $(less :: 'a \Rightarrow \text{-})$
and *TERM* $(less\text{-}eq :: 'a \Rightarrow \text{-})$
and *TERM* $(op = :: 'a \Rightarrow \text{-})$ ⟨*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*:
$(P\ \&\ (Q \mid R)) = ((P\&Q) \mid (P\&R))$
$((Q \mid R)\ \&\ P) = ((Q\&P) \mid (R\&P))$
⟨*proof*⟩

lemmas *weak-dnf-simps* = *simp-thms dnf*

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

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

lemmas *dnf-simps* = *weak-dnf-simps nnf-simps ex-distrib*

⟨*ML*⟩

23 Contructive dense linear orders yield QE for linear arithmetic over ordered Fields – see *Arith-Tools.thy*

Linear order without upper bounds

locale *linorder-stupid-syntax* = *linorder*
begin
notation

less-eq (*op* $\sqsubseteq$) **and**
less-eq ((-/ $\sqsubseteq$ -) [*51*, *51*] *50*) **and**
less (*op* $\sqsubset$) **and**
less ((-/ $\sqsubset$ -) [*51*, *51*] *50*)

end

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

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

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

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

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

end

Linear order without upper bounds

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

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

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

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

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

end

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

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

context *constr-dense-linear-order*
begin

lemma *rinf-U*:
assumes *fU*: *finite U*
and *lin-dense*: $\forall\, x\ l\ u.\ (\forall\ t.\ l \sqsubset t \wedge t \sqsubset u \longrightarrow t \notin U) \wedge l \sqsubset x \wedge x \sqsubset u \wedge P\ x \longrightarrow (\forall\ y.\ l \sqsubset y \wedge y \sqsubset u \longrightarrow P\ y\ )$
and *nmpiU*: $\forall\, x.\ \neg\ MP \wedge \neg PP \wedge P\ x \longrightarrow (\exists\ u{\in}\ U.\ \exists\ u' \in U.\ u \sqsubseteq x \wedge x \sqsubseteq u')$
and *nmi*: ¬ *MP* **and** *npi*: ¬ *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*:
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** - ≡ (- ∨ - ∨ *?F*) **is** *?E* ≡ *?D*)
⟨*proof*⟩

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

lemmas *nmi-thms* = *nmi-conj nmi-disj nmi-eq nmi-neq nmi-lt nmi-le nmi-gt nmi-ge*

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

lemma *ferrack-axiom*: *constr-dense-linear-order less-eq less between*
⟨*proof*⟩
lemma *atoms*:
includes *meta-term-syntax*
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-field*) < *0* ⟹ ((*c*∗*x* <= *0*) == (*x* >= *0*))
⟨*proof*⟩

lemma *pos-prod-le*:(*c*::*'a*::*ordered-field*) > *0* ⟹ ((*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*⟩

end

24 Efficient-Nat: Implementation of natural numbers by target-language integers

theory *Efficient-Nat*
imports *Code-Integer Code-Index*
begin

When generating code for functions on natural numbers, the canonical representation using *0* and *Suc* is unsuitable for computations involving large numbers. The efficiency of the generated code can be improved drastically by implementing natural numbers by target-language integers. To do this, just include this theory.

24.1 Basic arithmetic

Most standard arithmetic functions on natural numbers are implemented using their counterparts on the integers:

code-datatype *number-nat-inst.number-of-nat*

lemma *zero-nat-code* [*code*, *code unfold*]:
0 = (*Numeral0* :: *nat*)
⟨*proof*⟩
lemmas [*code post*] = *zero-nat-code* [*symmetric*]

lemma *one-nat-code* [*code*, *code unfold*]:
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 func del*]: *divmod-aux* = *divmod*

lemma [*code func*]:
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*]:
n = *m* ⟷ (*of-nat n* :: *int*) = *of-nat m*
⟨*proof*⟩

lemma *less-eq-nat-code* [*code*]:
n ≤ *m* ⟷ (*of-nat n* :: *int*) ≤ *of-nat m*
⟨*proof*⟩

lemma *less-nat-code* [*code*]:
$n < m \longleftrightarrow (\mathit{of\text{-}nat}\ n :: \mathit{int}) < \mathit{of\text{-}nat}\ m$
⟨*proof*⟩

24.2 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code func*, *code unfold*]:
$\mathit{nat\text{-}case} = (\lambda f\ g\ n.\ \mathit{if}\ n = 0\ \mathit{then}\ f\ \mathit{else}\ g\ (n - 1))$
⟨*proof*⟩

24.3 Preprocessors

In contrast to *Suc n*, the term $n + 1$ is no longer a constructor term. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a recursion equation or in the arguments of an inductive relation in an introduction rule) must be eliminated. This can be accomplished by applying the following transformation rules:

lemma *Suc-if-eq*: $(\bigwedge n.\ f\ (\mathit{Suc}\ n) = h\ n) \Longrightarrow f\ 0 = g \Longrightarrow$
$f\ n = (\mathit{if}\ n = 0\ \mathit{then}\ g\ \mathit{else}\ h\ (n - 1))$
⟨*proof*⟩

lemma *Suc-clause*: $(\bigwedge n.\ P\ n\ (\mathit{Suc}\ n)) \Longrightarrow n \neq 0 \Longrightarrow P\ (n - 1)\ n$
⟨*proof*⟩

The rules above are built into a preprocessor that is plugged into the code generator. Since the preprocessor for introduction rules does not know anything about modes, some of the modes that worked for the canonical representation of natural numbers may no longer work.

⟨*ML*⟩

24.4 Target language setup

For ML, we map *nat* to target language integers, where we assert that values are always non-negative.

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

types-code
nat (*int*)
attach (*term-of*) ⟪
val term-of-nat = HOLogic.mk-number HOLogic.natT;
⟫
attach (*test*) ⟪

```
fun gen-nat i =
  let val n = random-range 0 i
  in (n, fn () => term-of-nat n) end;
⟩⟩
```

For Haskell we define our own *nat* type. The reason is that we have to distinguish type class instances for *nat* and *int*.

```
code-include Haskell Nat ⟨⟨
newtype Nat = Nat Integer deriving (Show, Eq);

instance Num Nat where {
  fromInteger k = Nat (if k >= 0 then k else 0);
  Nat n + Nat m = Nat (n + m);
  Nat n - Nat m = fromInteger (n - m);
  Nat n * Nat m = Nat (n * m);
  abs n = n;
  signum - = 1;
  negate n = error negate Nat;
};

instance Ord Nat where {
  Nat n <= Nat m = n <= m;
  Nat n < Nat m = n < m;
};

instance Real Nat where {
  toRational (Nat n) = toRational n;
};

instance Enum Nat where {
  toEnum k = fromInteger (toEnum k);
  fromEnum (Nat n) = fromEnum n;
};

instance Integral Nat where {
  toInteger (Nat n) = n;
  divMod n m = quotRem n m;
  quotRem (Nat n) (Nat m) = (Nat k, Nat l) where (k, l) = quotRem n m;
};
⟩⟩

code-reserved Haskell Nat

code-type nat
  (Haskell Nat)

code-instance nat :: eq
  (Haskell -)
```

Natural numerals.

lemma [*code inline*, *symmetric*, *code post*]:
nat (*number-of i*) = *number-nat-inst.number-of-nat i*
— this interacts as desired with *number-of ?v* = *nat* (*number-of ?v*)
⟨*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 func del*]: *int* = *of-nat*

lemma *int-code'* [*code func*]:
int (*number-of l*) = (*if neg* (*number-of l* :: *int*) *then 0 else number-of l*)
⟨*proof*⟩

lemma *nat-code'* [*code func*]:
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 *index-of-nat*
(*SML IntInf.toInt*)
(*OCaml Big'-int.int'-of'-big'-int*)
(*Haskell toEnum*)

code-const *nat-of-index*
(*SML IntInf.fromInt*)
(*OCaml Big'-int.big'-int'-of'-int*)
(*Haskell fromEnum*)

Using target language arithmetic operations whenever appropriate

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

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

code-const *divmod-aux*
(*SML IntInf.divMod/* ((-),/ (-)))
(*OCaml Big'-int.quomod'-big'-int*)
(*Haskell divMod*)

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

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

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

consts-code
0 (*0*)
Suc ((- +/ *1*))
op + :: *nat* ⇒ *nat* ⇒ *nat* ((- +/ -))
op ∗ :: *nat* ⇒ *nat* ⇒ *nat* ((- ∗/ -))
op ≤ :: *nat* ⇒ *nat* ⇒ *bool* ((- <=/ -))
op < :: *nat* ⇒ *nat* ⇒ *bool* ((- </ -))

Module names

code-modulename *SML*
  Nat Integer
  Divides Integer
  Efficient-Nat Integer

code-modulename *OCaml*
  Nat Integer
  Divides Integer
  Efficient-Nat Integer

code-modulename *Haskell*
  Nat Integer
  Divides Integer
  Efficient-Nat Integer

hide *const int*

end

25 Enum: Finite types as explicit enumerations

theory *Enum*
imports *Main*
begin

25.1 Class *enum*

class *enum* = *itself* +
  fixes *enum* :: *'a list*
  assumes *UNIV-enum* [*code func*]: *UNIV* = *set enum*
    and *enum-distinct*: *distinct enum*
begin

lemma *finite-enum*: *finite* (*UNIV* :: *'a set*)
  ⟨*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

25.2 Equality and order on functions

instantiation *fun* :: (*enum*, *eq*) *eq*
begin

definition
eq-class.eq f g ⟷ (∀ *x* ∈ *set enum*. *f x* = *g x*)

instance ⟨*proof*⟩

end

lemma *order-fun* [*code func*]:
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*⟩

25.3 Quantifiers

lemma *all-code* [*code func*]: (∀ *x*. *P x*) ⟷ *list-all P enum*
⟨*proof*⟩

lemma *exists-code* [*code func*]: (∃ *x*. *P x*) ⟷ ¬ *list-all* (*Not o P*) *enum*
⟨*proof*⟩

25.4 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 $f :: 'a::enum \Rightarrow 'b::enum$
shows *map-of* (*zip xs* (*map f xs*)) = (λx. *if* $x \in$ *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 $\exists y$. *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 func del*]: *enum* = *map* (λys. *the o map-of* (*zip* (*enum*::$'a$ *list*) *ys*)) (*n-lists* (*length* (*enum*::$'a$::*enum list*)) *enum*)

instance ⟨*proof*⟩

end

lemma [*code func*]:
enum = *map* (λys. *the o map-of* (*zip* (*enum*::($'a$::{*enum*, *eq*}) *list*) *ys*)) (*n-lists* (*length* (*enum*::$'a$::{*enum*, *eq*} *list*)) *enum*)
⟨*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 func del*]: *enum* = *map* (*split Char*) (*product enum enum*)

lemma *enum-char* [*code func*]:
enum = [*Char Nibble0 Nibble0*, *Char Nibble0 Nibble1*, *Char Nibble0 Nibble2*,
Char Nibble0 Nibble3, *Char Nibble0 Nibble4*, *Char Nibble0 Nibble5*,
Char Nibble0 Nibble6, *Char Nibble0 Nibble7*, *Char Nibble0 Nibble8*,
Char Nibble0 Nibble9, *Char Nibble0 NibbleA*, *Char Nibble0 NibbleB*,
Char Nibble0 NibbleC, *Char Nibble0 NibbleD*, *Char Nibble0 NibbleE*,
Char Nibble0 NibbleF, *Char Nibble1 Nibble0*, *Char Nibble1 Nibble1*,
Char Nibble1 Nibble2, *Char Nibble1 Nibble3*, *Char Nibble1 Nibble4*,
Char Nibble1 Nibble5, *Char Nibble1 Nibble6*, *Char Nibble1 Nibble7*,
Char Nibble1 Nibble8, *Char Nibble1 Nibble9*, *Char Nibble1 NibbleA*,
Char Nibble1 NibbleB, *Char Nibble1 NibbleC*, *Char Nibble1 NibbleD*,
Char Nibble1 NibbleE, *Char Nibble1 NibbleF*, *CHR '' ''*, *CHR ''!''*,
Char Nibble2 Nibble2, *CHR ''#''*, *CHR ''$''*, *CHR ''%''*, *CHR ''&''*,
Char Nibble2 Nibble7, *CHR ''(''*, *CHR '')''*, *CHR ''*''*, *CHR ''+''*, *CHR '',''*,
CHR ''−'', *CHR ''.''*, *CHR ''/''*, *CHR ''0''*, *CHR ''1''*, *CHR ''2''*, *CHR ''3''*,
CHR ''4'', *CHR ''5''*, *CHR ''6''*, *CHR ''7''*, *CHR ''8''*, *CHR ''9''*, *CHR '':''*,
CHR '';'', *CHR ''<''*, *CHR ''=''*, *CHR ''>''*, *CHR ''?''*, *CHR ''@''*, *CHR ''A''*,
CHR ''B'', *CHR ''C''*, *CHR ''D''*, *CHR ''E''*, *CHR ''F''*, *CHR ''G''*, *CHR ''H''*,
CHR ''I'', *CHR ''J''*, *CHR ''K''*, *CHR ''L''*, *CHR ''M''*, *CHR ''N''*, *CHR ''O''*,
CHR ''P'', *CHR ''Q''*, *CHR ''R''*, *CHR ''S''*, *CHR ''T''*, *CHR ''U''*, *CHR ''V''*,
CHR ''W'', *CHR ''X''*, *CHR ''Y''*, *CHR ''Z''*, *CHR ''[''*, *Char Nibble5 NibbleC*,
CHR '']'', *CHR ''^''*, *CHR ''-''*, *Char Nibble6 Nibble0*, *CHR ''a''*, *CHR ''b''*,
CHR ''c'', *CHR ''d''*, *CHR ''e''*, *CHR ''f''*, *CHR ''g''*, *CHR ''h''*, *CHR ''i''*,
CHR ''j'', *CHR ''k''*, *CHR ''l''*, *CHR ''m''*, *CHR ''n''*, *CHR ''o''*, *CHR ''p''*,
CHR ''q'', *CHR ''r''*, *CHR ''s''*, *CHR ''t''*, *CHR ''u''*, *CHR ''v''*, *CHR ''w''*,

CHR ''x'', *CHR ''y''*, *CHR ''z''*, *CHR ''{''*, *CHR ''|''*, *CHR ''}''*, *CHR ''~''*,
Char Nibble7 NibbleF, *Char Nibble8 Nibble0*, *Char Nibble8 Nibble1*,
Char Nibble8 Nibble2, *Char Nibble8 Nibble3*, *Char Nibble8 Nibble4*,
Char Nibble8 Nibble5, *Char Nibble8 Nibble6*, *Char Nibble8 Nibble7*,
Char Nibble8 Nibble8, *Char Nibble8 Nibble9*, *Char Nibble8 NibbleA*,
Char Nibble8 NibbleB, *Char Nibble8 NibbleC*, *Char Nibble8 NibbleD*,
Char Nibble8 NibbleE, *Char Nibble8 NibbleF*, *Char Nibble9 Nibble0*,
Char Nibble9 Nibble1, *Char Nibble9 Nibble2*, *Char Nibble9 Nibble3*,
Char Nibble9 Nibble4, *Char Nibble9 Nibble5*, *Char Nibble9 Nibble6*,
Char Nibble9 Nibble7, *Char Nibble9 Nibble8*, *Char Nibble9 Nibble9*,
Char Nibble9 NibbleA, *Char Nibble9 NibbleB*, *Char Nibble9 NibbleC*,
Char Nibble9 NibbleD, *Char Nibble9 NibbleE*, *Char Nibble9 NibbleF*,
Char NibbleA Nibble0, *Char NibbleA Nibble1*, *Char NibbleA Nibble2*,
Char NibbleA Nibble3, *Char NibbleA Nibble4*, *Char NibbleA Nibble5*,
Char NibbleA Nibble6, *Char NibbleA Nibble7*, *Char NibbleA Nibble8*,
Char NibbleA Nibble9, *Char NibbleA NibbleA*, *Char NibbleA NibbleB*,
Char NibbleA NibbleC, *Char NibbleA NibbleD*, *Char NibbleA NibbleE*,
Char NibbleA NibbleF, *Char NibbleB Nibble0*, *Char NibbleB Nibble1*,
Char NibbleB Nibble2, *Char NibbleB Nibble3*, *Char NibbleB Nibble4*,
Char NibbleB Nibble5, *Char NibbleB Nibble6*, *Char NibbleB Nibble7*,
Char NibbleB Nibble8, *Char NibbleB Nibble9*, *Char NibbleB NibbleA*,
Char NibbleB NibbleB, *Char NibbleB NibbleC*, *Char NibbleB NibbleD*,
Char NibbleB NibbleE, *Char NibbleB NibbleF*, *Char NibbleC Nibble0*,
Char NibbleC Nibble1, *Char NibbleC Nibble2*, *Char NibbleC Nibble3*,
Char NibbleC Nibble4, *Char NibbleC Nibble5*, *Char NibbleC Nibble6*,
Char NibbleC Nibble7, *Char NibbleC Nibble8*, *Char NibbleC Nibble9*,
Char NibbleC NibbleA, *Char NibbleC NibbleB*, *Char NibbleC NibbleC*,
Char NibbleC NibbleD, *Char NibbleC NibbleE*, *Char NibbleC NibbleF*,
Char NibbleD Nibble0, *Char NibbleD Nibble1*, *Char NibbleD Nibble2*,
Char NibbleD Nibble3, *Char NibbleD Nibble4*, *Char NibbleD Nibble5*,
Char NibbleD Nibble6, *Char NibbleD Nibble7*, *Char NibbleD Nibble8*,
Char NibbleD Nibble9, *Char NibbleD NibbleA*, *Char NibbleD NibbleB*,
Char NibbleD NibbleC, *Char NibbleD NibbleD*, *Char NibbleD NibbleE*,
Char NibbleD NibbleF, *Char NibbleE Nibble0*, *Char NibbleE Nibble1*,
Char NibbleE Nibble2, *Char NibbleE Nibble3*, *Char NibbleE Nibble4*,
Char NibbleE Nibble5, *Char NibbleE Nibble6*, *Char NibbleE Nibble7*,
Char NibbleE Nibble8, *Char NibbleE Nibble9*, *Char NibbleE NibbleA*,
Char NibbleE NibbleB, *Char NibbleE NibbleC*, *Char NibbleE NibbleD*,
Char NibbleE NibbleE, *Char NibbleE NibbleF*, *Char NibbleF Nibble0*,
Char NibbleF Nibble1, *Char NibbleF Nibble2*, *Char NibbleF Nibble3*,
Char NibbleF Nibble4, *Char NibbleF Nibble5*, *Char NibbleF Nibble6*,
Char NibbleF Nibble7, *Char NibbleF Nibble8*, *Char NibbleF Nibble9*,
Char NibbleF NibbleA, *Char NibbleF NibbleB*, *Char NibbleF NibbleC*,
Char NibbleF NibbleD, *Char NibbleF NibbleE*, *Char NibbleF NibbleF*]
⟨*proof*⟩

instance ⟨*proof*⟩

end

end

26 RType: Reflecting Pure types into HOL

theory *RType*
imports *Main Code-Message Code-Index*
begin

datatype *rtype* = *RType message-string rtype list*

class *rtype* =
  fixes *rtype* :: *'a*::{} *itself* ⇒ *rtype*
begin

definition
  rtype-of :: *'a* ⇒ *rtype*
where
  [*simp*]: *rtype-of x* = *rtype TYPE*(*'a*)

end

⟨*ML*⟩

lemma [*code func*]:
  RType tyco1 tys1 = *RType tyco2 tys2* ⟷ *tyco1* = *tyco2*
    ∧ *list-all2* (*op* =) *tys1 tys2*
  ⟨*proof*⟩

code-type *rtype*
  (*SML Term.typ*)

code-const *RType*
  (*SML Term.Type*/ (-, -))

code-reserved *SML Term*

hide (**open**) *const rtype RType*

end

27 Eval: A simple term evaluation mechanism

theory *Eval*
imports
  RType
  Code-Index

begin

27.1 Term representation

27.1.1 Terms and class *term-of*

datatype *term* = *dummy-term*

definition
Const :: *message-string* ⇒ *rtype* ⇒ *term*
where
Const - - = *dummy-term*

definition
App :: *term* ⇒ *term* ⇒ *term*
where
App - - = *dummy-term*

code-datatype *Const App*

class *term-of* = *rtype* +
fixes *term-of* :: *'a* ⇒ *term*

lemma *term-of-anything*: *term-of x* ≡ *t*
⟨*proof*⟩

⟨*ML*⟩

27.1.2 *term-of* instances

⟨*ML*⟩

27.1.3 Code generator setup

lemmas [*code func del*] = *term.recs term.cases term.size*
lemma [*code func*, *code func del*]: (*t1*::*term*) = *t2* ⟷ *t1* = *t2* ⟨*proof*⟩

lemma [*code func*, *code func del*]: (*term-of* :: *rtype* ⇒ *term*) = *term-of* ⟨*proof*⟩
lemma [*code func*, *code func del*]: (*term-of* :: *term* ⇒ *term*) = *term-of* ⟨*proof*⟩
lemma [*code func*, *code func del*]: (*term-of* :: *index* ⇒ *term*) = *term-of* ⟨*proof*⟩
lemma [*code func*, *code func del*]: (*term-of* :: *message-string* ⇒ *term*) = *term-of* ⟨*proof*⟩

code-type *term*
(*SML Term.term*)

code-const *Const* **and** *App*
(*SML Term.Const*/ (-, -) **and** *Term.*$/ (-, -))

code-const *term-of* :: *index* ⇒ *term*

(*SML HOLogic.mk'-number*/ *HOLogic.indexT*)

code-const *term-of* :: *message-string* ⇒ *term*
(*SML Message'-String.mk*)

27.1.4 Syntax

⟨*ML*⟩

notation (**output**)
rterm-of (≪-≫)

locale (**open**) *rterm-syntax* =
fixes *rterm-of-syntax* :: *'a* ⇒ *'b* (≪-≫)

⟨*ML*⟩

hide *const dummy-term*
hide (**open**) *const Const App*
hide (**open**) *const term-of*

27.2 Evaluation setup

⟨*ML*⟩

end

28 Eval-Witness: Evaluation Oracle with ML witnesses

theory *Eval-Witness*
imports *List*
begin

We provide an oracle method similar to "eval", but with the possibility to provide ML values as witnesses for existential statements.

Our oracle can prove statements of the form $\exists\, x.\ P\ x$ where P is an executable predicate that can be compiled to ML. The oracle generates code for P and applies it to a user-specified ML value. If the evaluation returns true, this is effectively a proof of $\exists\, x.\ P\ x$.

However, this is only sound if for every ML value of the given type there exists a corresponding HOL value, which could be used in an explicit proof. Unfortunately this is not true for function types, since ML functions are not equivalent to the pure HOL functions. Thus, the oracle can only be used on first-order types.

We define a type class to mark types that can be safely used with the oracle.

class *ml-equiv* = *type*

Instances of *ml-equiv* should only be declared for those types, where the universe of ML values coincides with the HOL values.

Since this is essentially a statement about ML, there is no logical characterization.

instance *nat* :: *ml-equiv* ⟨*proof*⟩
instance *bool* :: *ml-equiv* ⟨*proof*⟩
instance *list* :: (*ml-equiv*) *ml-equiv* ⟨*proof*⟩

⟨*ML*⟩

28.1 Toy Examples

Note that we must use the generated data structure for the naturals, since ML integers are different.

Since polymorphism is not allowed, we must specify the type explicitly:

lemma ∃ *l*. *length* (*l*::*bool list*) = *3*
⟨*proof*⟩

Multiple witnesses

lemma ∃ *k l*. *length* (*k*::*bool list*) = *length* (*l*::*bool list*)
⟨*proof*⟩

28.2 Discussion

28.2.1 Conflicts

This theory conflicts with EfficientNat, since the *ml-equiv* instance for natural numbers is not valid when they are mapped to ML integers. With that theory loaded, we could use our oracle to prove ∃ *n*. *n* < (*0*::*'a*) by providing ~*1* as a witness.

This shows that *ml-equiv* declarations have to be used with care, taking the configuration of the code generator into account.

28.2.2 Haskell

If we were able to run generated Haskell code, the situation would be much nicer, since Haskell functions are pure and could be used as witnesses for HOL functions: Although Haskell functions are partial, we know that if the evaluation terminates, they are "sufficiently defined" and could be completed arbitrarily to a total (HOL) function.

This would allow us to provide access to very efficient data structures via lookup functions coded in Haskell and provided to HOL as witnesses.

end

29 Executable-Set: Implementation of finite sets by lists

theory *Executable-Set*
imports *List*
begin

29.1 Definitional rewrites

lemma [*code target*: *Set*]:
$A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
⟨*proof*⟩

declare *subset-eq* [*code*]

lemma [*code*]:
$a \in A \longleftrightarrow (\exists\, x{\in}A.\ x = a)$
⟨*proof*⟩

definition
filter-set :: $('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where**
filter-set $P\ xs = \{x{\in}xs.\ P\ x\}$

29.2 Operations on lists

29.2.1 Basic definitions

definition
flip :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$ **where**
flip $f\ a\ b = f\ b\ a$

definition
member :: $'a\ list \Rightarrow 'a \Rightarrow bool$ **where**
member $xs\ x \longleftrightarrow x \in set\ xs$

definition
insertl :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
insertl $x\ xs = (if\ member\ xs\ x\ then\ xs\ else\ x\#xs)$

lemma [*code target*: *List*]: *member* [] $y \longleftrightarrow False$
and [*code target*: *List*]: *member* $(x\#xs)\ y \longleftrightarrow y = x \vee member\ xs\ y$
⟨*proof*⟩

fun
drop-first :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
drop-first f [] = []
| *drop-first* $f\ (x\#xs) = (if\ f\ x\ then\ xs\ else\ x\ \#\ drop\text{-}first\ f\ xs)$
declare *drop-first.simps* [*code del*]
declare *drop-first.simps* [*code target*: *List*]

declare *remove1.simps* [*code del*]
lemma [*code target*: *List*]:
  remove1 x xs = (*if member xs x then drop-first* (λ*y*. *y* = *x*) *xs else xs*)
⟨*proof*⟩

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*⟩

29.2.2 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*)

29.3 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*⟩

29.4 code generator setup

⟨*ML*⟩

29.4.1 const serializations

consts-code
{} ({∗[]∗})
insert ({∗*insertl*∗})
op ∪ ({∗*unionl*∗})
op ∩ ({∗*intersect*∗})
op − :: *'a set* ⇒ *'a set* ⇒ *'a set* ({∗ *flip subtract* ∗})
image ({∗*map-distinct*∗})
Union ({∗*unions*∗})
Inter ({∗*intersects*∗})
UNION ({∗*map-union*∗})
INTER ({∗*map-inter*∗})
Ball ({∗*Blall*∗})
Bex ({∗*Blex*∗})
filter-set ({∗*filter*∗})

end

30 FuncSet: Pi and Function Sets

theory *FuncSet*
imports *Main*
begin

definition
Pi :: [*'a set*, *'a* => *'b set*] => (*'a* => *'b*) *set* **where**
Pi A B = {*f*. ∀ *x*. *x* ∈ *A* −−> *f x* ∈ *B x*}

definition
extensional :: *'a set => ('a => 'b) set* **where**
extensional A = {*f*. ∀*x*. *x*~:*A* --> *f x* = *arbitrary*}

definition
restrict :: [*'a => 'b*, *'a set*] => (*'a => 'b*) **where**
restrict f A = (%*x*. *if x* ∈ *A then f x else arbitrary*)

abbreviation
funcset :: [*'a set*, *'b set*] => (*'a => 'b*) *set*
(**infixr** --> *60*) **where**
A --> *B* == *Pi A* (%-. *B*)

notation (*xsymbols*)
funcset (**infixr** → *60*)

syntax
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a => 'b*) *set* ((*3PI* -:-./ -) *10*)
-lam :: [*pttrn*, *'a set*, *'a => 'b*] => (*'a=>'b*) ((*3*%-:-./ -) [*0*,*0*,*3*] *3*)

syntax (*xsymbols*)
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a => 'b*) *set* ((*3*Π -∈-./ -) *10*)
-lam :: [*pttrn*, *'a set*, *'a => 'b*] => (*'a=>'b*) ((*3*λ-∈-./ -) [*0*,*0*,*3*] *3*)

syntax (*HTML* **output**)
-Pi :: [*pttrn*, *'a set*, *'b set*] => (*'a => 'b*) *set* ((*3*Π -∈-./ -) *10*)
-lam :: [*pttrn*, *'a set*, *'a => 'b*] => (*'a=>'b*) ((*3*λ-∈-./ -) [*0*,*0*,*3*] *3*)

translations
PI x:*A*. *B* == *CONST Pi A* (%*x*. *B*)
%*x*:*A*. *f* == *CONST restrict* (%*x*. *f*) *A*

definition
compose :: [*'a set*, *'b => 'c*, *'a => 'b*] => (*'a => 'c*) **where**
compose A g f = (λ*x*∈*A*. *g* (*f x*))

30.1 Basic Properties of *Pi*

lemma *Pi-I*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B x*) ==> *f* ∈ *Pi A B*
⟨*proof*⟩

lemma *funcsetI*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B*) ==> *f* ∈ *A* --> *B*
⟨*proof*⟩

lemma *Pi-mem*: [|*f*: *Pi A B*; *x* ∈ *A*|] ==> *f x* ∈ *B x*
⟨*proof*⟩

lemma *funcset-mem*: [|*f* ∈ *A* --> *B*; *x* ∈ *A*|] ==> *f x* ∈ *B*

⟨*proof*⟩

lemma *funcset-image*: $f \in A \to B \Longrightarrow f \text{ ` } A \subseteq B$
⟨*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*]: $A \mathrel{->} UNIV = UNIV$
⟨*proof*⟩

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(!!x.\ x \in A \Longrightarrow B\ x <= C\ x) \Longrightarrow Pi\ A\ B <= Pi\ A\ C$
⟨*proof*⟩

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' <= A \Longrightarrow Pi\ A\ B <= Pi\ A'\ B$
⟨*proof*⟩

30.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*:
$[|\ f \in A \mathrel{->} B;\ g \in B \mathrel{->} C\ |] \Longrightarrow compose\ A\ g\ f \in A \mathrel{->} C$
⟨*proof*⟩

lemma *compose-assoc*:
$[|\ f \in A \mathrel{->} B;\ g \in B \mathrel{->} C;\ h \in C \mathrel{->} D\ |]$
$\Longrightarrow compose\ A\ h\ (compose\ A\ g\ f) = compose\ A\ (compose\ B\ h\ g)\ f$
⟨*proof*⟩

lemma *compose-eq*: $x \in A \Longrightarrow compose\ A\ g\ f\ x = g(f(x))$
⟨*proof*⟩

lemma *surj-compose*: $[|\ f \text{ ` } A = B;\ g \text{ ` } B = C\ |] \Longrightarrow compose\ A\ g\ f \text{ ` } A = C$
⟨*proof*⟩

30.3 Bounded Abstraction: *restrict*

lemma *restrict-in-funcset*: $(!!x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in A \mathrel{->} B$
⟨*proof*⟩

lemma *restrictI*: $(!!x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in Pi\ A\ B$
⟨*proof*⟩

lemma *restrict-apply* [*simp*]:
$(\lambda y{\in}A.\ f\ y)\ x = (if\ x \in A\ then\ f\ x\ else\ arbitrary)$

⟨*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-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*⟩

30.4 Bijections Between Sets

The definition of *bij-betw* is in *Fun.thy*, but most of the theorems belong here, or need at least *Hilbert-Choice*.

lemma *bij-betw-imp-funcset*: *bij-betw* $f\ A\ B \Longrightarrow f \in A \to B$
⟨*proof*⟩

lemma *inj-on-compose*:
[| *bij-betw* $f\ A\ B$; *inj-on* $g\ B$ |] ==> *inj-on* (*compose* $A\ g\ f$) A
⟨*proof*⟩

lemma *bij-betw-compose*:
[| *bij-betw* $f\ A\ B$; *bij-betw* $g\ B\ C$ |] ==> *bij-betw* (*compose* $A\ g\ f$) $A\ C$
⟨*proof*⟩

lemma *bij-betw-restrict-eq* [*simp*]:
bij-betw (*restrict* $f\ A$) $A\ B$ = *bij-betw* $f\ A\ B$
⟨*proof*⟩

30.5 Extensionality

lemma *extensional-arb*: [|$f \in$ *extensional* A; $x \notin A$|] ==> $f\ x$ = *arbitrary*
⟨*proof*⟩

lemma *restrict-extensional* [*simp*]: *restrict* $f\ A \in$ *extensional* A
⟨*proof*⟩

lemma *compose-extensional* [*simp*]: *compose* $A\ f\ g \in$ *extensional* A
⟨*proof*⟩

lemma *extensionalityI*:
[| $f \in$ *extensional* A; $g \in$ *extensional* A;
!!x. $x \in A$ ==> $f\ x = g\ x$ |] ==> $f = g$
⟨*proof*⟩

lemma *Inv-funcset*: f ' $A = B$ ==> $(\lambda x \in B.\ Inv\ A\ f\ x) : B \to A$
⟨*proof*⟩

lemma *compose-Inv-id*:
bij-betw $f\ A\ B$ ==> *compose* $A\ (\lambda y \in B.\ Inv\ A\ f\ y)\ f = (\lambda x \in A.\ x)$
⟨*proof*⟩

lemma *compose-id-Inv*:
f ' $A = B$ ==> *compose* $B\ f\ (\lambda y \in B.\ Inv\ A\ f\ y) = (\lambda x \in B.\ x)$
⟨*proof*⟩

30.6 Cardinality

lemma *card-inj*: [|$f \in A \to B$; *inj-on* $f\ A$; *finite* B|] ==> $card(A) \leq card(B)$
⟨*proof*⟩

lemma *card-bij*:
[|$f \in A \to B$; *inj-on* $f\ A$;
$g \in B \to A$; *inj-on* $g\ B$; *finite* A; *finite* B|] ==> $card(A) = card(B)$
⟨*proof*⟩

declare *FuncSet.Pi-I* [*skolem*]
declare *FuncSet.Pi-mono* [*skolem*]
declare *FuncSet.extensionalityI* [*skolem*]
declare *FuncSet.funcsetI* [*skolem*]
declare *FuncSet.restrictI* [*skolem*]
declare *FuncSet.restrict-in-funcset* [*skolem*]

end

31 Heap: A polymorphic heap based on cantor encodings

theory *Heap*
imports *Main Countable RType*
begin

31.1 Representable types

The type class of representable types

class *heap* = *rtype* + *countable*

Instances for common HOL types

instance *nat* :: *heap* ⟨*proof*⟩

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

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

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

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

instance *int* :: *heap* ⟨*proof*⟩

instance *message-string* :: *countable*
⟨*proof*⟩

instance *message-string* :: *heap* ⟨*proof*⟩

Reflected types themselves are heap-representable

instantiation *rtype* :: *countable*
begin

lemma *list-size-size-append*:
list-size size (*xs* @ *ys*) = *list-size size xs* + *list-size size ys*
⟨*proof*⟩

lemma *rtype-size*: *t* = *RType.RType c ts* ⟹ *t′* ∈ *set ts* ⟹ *size t′* < *size t*
⟨*proof*⟩

function *to-nat-rtype* :: *rtype* ⇒ *nat* **where**
to-nat-rtype (*RType.RType c ts*) = *to-nat* (*to-nat c*, *to-nat* (*map to-nat-rtype ts*))
⟨*proof*⟩

termination ⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *rtype* :: *heap* ⟨*proof*⟩

31.2 A polymorphic heap with dynamic arrays and references

types *addr* = *nat* — untyped heap references

datatype *'a array* = *Array addr*
datatype *'a ref* = *Ref addr* — note the phantom type 'a

primrec *addr-of-array* :: *'a array* ⇒ *addr* **where**
addr-of-array (*Array x*) = *x*

primrec *addr-of-ref* :: *'a ref* ⇒ *addr* **where**
addr-of-ref (*Ref x*) = *x*

lemma *addr-of-array-inj* [*simp*]:
addr-of-array a = *addr-of-array a'* ⟷ *a* = *a'*
⟨*proof*⟩

lemma *addr-of-ref-inj* [*simp*]:
addr-of-ref r = *addr-of-ref r'* ⟷ *r* = *r'*
⟨*proof*⟩

instance *array* :: (*type*) *countable*
⟨*proof*⟩

instance *ref* :: (*type*) *countable*
⟨*proof*⟩

⟨*ML*⟩

types *heap-rep* = *nat* — representable values

record *heap* =
arrays :: *rtype* ⇒ *addr* ⇒ *heap-rep list*
refs :: *rtype* ⇒ *addr* ⇒ *heap-rep*
lim :: *addr*

definition *empty* :: *heap* **where**
empty = (|*arrays* = (λ-. *arbitrary*), *refs* = (λ-. *arbitrary*), *lim* = *0*|) — why arbitrary?

31.3 Imperative references and arrays

References and arrays are developed in parallel, but keeping them separate makes some later proofs simpler.

31.3.1 Primitive operations

definition

new-ref :: *heap* ⇒ (*'a*::*heap*) *ref* × *heap* **where**
new-ref h = (*let l* = *lim h in* (*Ref l*, *h*(|*lim* := *l* + *1*|)))

definition
new-array :: *heap* ⇒ (*'a*::*heap*) *array* × *heap* **where**
new-array h = (*let l* = *lim h in* (*Array l*, *h*(|*lim* := *l* + *1*|)))

definition
ref-present :: *'a*::*heap ref* ⇒ *heap* ⇒ *bool* **where**
ref-present r h ⟷ *addr-of-ref r* < *lim h*

definition
array-present :: *'a*::*heap array* ⇒ *heap* ⇒ *bool* **where**
array-present a h ⟷ *addr-of-array a* < *lim h*

definition
get-ref :: *'a*::*heap ref* ⇒ *heap* ⇒ *'a* **where**
get-ref r h = *from-nat* (*refs h* (*RTYPE*(*'a*)) (*addr-of-ref r*))

definition
get-array :: *'a*::*heap array* ⇒ *heap* ⇒ *'a list* **where**
get-array a h = *map from-nat* (*arrays h* (*RTYPE*(*'a*)) (*addr-of-array a*))

definition
set-ref :: *'a*::*heap ref* ⇒ *'a* ⇒ *heap* ⇒ *heap* **where**
set-ref r x =
refs-update (λ*h*. *h*(*RTYPE*(*'a*) := ((*h* (*RTYPE*(*'a*))) (*addr-of-ref r*:=*to-nat x*))))

definition
set-array :: *'a*::*heap array* ⇒ *'a list* ⇒ *heap* ⇒ *heap* **where**
set-array a x =
arrays-update (λ*h*. *h*(*RTYPE*(*'a*) := ((*h* (*RTYPE*(*'a*))) (*addr-of-array a*:=*map to-nat x*))))

31.3.2 Interface operations

definition
ref :: *'a* ⇒ *heap* ⇒ *'a*::*heap ref* × *heap* **where**
ref x h = (*let* (*r*, *h'*) = *new-ref h*;
h'' = *set-ref r x h'*
in (*r*, *h''*))

definition
array :: *nat* ⇒ *'a* ⇒ *heap* ⇒ *'a*::*heap array* × *heap* **where**
array n x h = (*let* (*r*, *h'*) = *new-array h*;
h'' = *set-array r* (*replicate n x*) *h'*
in (*r*, *h''*))

definition

array-of-list :: *'a list* ⇒ *heap* ⇒ *'a*::*heap array* × *heap* **where**
array-of-list xs h = (*let* (*r*, *h'*) = *new-array h*;
h'' = *set-array r xs h'*
in (*r*, *h''*))

definition
upd :: *'a*::*heap array* ⇒ *nat* ⇒ *'a* ⇒ *heap* ⇒ *heap* **where**
upd a i x h = *set-array a* ((*get-array a h*)[*i*:=*x*]) *h*

definition
length :: *'a*::*heap array* ⇒ *heap* ⇒ *nat* **where**
length a h = *size* (*get-array a h*)

definition
array-ran :: (*'a*::*heap*) *option array* ⇒ *heap* ⇒ *'a set* **where**
array-ran a h = {*e*. *Some e* ∈ *set* (*get-array a h*)}
— FIXME

31.3.3 Reference equality

The following relations are useful for comparing arrays and references.

definition
noteq-refs :: (*'a*::*heap*) *ref* ⇒ (*'b*::*heap*) *ref* ⇒ *bool* (**infix** =!= *70*)
where
r =!= *s* ⟷ *RTYPE*(*'a*) ≠ *RTYPE*(*'b*) ∨ *addr-of-ref r* ≠ *addr-of-ref s*

definition
noteq-arrs :: (*'a*::*heap*) *array* ⇒ (*'b*::*heap*) *array* ⇒ *bool* (**infix** =!!= *70*)
where
r =!!= *s* ⟷ *RTYPE*(*'a*) ≠ *RTYPE*(*'b*) ∨ *addr-of-array r* ≠ *addr-of-array s*

lemma *noteq-refs-sym*: *r* =!= *s* ⟹ *s* =!= *r*
and *noteq-arrs-sym*: *a* =!!= *b* ⟹ *b* =!!= *a*
and *unequal-refs* [*simp*]: *r* ≠ *r'* ⟷ *r* =!= *r'* — same types!
and *unequal-arrs* [*simp*]: *a* ≠ *a'* ⟷ *a* =!!= *a'*
⟨*proof*⟩

lemma *present-new-ref*: *ref-present r h* ⟹ *r* =!= *fst* (*ref v h*)
⟨*proof*⟩

lemma *present-new-arr*: *array-present a h* ⟹ *a* =!!= *fst* (*array v x h*)
⟨*proof*⟩

31.3.4 Properties of heap containers

Properties of imperative arrays

FIXME: Does there exist a "canonical" array axiomatisation in the literature?

lemma *array-get-set-eq* [*simp*]: *get-array r* (*set-array r x h*) = *x*
⟨*proof*⟩

lemma *array-get-set-neq* [*simp*]: *r* =!!= *s* ⟹ *get-array r* (*set-array s x h*) = *get-array r h*
⟨*proof*⟩

lemma *set-array-same* [*simp*]:
set-array r x (*set-array r y h*) = *set-array r x h*
⟨*proof*⟩

lemma *array-set-set-swap*:
r =!!= *r'* ⟹ *set-array r x* (*set-array r' x' h*) = *set-array r' x'* (*set-array r x h*)
⟨*proof*⟩

lemma *array-ref-set-set-swap*:
set-array r x (*set-ref r' x' h*) = *set-ref r' x'* (*set-array r x h*)
⟨*proof*⟩

lemma *get-array-upd-eq* [*simp*]:
get-array a (*upd a i v h*) = (*get-array a h*) [*i* := *v*]
⟨*proof*⟩

lemma *nth-upd-array-neq-array* [*simp*]:
a =!!= *b* ⟹ *get-array a* (*upd b j v h*) ! *i* = *get-array a h* ! *i*
⟨*proof*⟩

lemma *get-arry-array-upd-elem-neqIndex* [*simp*]:
i ≠ *j* ⟹ *get-array a* (*upd a j v h*) ! *i* = *get-array a h* ! *i*
⟨*proof*⟩

lemma *length-upd-eq* [*simp*]:
length a (*upd a i v h*) = *length a h*
⟨*proof*⟩

lemma *length-upd-neq* [*simp*]:
length a (*upd b i v h*) = *length a h*
⟨*proof*⟩

lemma *upd-swap-neqArray*:
a =!!= *a'* ⟹
upd a i v (*upd a' i' v' h*)
= *upd a' i' v'* (*upd a i v h*)
⟨*proof*⟩

lemma *upd-swap-neqIndex*:
⟦ *i* ≠ *i'* ⟧ ⟹ *upd a i v* (*upd a i' v' h*) = *upd a i' v'* (*upd a i v h*)
⟨*proof*⟩

lemma *get-array-init-array-list*:
get-array (*fst* (*array-of-list ls h*)) (*snd* (*array-of-list ls' h*)) = *ls'*
⟨*proof*⟩

lemma *set-array*:
set-array (*fst* (*array-of-list ls h*))
new-ls (*snd* (*array-of-list ls h*))
= *snd* (*array-of-list new-ls h*)
⟨*proof*⟩

lemma *array-present-upd* [*simp*]:
array-present a (*upd b i v h*) = *array-present a h*
⟨*proof*⟩

lemma *array-of-list-replicate*:
array-of-list (*replicate n x*) = *array n x*
⟨*proof*⟩

Properties of imperative references

lemma *next-ref-fresh* [*simp*]:
assumes (*r*, *h'*) = *new-ref h*
shows ¬ *ref-present r h*
⟨*proof*⟩

lemma *next-ref-present* [*simp*]:
assumes (*r*, *h'*) = *new-ref h*
shows *ref-present r h'*
⟨*proof*⟩

lemma *ref-get-set-eq* [*simp*]: *get-ref r* (*set-ref r x h*) = *x*
⟨*proof*⟩

lemma *ref-get-set-neq* [*simp*]: *r* =!= *s* ⟹ *get-ref r* (*set-ref s x h*) = *get-ref r h*
⟨*proof*⟩

lemma *ref-set-get*: *set-ref r* (*get-ref r h*) *h* = *h*
⟨*proof*⟩

lemma *set-ref-same*[*simp*]:
set-ref r x (*set-ref r y h*) = *set-ref r x h*
⟨*proof*⟩

lemma *ref-set-set-swap*:
r =!= *r'* ⟹ *set-ref r x* (*set-ref r' x' h*) = *set-ref r' x'* (*set-ref r x h*)
⟨*proof*⟩

lemma *ref-new-set*: *fst* (*ref v* (*set-ref r v' h*)) = *fst* (*ref v h*)

⟨*proof*⟩

lemma *ref-get-new* [*simp*]:
get-ref (*fst* (*ref v h*)) (*snd* (*ref v' h*)) = *v'*
⟨*proof*⟩

lemma *ref-set-new* [*simp*]:
set-ref (*fst* (*ref v h*)) *new-v* (*snd* (*ref v h*)) = *snd* (*ref new-v h*)
⟨*proof*⟩

lemma *ref-get-new-neq*: *r* =!= (*fst* (*ref v h*)) ⟹
get-ref r (*snd* (*ref v h*)) = *get-ref r h*
⟨*proof*⟩

lemma *lim-set-ref* [*simp*]:
lim (*set-ref r v h*) = *lim h*
⟨*proof*⟩

lemma *ref-present-new-ref* [*simp*]:
ref-present r h ⟹ *ref-present r* (*snd* (*ref v h*))
⟨*proof*⟩

lemma *ref-present-set-ref* [*simp*]:
ref-present r (*set-ref r' v h*) = *ref-present r h*
⟨*proof*⟩

lemma *array-ranI*: ⟦ *Some b* = *get-array a h* ! *i*; *i* < *Heap.length a h* ⟧ ⟹ *b* ∈ *array-ran a h*
⟨*proof*⟩

lemma *array-ran-upd-array-Some*:
assumes *cl* ∈ *array-ran a* (*Heap.upd a i* (*Some b*) *h*)
shows *cl* ∈ *array-ran a h* ∨ *cl* = *b*
⟨*proof*⟩

lemma *array-ran-upd-array-None*:
assumes *cl* ∈ *array-ran a* (*Heap.upd a i None h*)
shows *cl* ∈ *array-ran a h*
⟨*proof*⟩

Non-interaction between imperative array and imperative references

lemma *get-array-set-ref* [*simp*]: *get-array a* (*set-ref r v h*) = *get-array a h*
⟨*proof*⟩

lemma *nth-set-ref* [*simp*]: *get-array a* (*set-ref r v h*) ! *i* = *get-array a h* ! *i*
⟨*proof*⟩

lemma *get-ref-upd* [*simp*]: *get-ref r* (*upd a i v h*) = *get-ref r h*
⟨*proof*⟩

lemma *new-ref-upd*: *fst* (*ref v* (*upd a i v'* *h*)) = *fst* (*ref v h*)
⟨*proof*⟩

not actually true ???

lemma *upd-set-ref-swap*: *upd a i v* (*set-ref r v'* *h*) = *set-ref r v'* (*upd a i v h*)
⟨*proof*⟩

lemma *length-new-ref*[*simp*]:
length a (*snd* (*ref v h*)) = *length a h*
⟨*proof*⟩

lemma *get-array-new-ref* [*simp*]:
get-array a (*snd* (*ref v h*)) = *get-array a h*
⟨*proof*⟩

lemma *ref-present-upd* [*simp*]:
ref-present r (*upd a i v h*) = *ref-present r h*
⟨*proof*⟩

lemma *array-present-set-ref* [*simp*]:
array-present a (*set-ref r v h*) = *array-present a h*
⟨*proof*⟩

lemma *array-present-new-ref* [*simp*]:
array-present a h ⟹ *array-present a* (*snd* (*ref v h*))
⟨*proof*⟩

hide (**open**) *const empty array array-of-list upd length ref*

end

32 Heap-Monad: A monad with a polymorphic heap

theory *Heap-Monad*
imports *Heap*
begin

32.1 The monad

32.1.1 Monad combinators

datatype *exception* = *Exn*

Monadic heap actions either produce values and transform the heap, or fail

datatype *'a Heap* = *Heap heap* ⇒ (*'a* + *exception*) × *heap*

primrec
execute :: *'a Heap* ⇒ *heap* ⇒ (*'a* + *exception*) × *heap* **where**
execute (*Heap f*) = *f*
lemmas [*code del*] = *execute.simps*

lemma *Heap-execute* [*simp*]:
Heap (*execute f*) = *f* ⟨*proof*⟩

lemma *Heap-eqI*:
(⋀*h*. *execute f h* = *execute g h*) ⟹ *f* = *g*
⟨*proof*⟩

lemma *Heap-eqI'*:
(⋀*h*. (λ*x*. *execute* (*f x*) *h*) = (λ*y*. *execute* (*g y*) *h*)) ⟹ *f* = *g*
⟨*proof*⟩

lemma *Heap-strip*: (⋀*f*. *PROP P f*) ≡ (⋀*g*. *PROP P* (*Heap g*))
⟨*proof*⟩

definition
heap :: (*heap* ⇒ *'a* × *heap*) ⇒ *'a Heap* **where**
[*code del*]: *heap f* = *Heap* (λ*h*. *apfst Inl* (*f h*))

lemma *execute-heap* [*simp*]:
execute (*heap f*) *h* = *apfst Inl* (*f h*)
⟨*proof*⟩

definition
run :: *'a Heap* ⇒ *'a Heap* **where**
run-drop [*code del*]: *run f* = *f*

definition
bindM :: *'a Heap* ⇒ (*'a* ⇒ *'b Heap*) ⇒ *'b Heap* (**infixl** >>= *54*) **where**
[*code del*]: *f* >>= *g* = *Heap* (λ*h*. *case execute f h of*
(*Inl x*, *h'*) ⇒ *execute* (*g x*) *h'*
| *r* ⇒ *r*)

notation
bindM (**infixl** ≫= *54*)

abbreviation
chainM :: *'a Heap* ⇒ *'b Heap* ⇒ *'b Heap* (**infixl** >> *54*) **where**
f >> *g* ≡ *f* >>= (λ-. *g*)

notation
chainM (**infixl** ≫ *54*)

definition

return :: *'a* ⇒ *'a Heap* **where**
[*code del*]: *return x* = *heap* (*Pair x*)

lemma *execute-return* [*simp*]:
execute (*return x*) *h* = *apfst Inl* (*x*, *h*)
⟨*proof*⟩

definition
raise :: *string* ⇒ *'a Heap* **where** — the string is just decoration
[*code del*]: *raise s* = *Heap* (*Pair* (*Inr Exn*))

notation (*latex* **output**)
raise (*raise*)

lemma *execute-raise* [*simp*]:
execute (*raise s*) *h* = (*Inr Exn*, *h*)
⟨*proof*⟩

32.1.2 do-syntax

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

nonterminals *do-expr*

syntax
-do :: *do-expr* ⇒ *'a*
((*do* (-)//*done*) [*12*] *100*)
-bindM :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(- <− -;//- [*1000*, *13*, *12*] *12*)
-chainM :: *'a* ⇒ *do-expr* ⇒ *do-expr*
(-;//- [*13*, *12*] *12*)
-let :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*let* - = -;//- [*1000*, *13*, *12*] *12*)
-nil :: *'a* ⇒ *do-expr*
(- [*12*] *12*)

syntax (*xsymbols*)
-bindM :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(- ← -;//- [*1000*, *13*, *12*] *12*)
syntax (*latex* **output**)
-do :: *do-expr* ⇒ *'a*
((*do* (-)) [*12*] *100*)
-let :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*let* - = -;//- [*1000*, *13*, *12*] *12*)
notation (*latex* **output**)
return (*return*)

translations
-do f => *CONST run f*

-bindM x f g => f ≫= (λx. g)
-chainM f g => f ≫ g
-let x t f => CONST Let t (λx. f)
-nil f => f

⟨*ML*⟩

32.1.3 Plain evaluation

definition
evaluate :: *'a Heap ⇒ 'a*
where
[*code del*]: *evaluate f* = (*case execute f Heap.empty*
of (*Inl x*, -) ⇒ *x*)

32.2 Monad properties

32.2.1 Superfluous runs

run is just a doodle

lemma *run-simp* [*simp*]:
⋀*f*. *run* (*run f*) = *run f*
⋀*f g*. *run f* ≫= *g* = *f* ≫= *g*
⋀*f g*. *run f* ≫ *g* = *f* ≫ *g*
⋀*f g*. *f* ≫= (λ*x*. *run g*) = *f* ≫= (λ*x*. *g*)
⋀*f g*. *f* ≫ *run g* = *f* ≫ *g*
⋀*f*. *f* = *run g* ⟷ *f* = *g*
⋀*f*. *run f* = *g* ⟷ *f* = *g*
⟨*proof*⟩

32.2.2 Monad laws

lemma *return-bind*: *return x* ≫= *f* = *f x*
⟨*proof*⟩

lemma *bind-return*: *f* ≫= *return* = *f*
⟨*proof*⟩

lemma *bind-bind*: (*f* ≫= *g*) ≫= *h* = *f* ≫= (λ*x*. *g x* ≫= *h*)
⟨*proof*⟩

lemma *bind-bind'*: *f* ≫= (λ*x*. *g x* ≫= *h x*) = *f* ≫= (λ*x*. *g x* ≫= (λ*y*. *return* (*x*, *y*))) ≫= (λ(*x*, *y*). *h x y*)
⟨*proof*⟩

lemma *raise-bind*: *raise e* ≫= *f* = *raise e*
⟨*proof*⟩

lemmas *monad-simp* = *return-bind bind-return bind-bind raise-bind*

32.3 Generic combinators

definition
liftM :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b\ Heap$
where
$liftM\ f = return\ o\ f$

definition
compM :: $('a \Rightarrow 'b\ Heap) \Rightarrow ('b \Rightarrow 'c\ Heap) \Rightarrow 'a \Rightarrow 'c\ Heap$ (**infixl** $>>==\ 54$)
where
$(f\ >>==\ g) = (\lambda x.\ f\ x \gg= g)$

notation
compM (**infixl** $\gg==\ 54$)

lemma *liftM-collapse*: $liftM\ f\ x = return\ (f\ x)$
⟨*proof*⟩

lemma *liftM-compM*: $liftM\ f \gg== g = g\ o\ f$
⟨*proof*⟩

lemma *compM-return*: $f \gg== return = f$
⟨*proof*⟩

lemma *compM-compM*: $(f \gg== g) \gg== h = f \gg== (g \gg== h)$
⟨*proof*⟩

lemma *liftM-bind*:
$(\lambda x.\ liftM\ f\ x \gg= liftM\ g) = liftM\ (\lambda x.\ g\ (f\ x))$
⟨*proof*⟩

lemma *liftM-comp*:
$liftM\ f\ o\ g = liftM\ (f\ o\ g)$
⟨*proof*⟩

lemmas *monad-simp'* = *monad-simp liftM-compM compM-return*
compM-compM liftM-bind liftM-comp

primrec
mapM :: $('a \Rightarrow 'b\ Heap) \Rightarrow 'a\ list \Rightarrow 'b\ list\ Heap$
where
$mapM\ f\ [] = return\ []$
$|\ mapM\ f\ (x\#xs) = do\ y \leftarrow f\ x;$
$\qquad\qquad ys \leftarrow mapM\ f\ xs;$
$\qquad\qquad return\ (y\ \#\ ys)$
$\qquad done$

primrec
foldM :: $('a \Rightarrow 'b \Rightarrow 'b\ Heap) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'b\ Heap$
where

foldM f [] *s* = *return s*
| *foldM f* (*x*#*xs*) *s* = *f x s* ⩾= *foldM f xs*

hide (**open**) *const heap execute*

32.4 Code generator setup

32.4.1 Logical intermediate layer

definition
Fail :: *message-string* ⇒ *exception*
where
[*code func del*]: *Fail s* = *Exn*

definition
raise-exc :: *exception* ⇒ *'a Heap*
where
[*code func del*]: *raise-exc e* = *raise* []

lemma *raise-raise-exc* [*code func*, *code inline*]:
raise s = *raise-exc* (*Fail* (*STR s*))
⟨*proof*⟩

hide (**open**) *const Fail raise-exc*

32.4.2 SML

code-type *Heap* (*SML unit*/ −>/ -)
code-const *Heap* (*SML raise*/ (*Fail*/ *bare Heap*))
code-monad *run op* ⩾= *return* () *SML*
code-const *run* (*SML* -)
code-const *return* (*SML* (*fn*/ ()/ =>/ -))
code-const *Heap-Monad.Fail* (*SML Fail*)
code-const *Heap-Monad.raise-exc* (*SML* (*fn*/ ()/ =>/ *raise*/ -))

32.4.3 OCaml

code-type *Heap* (*OCaml* -)
code-const *Heap* (*OCaml failwith*/ *bare Heap*)
code-monad *run op* ⩾= *return* () *OCaml*
code-const *run* (*OCaml* -)
code-const *return* (*OCaml* (*fn*/ ()/ =>/ -))
code-const *Heap-Monad.Fail* (*OCaml Failure*)
code-const *Heap-Monad.raise-exc* (*OCaml* (*fn*/ ()/ =>/ *raise*/ -))

code-reserved *OCaml Failure raise*

32.4.4 Haskell

Adaption layer

code-include *Haskell STMonad*
⟨⟨*import qualified Control.Monad*;
import qualified Control.Monad.ST;
import qualified Data.STRef;
import qualified Data.Array.ST;

type ST s a = Control.Monad.ST.ST s a;
type STRef s a = Data.STRef.STRef s a;
type STArray s a = Data.Array.ST.STArray s Integer a;

runST :: (*forall s. ST s a*) −> *a*;
runST s = Control.Monad.ST.runST s;

newSTRef = Data.STRef.newSTRef;
readSTRef = Data.STRef.readSTRef;
writeSTRef = Data.STRef.writeSTRef;

newArray :: (*Integer*, *Integer*) −> *a* −> *ST s* (*STArray s a*);
newArray = Data.Array.ST.newArray;

newListArray :: (*Integer*, *Integer*) −> [*a*] −> *ST s* (*STArray s a*);
newListArray = Data.Array.ST.newListArray;

length :: *STArray s a* −> *ST s Integer*;
length a = Control.Monad.liftM snd (*Data.Array.ST.getBounds a*);

readArray :: *STArray s a* −> *Integer* −> *ST s a*;
readArray = Data.Array.ST.readArray;

writeArray :: *STArray s a* −> *Integer* −> *a* −> *ST s* ();
writeArray = Data.Array.ST.writeArray;⟩⟩

code-reserved *Haskell ST STRef Array*
runST
newSTRef reasSTRef writeSTRef
newArray newListArray bounds readArray writeArray

Monad

code-type *Heap* (*Haskell ST 's -*)
code-const *Heap* (*Haskell error bare Heap*))
code-const *evaluate* (*Haskell runST*)
code-monad *run op* ≫= *Haskell*
code-const *return* (*Haskell return*)
code-const *Heap-Monad.Fail* (*Haskell -*)
code-const *Heap-Monad.raise-exc* (*Haskell error*)

end

33 Array: Monadic arrays

theory *Array*
imports *Heap-Monad Code-Index*
begin

33.1 Primitives

definition
new :: *nat* ⇒ *'a*::*heap* ⇒ *'a array Heap* **where**
[*code del*]: *new n x* = *Heap-Monad.heap* (*Heap.array n x*)

definition
of-list :: *'a*::*heap list* ⇒ *'a array Heap* **where**
[*code del*]: *of-list xs* = *Heap-Monad.heap* (*Heap.array-of-list xs*)

definition
length :: *'a*::*heap array* ⇒ *nat Heap* **where**
[*code del*]: *length arr* = *Heap-Monad.heap* (λ*h*. (*Heap.length arr h*, *h*))

definition
nth :: *'a*::*heap array* ⇒ *nat* ⇒ *'a Heap*
where
[*code del*]: *nth a i* = (*do len* ← *length a*;
(*if i* < *len*
then Heap-Monad.heap (λ*h*. (*get-array a h* ! *i*, *h*))
else raise (*''array lookup*: *index out of range''*))
done)

— FIXME adjustion for List theory
no-syntax
nth :: *'a list* ⇒ *nat* ⇒ *'a* (**infixl** ! *100*)

abbreviation
nth-list :: *'a list* ⇒ *nat* ⇒ *'a* (**infixl** ! *100*)
where
nth-list ≡ *List.nth*

definition
upd :: *nat* ⇒ *'a* ⇒ *'a*::*heap array* ⇒ *'a*::*heap array Heap*
where
[*code del*]: *upd i x a* = (*do len* ← *length a*;
(*if i* < *len*
then Heap-Monad.heap (λ*h*. (*a*, *Heap.upd a i x h*))
else raise (*''array update*: *index out of range''*))
done)

lemma *upd-return*:
upd i x a ≫ *return a* = *upd i x a*
⟨*proof*⟩

33.2 Derivates

definition
map-entry :: *nat* ⇒ (*'a*::*heap* ⇒ *'a*) ⇒ *'a array* ⇒ *'a array Heap*
where
map-entry i f a = (*do*
x ← *nth a i*;
upd i (*f x*) *a*
done)

definition
swap :: *nat* ⇒ *'a* ⇒ *'a*::*heap array* ⇒ *'a Heap*
where
swap i x a = (*do*
y ← *nth a i*;
upd i x a;
return x
done)

definition
make :: *nat* ⇒ (*nat* ⇒ *'a*::*heap*) ⇒ *'a array Heap*
where
make n f = *of-list* (*map f* [*0* ..< *n*])

definition
freeze :: *'a*::*heap array* ⇒ *'a list Heap*
where
freeze a = (*do*
n ← *length a*;
mapM (*nth a*) [*0*..<*n*]
done)

definition
map :: (*'a*::*heap* ⇒ *'a*) ⇒ *'a array* ⇒ *'a array Heap*
where
map f a = (*do*
n ← *length a*;
foldM (λ*n*. *map-entry n f*) [*0*..<*n*] *a*
done)

hide (**open**) *const new map* — avoid clashed with some popular names

33.3 Properties

lemma *array-make* [*code func*]:
Array.new n x = *make n* (λ-. *x*)
⟨*proof*⟩

lemma *array-of-list-make* [*code func*]:
of-list xs = *make* (*List.length xs*) (λ*n*. *xs* ! *n*)

⟨*proof*⟩

33.4 Code generator setup

33.4.1 Logical intermediate layer

definition *new'* **where**
  [*code del*]: *new' = Array.new o nat-of-index*
hide (**open**) *const new'*
lemma [*code func*]:
  Array.new = Array.new' o index-of-nat
  ⟨*proof*⟩

definition *of-list'* **where**
  [*code del*]: *of-list' i xs = Array.of-list (take (nat-of-index i) xs)*
hide (**open**) *const of-list'*
lemma [*code func*]:
  Array.of-list xs = Array.of-list' (index-of-nat (List.length xs)) xs
  ⟨*proof*⟩

definition *make'* **where**
  [*code del*]: *make' i f = Array.make (nat-of-index i) (f o index-of-nat)*
hide (**open**) *const make'*
lemma [*code func*]:
  Array.make n f = Array.make' (index-of-nat n) (f o nat-of-index)
  ⟨*proof*⟩

definition *length'* **where**
  [*code del*]: *length' = Array.length ⤜== liftM index-of-nat*
hide (**open**) *const length'*
lemma [*code func*]:
  Array.length = Array.length' ⤜== liftM nat-of-index
  ⟨*proof*⟩

definition *nth'* **where**
  [*code del*]: *nth' a = Array.nth a o nat-of-index*
hide (**open**) *const nth'*
lemma [*code func*]:
  Array.nth a n = Array.nth' a (index-of-nat n)
  ⟨*proof*⟩

definition *upd'* **where**
  [*code del*]: *upd' a i x = Array.upd (nat-of-index i) x a ⤜ return ()*
hide (**open**) *const upd'*
lemma [*code func*]:
  Array.upd i x a = Array.upd' a (index-of-nat i) x ⤜ return a
  ⟨*proof*⟩

33.4.2 SML

code-type *array* (*SML -/ array*)
code-const *Array* (*SML raise/* (*Fail/ bare Array*))
code-const *Array.new'* (*SML* (*fn/* ()*/ =>/ Array.array/* ((-),/ (-))))
code-const *Array.of-list* (*SML* (*fn/* ()*/ =>/ Array.fromList/* -))
code-const *Array.make'* (*SML* (*fn/* ()*/ =>/ Array.tabulate/* ((-),/ (-))))
code-const *Array.length'* (*SML* (*fn/* ()*/ =>/ Array.length/* -))
code-const *Array.nth'* (*SML* (*fn/* ()*/ =>/ Array.sub/* ((-),/ (-))))
code-const *Array.upd'* (*SML* (*fn/* ()*/ =>/ Array.update/* ((-),/ (-),/ (-))))

code-reserved *SML Array*

33.4.3 OCaml

code-type *array* (*OCaml -/ array*)
code-const *Array* (*OCaml failwith/ bare Array*)
code-const *Array.new'* (*OCaml* (*fn/* ()*/ =>/ Array.make/ -/* -))
code-const *Array.of-list* (*OCaml* (*fn/* ()*/ =>/ Array.of'-list/* -))
code-const *Array.make'* (*OCaml* (*fn/* ()*/ =>/ Array.init/ -/* -))
code-const *Array.length'* (*OCaml* (*fn/* ()*/ =>/ Array.length/* -))
code-const *Array.nth'* (*OCaml* (*fn/* ()*/ =>/ Array.get/ -/* -))
code-const *Array.upd'* (*OCaml* (*fn/* ()*/ =>/ Array.set/ -/ -/* -))

code-reserved *OCaml Array*

33.4.4 Haskell

code-type *array* (*Haskell STArray '-s* -)
code-const *Array* (*Haskell error/ bare Array*)
code-const *Array.new'* (*Haskell newArray/* (*0*,/ -))
code-const *Array.of-list'* (*Haskell newListArray/* (*0*,/ -))
code-const *Array.length'* (*Haskell length*)
code-const *Array.nth'* (*Haskell readArray*)
code-const *Array.upd'* (*Haskell writeArray*)

end

34 Ref: Monadic references

theory *Ref*
imports *Heap-Monad*
begin

Imperative reference operations; modeled after their ML counterparts.
See http://caml.inria.fr/pub/docs/manual-caml-light/node14.15.html and http://www.smlnj.org/doc
level-comparison.html

34.1 Primitives

definition
new :: *'a*::*heap* ⇒ *'a ref Heap* **where**
[*code del*]: *new v* = *Heap-Monad.heap* (*Heap.ref v*)

definition
lookup :: *'a*::*heap ref* ⇒ *'a Heap* (!- *61*) **where**
[*code del*]: *lookup r* = *Heap-Monad.heap* (λ*h*. (*get-ref r h*, *h*))

definition
update :: *'a ref* ⇒ (*'a*::*heap*) ⇒ *unit Heap* (- := - *62*) **where**
[*code del*]: *update r e* = *Heap-Monad.heap* (λ*h*. ((), *set-ref r e h*))

34.2 Derivates

definition
change :: (*'a*::*heap* ⇒ *'a*) ⇒ *'a ref* ⇒ *'a Heap*
where
change f r = (*do x* ← ! *r*;
let y = *f x*;
r := *y*;
return y
done)

hide (**open**) *const new lookup update change*

34.3 Properties

lemma *lookup-chain*:
(!*r* ≫ *f*) = *f*
⟨*proof*⟩

lemma *update-change* [*code func*]:
r := *e* = *Ref.change* (λ-. *e*) *r* ≫ *return* ()
⟨*proof*⟩

34.4 Code generator setup

34.4.1 SML

code-type *ref* (*SML* -/ *ref*)
code-const *Ref* (*SML raise*/ (*Fail*/ *bare Ref*))
code-const *Ref.new* (*SML* (*fn*/ ()/ =>/ *ref*/ -))
code-const *Ref.lookup* (*SML* (*fn*/ ()/ =>/ !/ -))
code-const *Ref.update* (*SML* (*fn*/ ()/ =>/ -/ :=/ -))

code-reserved *SML ref*

34.4.2 OCaml

code-type *ref* (*OCaml* -/ *ref*)
code-const *Ref* (*OCaml failwith*/ *bare Ref*))
code-const *Ref.new* (*OCaml* (*fn*/ ()/ =>/ *ref*/ -))
code-const *Ref.lookup* (*OCaml* (*fn*/ ()/ =>/ !/ -))
code-const *Ref.update* (*OCaml* (*fn*/ ()/ =>/ -/ :=/ -))

code-reserved *OCaml ref*

34.4.3 Haskell

code-type *ref* (*Haskell STRef* 'a -)
code-const *Ref* (*Haskell error*/ *bare Ref*)
code-const *Ref.new* (*Haskell newSTRef*)
code-const *Ref.lookup* (*Haskell readSTRef*)
code-const *Ref.update* (*Haskell writeSTRef*)

end

35 Imperative-HOL: Entry point into monadic imperative HOL

theory *Imperative-HOL*
imports *Array Ref*
begin

end

36 Infinite-Set: Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *ATP-Linkup*
begin

36.1 Infinite Sets

Some elementary facts about infinite sets, mostly by Stefan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

abbreviation
infinite :: 'a *set* ⇒ *bool* **where**
infinite S == ¬ *finite S*

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

lemma *infinite-imp-nonempty*: *infinite* S ==> $S \neq \{\}$
⟨*proof*⟩

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

lemma *int-infinite* [*simp*]:
shows *infinite* $(UNIV::int\ set)$
⟨*proof*⟩

The "only if" direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

lemma *linorder-injI*:
assumes *hyp*: $!!x\ y.\ x < (y::'a::linorder) ==> f\ x \neq f\ y$
shows *inj f*
⟨*proof*⟩

lemma *infinite-countable-subset*:
assumes *inf*: *infinite* $(S::'a\ set)$
shows $\exists f.$ *inj* $(f::nat \Rightarrow 'a) \wedge range\ f \subseteq S$
⟨*proof*⟩

lemma *infinite-iff-countable-subset*:
infinite $S = (\exists f.$ *inj* $(f::nat \Rightarrow 'a) \wedge range\ f \subseteq S)$
⟨*proof*⟩

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

lemma *inf-img-fin-dom*:
assumes *img*: *finite* $(f`A)$ **and** *dom*: *infinite A*
shows $\exists y \in f`A.$ *infinite* $(f -` \{y\})$
⟨*proof*⟩

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

36.2 Infinitely Many and Almost All

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

definition
Inf-many :: (′*a* ⇒ *bool*) ⇒ *bool* (**binder** *INFM* 10) **where**
Inf-many P = *infinite* {*x*. *P x*}

definition
Alm-all :: (′*a* ⇒ *bool*) ⇒ *bool* (**binder** *MOST* 10) **where**
Alm-all P = (¬ (*INFM x*. ¬ *P x*))

notation (*xsymbols*)
Inf-many (**binder** ∃∞ 10) **and**
Alm-all (**binder** ∀∞ 10)

notation (*HTML* **output**)
Inf-many (**binder** ∃∞ 10) **and**
Alm-all (**binder** ∀∞ 10)

lemma *INF-EX*:
(∃∞*x*. *P x*) ⟹ (∃*x*. *P x*)
⟨*proof*⟩

lemma *MOST-iff-finiteNeg*: (∀∞*x*. *P x*) = *finite* {*x*. ¬ *P x*}
⟨*proof*⟩

lemma *ALL-MOST*: ∀*x*. *P x* ⟹ ∀∞*x*. *P x*
⟨*proof*⟩

lemma *INF-mono*:
assumes *inf*: ∃∞*x*. *P x* **and** *q*: ⋀*x*. *P x* ⟹ *Q x*
shows ∃∞*x*. *Q x*
⟨*proof*⟩

lemma *MOST-mono*: ∀∞*x*. *P x* ⟹ (⋀*x*. *P x* ⟹ *Q x*) ⟹ ∀∞*x*. *Q x*
⟨*proof*⟩

lemma *INF-nat*: (∃∞*n*. *P* (*n*::*nat*)) = (∀*m*. ∃*n*. *m*<*n* ∧ *P n*)
⟨*proof*⟩

lemma *INF-nat-le*: (∃∞*n*. *P* (*n*::*nat*)) = (∀*m*. ∃*n*. *m*≤*n* ∧ *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*⟩

36.3 Enumeration of an Infinite Set

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

consts
enumerate :: *'a::wellorder set => (nat => '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*⟩

36.4 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

definition
atmost-one :: *'a set* ⇒ *bool* **where**
atmost-one S = $(\forall x\ y.\ x\in S \wedge y\in S \longrightarrow x=y)$

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* ∈ *S* $\Longrightarrow$ *y* ∈ *S* $\Longrightarrow$ *y* = *x*
⟨*proof*⟩

end

37 ListVector: Lists as vectors

theory *ListVector*
imports *Main*
begin

A vector-space like structure of lists and arithmetic operations on them. Is only a vector space if restricted to lists of the same length.

Multiplication with a scalar:

abbreviation *scale* :: ('*a*::*times*) ⇒ '*a list* ⇒ '*a list* (**infix** $*_s$ *70*)
where $x *_s xs \equiv map\ (op * x)\ xs$

lemma *scale1*[*simp*]: (*1*::'*a*::*monoid-mult*) $*_s$ *xs* = *xs*
⟨*proof*⟩

37.1 + and −

fun *zipwith0* :: ('*a*::*zero* ⇒ '*b*::*zero* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list*
where
zipwith0 f [] [] = [] |
zipwith0 f (*x*#*xs*) (*y*#*ys*) = *f x y* # *zipwith0 f xs ys* |
zipwith0 f (*x*#*xs*) [] = *f x 0* # *zipwith0 f xs* [] |
zipwith0 f [] (*y*#*ys*) = *f 0 y* # *zipwith0 f* [] *ys*

instance *list* :: ({*zero*,*plus*})*plus*
list-add-def : *op* + ≡ *zipwith0* (*op* +) ⟨*proof*⟩

instance *list* :: ({*zero*,*uminus*})*uminus*
list-uminus-def: *uminus* ≡ *map uminus* ⟨*proof*⟩

instance *list* :: ({*zero*,*minus*})*minus*
list-diff-def: *op* − ≡ *zipwith0* (*op* −) ⟨*proof*⟩

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*⟩

37.2 Inner product

definition *iprod* :: '*a*::*ring list* ⇒ '*a list* ⇒ '*a* (⟨-,-⟩) **where**
⟨*xs*,*ys*⟩ = (∑ (*x*,*y*) ← *zip xs ys*. *x*∗*y*)

lemma *iprod-Nil*[*simp*]: ⟨[],*ys*⟩ = *0*
⟨*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*: ⟨*xs* + *ys*,*zs*⟩ = ⟨*xs*,*zs*⟩ + ⟨*ys*,*zs*⟩
⟨*proof*⟩

lemma *iprod-left-diff-distrib*: ⟨*xs* − *ys*, *zs*⟩ = ⟨*xs*,*zs*⟩ − ⟨*ys*,*zs*⟩
⟨*proof*⟩

lemma *iprod-assoc*: ⟨*x* ∗$_s$ *xs*, *ys*⟩ = *x* ∗ ⟨*xs*,*ys*⟩

⟨*proof*⟩

end

38 Multiset: Multisets

theory *Multiset*
imports *List*
begin

38.1 The type of multisets

typedef *'a multiset* = {*f*::*'a* => *nat*. *finite* {*x* . *f x* > *0*}}
⟨*proof*⟩

lemmas *multiset-typedef* [*simp*] =
Abs-multiset-inverse Rep-multiset-inverse Rep-multiset
and [*simp*] = *Rep-multiset-inject* [*symmetric*]

definition
Mempty :: *'a multiset* ({#}) **where**
{#} = *Abs-multiset* (λ*a*. *0*)

definition
single :: *'a* => *'a multiset* **where**
single a = *Abs-multiset* (λ*b*. *if b* = *a then 1 else 0*)

declare
Mempty-def[*code func del*] *single-def*[*code func del*]

definition
count :: *'a multiset* => *'a* => *nat* **where**
count = *Rep-multiset*

definition
MCollect :: *'a multiset* => (*'a* => *bool*) => *'a multiset* **where**
MCollect M P = *Abs-multiset* (λ*x*. *if P x then Rep-multiset M x else 0*)

abbreviation
Melem :: *'a* => *'a multiset* => *bool* ((-/ :# -) [*50*, *51*] *50*) **where**
a :# *M* == *0* < *count M a*

notation (*xsymbols*)
Melem (**infix** ∈# *50*)

syntax
-MCollect :: *pttrn* => *'a multiset* => *bool* => *'a multiset* ((*1*{# - :# -./ -#}))
translations
{#*x* :# *M*. *P*#} == *CONST MCollect M* (λ*x*. *P*)

definition
set-of :: *'a multiset => 'a set* **where**
set-of M = {*x*. *x* :# *M*}

instantiation *multiset* :: (*type*) {*plus*, *minus*, *zero*, *size*}
begin

definition
union-def[*code func del*]:
M + *N* = *Abs-multiset* (λ*a*. *Rep-multiset M a* + *Rep-multiset N a*)

definition
diff-def: *M* − *N* = *Abs-multiset* (λ*a*. *Rep-multiset M a* − *Rep-multiset N a*)

definition
Zero-multiset-def [*simp*]: *0* = {#}

definition
size-def[*code func del*]: *size M* = *setsum* (*count M*) (*set-of M*)

instance ⟨*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

38.2 Algebraic properties

38.2.1 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*⟩

38.2.2 Difference

lemma *diff-empty* [*simp*]: *M* − {#} = *M* ∧ {#} − *M* = {#}
⟨*proof*⟩

lemma *diff-union-inverse2* [*simp*]: *M* + {#*a*#} − {#*a*#} = *M*
⟨*proof*⟩

lemma *diff-cancel*: *A* − *A* = {#}
⟨*proof*⟩

38.2.3 Count of elements

lemma *count-empty* [*simp*]: *count* {#} *a* = *0*
⟨*proof*⟩

lemma *count-single* [*simp*]: *count* {#*b*#} *a* = (*if b* = *a then 1 else 0*)
⟨*proof*⟩

lemma *count-union* [*simp*]: *count* (*M* + *N*) *a* = *count M a* + *count N a*

⟨*proof*⟩

lemma *count-diff* [*simp*]: *count* (*M* − *N*) *a* = *count M a* − *count N a*
⟨*proof*⟩

lemma *count-MCollect* [*simp*]:
count {# *x*:#*M*. *P x* #} *a* = (*if P a then count M a else 0*)
⟨*proof*⟩

38.2.4 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*) = *set-of M* ∪ *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*⟩

38.2.5 Size

lemma *size-empty* [*simp*,*code func*]: *size* {#} = *0*
⟨*proof*⟩

lemma *size-single* [*simp*,*code func*]: *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*,*code func*]: *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*: $(S \neq \{\#\}) = (0 < size\ S)$
⟨*proof*⟩

lemma *size-eq-Suc-imp-elem*: $size\ M = Suc\ n ==> \exists a.\ a :\# M$
⟨*proof*⟩

38.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: $(M = N) = (\forall a.\ count\ M\ a = count\ N\ a)$
⟨*proof*⟩

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
⟨*proof*⟩

lemma *single-eq-single* [*simp*]: $(\{\#a\#\} = \{\#b\#\}) = (a = b)$
⟨*proof*⟩

lemma *union-eq-empty* [*iff*]: $(M + N = \{\#\}) = (M = \{\#\} \wedge N = \{\#\})$
⟨*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*⟩

38.2.7 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*⟩

38.2.8 Comprehension (filter)

lemma *MCollect-empty*[*simp*, *code func*]: $MCollect \ \{\#\} \ P = \{\#\}$
⟨*proof*⟩

lemma *MCollect-single*[*simp*, *code func*]:
$MCollect \ \{\#x\#\} \ P = (if \ P \ x \ then \ \{\#x\#\} \ else \ \{\#\})$
⟨*proof*⟩

lemma *MCollect-union*[*simp*, *code func*]:
$MCollect \ (M+N) \ f = MCollect \ M \ f + MCollect \ N \ f$
⟨*proof*⟩

38.3 Induction and case splits

lemma *setsum-decr*:
$finite \ F \ ==> \ (0::nat) < f \ a \ ==>$
$setsum \ (f \ (a := f \ a - 1)) \ F = (if \ a \in F \ then \ setsum \ f \ F - 1 \ else \ setsum \ f \ F)$
⟨*proof*⟩

lemma *rep-multiset-induct-aux*:
assumes *1*: $P \ (\lambda a. \ (0::nat))$
and *2*: $!!f \ b. \ f \in multiset \ ==> \ P \ f \ ==> \ P \ (f \ (b := f \ b + 1))$
shows $\forall f. \ f \in multiset \ --> \ setsum \ f \ \{x. \ f \ x \neq 0\} = n \ --> \ P \ f$
⟨*proof*⟩

theorem *rep-multiset-induct*:
$f \in multiset \ ==> \ P \ (\lambda a. \ 0) \ ==>$
$(!!f \ b. \ f \in multiset \ ==> \ P \ f \ ==> \ P \ (f \ (b := f \ b + 1))) \ ==> \ P \ f$
⟨*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*⟩

38.4 Orderings

38.4.1 Well-foundedness

definition
mult1 :: $('a \times 'a)\ set \Rightarrow ('a\ multiset \times 'a\ multiset)\ set$ **where**
mult1 r =
$\{(N,\ M).\ \exists a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge$
$(\forall b.\ b :\# K \longrightarrow (b,\ a) \in r)\}$

definition
mult :: $('a \times 'a)\ set \Rightarrow ('a\ multiset \times 'a\ multiset)\ set$ **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*⟩

38.4.2 Closure-free presentation

lemma *diff-union-single-conv*: $a :\# J ==> I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

One direction.

lemma *mult-implies-one-step*:
$trans\ r ==> (M, N) \in mult\ r ==>$
$\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge$
$(\forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r)$
⟨*proof*⟩

lemma *elem-imp-eq-diff-union*: $a :\# M ==> M = M - \{\#a\#\} + \{\#a\#\}$
⟨*proof*⟩

lemma *size-eq-Suc-imp-eq-union*: $size\ M = Suc\ n ==> \exists a\ N.\ M = N + \{\#a\#\}$
⟨*proof*⟩

lemma *one-step-implies-mult-aux*:
$trans\ r ==>$
$\forall I\ J\ K.\ (size\ J = n \wedge J \neq \{\#\} \wedge (\forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r))$
$--> (I + K, I + J) \in mult\ r$
⟨*proof*⟩

lemma *one-step-implies-mult*:
$trans\ r ==> J \neq \{\#\} ==> \forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r$
$==> (I + K, I + J) \in mult\ r$
⟨*proof*⟩

38.4.3 Partial-order properties

instantiation *multiset* :: (*order*) *order*
begin

definition
less-multiset-def: $M' < M \longleftrightarrow (M', M) \in mult\ \{(x', x).\ x' < x\}$

definition
le-multiset-def: $M' <= M \longleftrightarrow M' = M \vee M' < (M::'a\ multiset)$

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

38.4.4 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*⟩

38.5 Link with lists

primrec *multiset-of* :: *'a list ⇒ 'a multiset* **where**
multiset-of [] = {#} |
multiset-of (a # x) = multiset-of x + {# a #}

lemma *multiset-of-zero-iff*[*simp*]: *(multiset-of x = {#}) = (x = [])*
⟨*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*⟩

38.6 Pointwise ordering induced by count

definition
mset-le :: *'a multiset ⇒ 'a multiset ⇒ bool* (**infix** ≤# *50*) **where**
(A ≤# B) = (∀ a. count A a ≤ count B a)

definition
mset-less :: *'a multiset ⇒ 'a multiset ⇒ bool* (**infix** <# *50*) **where**

$(A <\# B) = (A \leq\# B \wedge A \neq B)$

notation *mset-le* (**infix** $\subseteq\#$ *50*)
notation *mset-less* (**infix** $\subset\#$ *50*)

lemma *mset-le-refl*[*simp*]: $A \leq\# A$
⟨*proof*⟩

lemma *mset-le-trans*: $A \leq\# B \Longrightarrow B \leq\# C \Longrightarrow A \leq\# C$
⟨*proof*⟩

lemma *mset-le-antisym*: $A \leq\# B \Longrightarrow B \leq\# A \Longrightarrow A = B$
⟨*proof*⟩

lemma *mset-le-exists-conv*: $(A \leq\# B) = (\exists\, C.\ B = A + C)$
⟨*proof*⟩

lemma *mset-le-mono-add-right-cancel*[*simp*]: $(A + C \leq\# B + C) = (A \leq\# B)$
⟨*proof*⟩

lemma *mset-le-mono-add-left-cancel*[*simp*]: $(C + A \leq\# C + B) = (A \leq\# B)$
⟨*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*:
$i < \mathit{length}\ \mathit{ls} \Longrightarrow \mathit{multiset\text{-}of}\ (\mathit{ls}[i := v]) = \mathit{multiset\text{-}of}\ \mathit{ls} - \{\#\mathit{ls}\ !\ i\#\} + \{\#v\#\}$
⟨*proof*⟩

lemma *multiset-of-swap*:
$i < \mathit{length}\ \mathit{ls} \Longrightarrow j < \mathit{length}\ \mathit{ls} \Longrightarrow$
$\mathit{multiset\text{-}of}\ (\mathit{ls}[j := \mathit{ls}\ !\ i,\ i := \mathit{ls}\ !\ j]) = \mathit{multiset\text{-}of}\ \mathit{ls}$
⟨*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-eq-less.less-trans*

lemma *mset-less-diff-self*: $c \in\# B \Longrightarrow B - \{\#c\#\} \subset\# B$
⟨*proof*⟩

38.7 Strong induction and subset induction for multisets

Well-foundedness of proper subset operator:

proper multiset subset

definition
mset-less-rel :: ('*a multiset* ∗ '*a multiset*) *set* **where**
mset-less-rel = {(*A*,*B*). *A* ⊂# *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*]

38.8 The fold combinator

The intended behaviour is *fold-mset f z* $\{\#x_1, ..., x_n\#\} = f\ x_1\ (\ldots\ (f\ x_n\ z)\ldots)$ if f is associative-commutative.

The graph of *fold-mset*, z: the start element, f: folding function, A: the multiset, y: the result.

inductive

fold-msetG :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a$ *multiset* $\Rightarrow 'b \Rightarrow$ *bool*
for $f :: 'a \Rightarrow 'b \Rightarrow 'b$
and $z :: 'b$
where
emptyI [*intro*]: *fold-msetG f z* $\{\#\}$ *z*
| *insertI* [*intro*]: *fold-msetG f z A y* $\Longrightarrow$ *fold-msetG f z* $(A + \{\#x\#\})$ $(f\ x\ y)$

inductive-cases *empty-fold-msetGE* [*elim*!]: *fold-msetG f z* $\{\#\}$ *x*
inductive-cases *insert-fold-msetGE*: *fold-msetG f z* $(A + \{\#\})$ *y*

definition
fold-mset :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a$ *multiset* $\Rightarrow 'b$ **where**
fold-mset f z A = $(THE\ x.$ *fold-msetG f z A x*$)$

lemma *Diff1-fold-msetG*:
fold-msetG f z $(A - \{\#x\#\})$ *y* $\Longrightarrow$ *x* $\in\#$ *A* $\Longrightarrow$ *fold-msetG f z A* $(f\ x\ y)$
⟨*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-msetG f z A x $\Longrightarrow$ *fold-msetG f z A y* $\Longrightarrow$ *y* = *x*
⟨*proof*⟩

lemma *fold-mset-insert-aux*:
$($*fold-msetG f z* $(A + \{\#x\#\})$ *v*$)$ =
$(\exists y.$ *fold-msetG f z A y* $\wedge$ *v* = *f x y*$)$
⟨*proof*⟩

lemma *fold-mset-equality*: *fold-msetG f z A y* $\Longrightarrow$ *fold-mset f z A* = *y*
⟨*proof*⟩

lemma *fold-mset-insert*:
fold-mset f z $(A + \{\#x\#\})$ = *f x* $($*fold-mset f z A*$)$
⟨*proof*⟩

lemma *fold-mset-insert-idem*:
fold-mset f z $(A + \{\#a\#\})$ = *f a* $($*fold-mset f z A*$)$
⟨*proof*⟩

lemma *fold-mset-commute*: $f\ x\ (\mathit{fold\text{-}mset}\ f\ z\ A) = \mathit{fold\text{-}mset}\ f\ (f\ x\ z)\ A$
⟨*proof*⟩

lemma *fold-mset-single* [*simp*]: $\mathit{fold\text{-}mset}\ f\ z\ \{\#x\#\} = f\ x\ z$
⟨*proof*⟩

lemma *fold-mset-union* [*simp*]:
$\mathit{fold\text{-}mset}\ f\ z\ (A+B) = \mathit{fold\text{-}mset}\ f\ (\mathit{fold\text{-}mset}\ f\ z\ A)\ B$
⟨*proof*⟩

lemma *fold-mset-fusion*:
includes *left-commutative g*
shows $(\bigwedge x\ y.\ h\ (g\ x\ y) = f\ x\ (h\ y)) \Longrightarrow h\ (\mathit{fold\text{-}mset}\ g\ w\ A) = \mathit{fold\text{-}mset}\ f\ (h\ w)\ A$
⟨*proof*⟩

lemma *fold-mset-rec*:
assumes $a \in\# A$
shows $\mathit{fold\text{-}mset}\ f\ z\ A = f\ a\ (\mathit{fold\text{-}mset}\ f\ z\ (A - \{\#a\#\}))$
⟨*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 $\mathit{fold\text{-}mset}\ F\ z\ \{\#\} = z$ where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

38.9 Image

definition [*code func del*]: $\mathit{image\text{-}mset}\ f == \mathit{fold\text{-}mset}\ (op\ +\ o\ \mathit{single}\ o\ f)\ \{\#\}$

interpretation *image-left-comm*: *left-commutative* [$op\ +\ o\ \mathit{single}\ o\ f$]
⟨*proof*⟩

lemma *image-mset-empty* [*simp*, *code func*]: $\mathit{image\text{-}mset}\ f\ \{\#\} = \{\#\}$
⟨*proof*⟩

lemma *image-mset-single* [*simp*, *code func*]: $\mathit{image\text{-}mset}\ f\ \{\#x\#\} = \{\#f\ x\#\}$
⟨*proof*⟩

lemma *image-mset-insert*:
$\mathit{image\text{-}mset}\ f\ (M\ +\ \{\#a\#\}) = \mathit{image\text{-}mset}\ f\ M\ +\ \{\#f\ a\#\}$
⟨*proof*⟩

lemma *image-mset-union*[*simp*, *code func*]:
$\mathit{image\text{-}mset}\ f\ (M+N) = \mathit{image\text{-}mset}\ f\ M\ +\ \mathit{image\text{-}mset}\ f\ N$

⟨*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*#}#}.

end

39 NatPair: Pairs of Natural Numbers

theory *NatPair*
imports *ATP-Linkup*
begin

An injective function from $\mathbb{N}^2$ to $\mathbb{N}$. Definition and proofs are from [4, page 85].

definition
nat2-to-nat:: (*nat* ∗ *nat*) ⇒ *nat* **where**
nat2-to-nat pair = (*let* (*n*,*m*) = *pair in* (*n*+*m*) ∗ *Suc* (*n*+*m*) *div* 2 + *n*)

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

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*⟩

end

40 Nat-Infinity: Natural numbers with infinity

theory *Nat-Infinity*
imports *ATP-Linkup*
begin

40.1 Definitions

We extend the standard natural numbers by a special value indicating infinity. This includes extending the ordering relations $op <$ and $op \leq$.

datatype *inat* = *Fin nat* | *Infty*

notation (*xsymbols*)
Infty (∞)

notation (*HTML* **output**)
Infty (∞)

definition
iSuc :: *inat* => *inat* **where**
iSuc i = (*case i of Fin n* => *Fin* (*Suc n*) | ∞ => ∞)

instantiation *inat* :: {*ord*, *zero*}
begin

definition
Zero-inat-def: *0* == *Fin 0*

definition
iless-def: *m* < *n* ==
case m of Fin m1 => (*case n of Fin n1* => *m1* < *n1* | ∞ => *True*)
| ∞ => *False*

definition
ile-def: *m* ≤ *n* ==
case n of Fin n1 => (*case m of Fin m1* => *m1* ≤ *n1* | ∞ => *False*)
| ∞ => *True*

instance ⟨*proof*⟩

end

lemmas *inat-defs* = *Zero-inat-def iSuc-def iless-def ile-def*

lemmas *inat-splits = inat.split inat.split-asm*

Below is a not quite complete set of theorems. Use the method (*simp add*: *inat-defs split*:*inat-splits*, *arith?*) to prove new theorems or solve arithmetic subgoals involving *inat* on the fly.

40.2 Constructors

lemma *Fin-0*: $Fin\ 0 = 0$
⟨*proof*⟩

lemma *Infty-ne-i0* [*simp*]: $\infty \neq 0$
⟨*proof*⟩

lemma *i0-ne-Infty* [*simp*]: $0 \neq \infty$
⟨*proof*⟩

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

lemma *iSuc-Infty* [*simp*]: $iSuc\ \infty = \infty$
⟨*proof*⟩

lemma *iSuc-ne-0* [*simp*]: $iSuc\ n \neq 0$
⟨*proof*⟩

lemma *iSuc-inject* [*simp*]: $(iSuc\ x = iSuc\ y) = (x = y)$
⟨*proof*⟩

40.3 Ordering relations

instance *inat* :: *linorder*
⟨*proof*⟩

lemma *Infty-ilessE* [*elim*!]: $\infty < Fin\ m \Longrightarrow R$
⟨*proof*⟩

lemma *iless-linear*: $m < n \vee m = n \vee n < (m::inat)$
⟨*proof*⟩

lemma *iless-not-refl*: $\neg\ n < (n::inat)$
⟨*proof*⟩

lemma *iless-trans*: $i < j \Longrightarrow j < k \Longrightarrow i < (k::inat)$
⟨*proof*⟩

lemma *iless-not-sym*: $n < m \Longrightarrow \neg\ m < (n::inat)$
⟨*proof*⟩

lemma *Fin-iless-mono* [*simp*]: $(Fin\ n < Fin\ m) = (n < m)$
⟨*proof*⟩

lemma *Fin-iless-Infty* [*simp*]: $Fin\ n < \infty$
⟨*proof*⟩

lemma *Infty-eq* [*simp*]: $(n < \infty) = (n \neq \infty)$
⟨*proof*⟩

lemma *i0-eq* [*simp*]: $((0::inat) < n) = (n \neq 0)$
⟨*proof*⟩

lemma *i0-iless-iSuc* [*simp*]: $0 < iSuc\ n$
⟨*proof*⟩

lemma *not-ilessi0* [*simp*]: $\neg\ n < (0::inat)$
⟨*proof*⟩

lemma *Fin-iless*: $n < Fin\ m \Longrightarrow \exists k.\ n = Fin\ k$
⟨*proof*⟩

lemma *iSuc-mono* [*simp*]: $(iSuc\ n < iSuc\ m) = (n < m)$
⟨*proof*⟩

lemma *ile-def2*: $(m \leq n) = (m < n \vee m = (n::inat))$
⟨*proof*⟩

lemma *ile-refl* [*simp*]: $n \leq (n::inat)$
⟨*proof*⟩

lemma *ile-trans*: $i \leq j \Longrightarrow j \leq k \Longrightarrow i \leq (k::inat)$
⟨*proof*⟩

lemma *ile-iless-trans*: $i \leq j \Longrightarrow j < k \Longrightarrow i < (k::inat)$
⟨*proof*⟩

lemma *iless-ile-trans*: $i < j \Longrightarrow j \leq k \Longrightarrow i < (k::inat)$
⟨*proof*⟩

lemma *Infty-ub* [*simp*]: $n \leq \infty$
⟨*proof*⟩

lemma *i0-lb* [*simp*]: $(0::inat) \leq n$
⟨*proof*⟩

lemma *Infty-ileE* [*elim!*]: $\infty \leq Fin\ m \Longrightarrow R$
⟨*proof*⟩

lemma *Fin-ile-mono* [*simp*]: $(Fin\ n \leq Fin\ m) = (n \leq m)$
⟨*proof*⟩

lemma *ilessI1*: $n \leq m \Longrightarrow n \neq m \Longrightarrow n < (m::inat)$
⟨*proof*⟩

lemma *ileI1*: $m < n \Longrightarrow iSuc\ m \leq n$
⟨*proof*⟩

lemma *Suc-ile-eq*: $(Fin\ (Suc\ m) \leq n) = (Fin\ m < n)$
⟨*proof*⟩

lemma *iSuc-ile-mono* [*simp*]: $(iSuc\ n \leq iSuc\ m) = (n \leq m)$
⟨*proof*⟩

lemma *iless-Suc-eq* [*simp*]: $(Fin\ m < iSuc\ n) = (Fin\ m \leq n)$
⟨*proof*⟩

lemma *not-iSuc-ilei0* [*simp*]: $\neg\ iSuc\ n \leq 0$
⟨*proof*⟩

lemma *ile-iSuc* [*simp*]: $n \leq iSuc\ n$
⟨*proof*⟩

lemma *Fin-ile*: $n \leq Fin\ m \Longrightarrow \exists k.\ n = Fin\ k$
⟨*proof*⟩

lemma *chain-incr*: $\forall i.\ \exists j.\ Y\ i < Y\ j \Longrightarrow \exists j.\ Fin\ k < Y\ j$
⟨*proof*⟩

40.4 Well-ordering

lemma *less-FinE*:
$[\!|\ n < Fin\ m;\ !!k.\ n = Fin\ k \Longrightarrow k < m \Longrightarrow P\ |\!] \Longrightarrow P$
⟨*proof*⟩

lemma *less-InftyE*:
$[\!|\ n < Infty;\ !!k.\ n = Fin\ k \Longrightarrow P\ |\!] \Longrightarrow P$
⟨*proof*⟩

lemma *inat-less-induct*:
assumes *prem*: $!!n.\ \forall m::inat.\ m < n \longrightarrow P\ m \Longrightarrow P\ n$ **shows** $P\ n$
⟨*proof*⟩

instance *inat* :: *wellorder*
⟨*proof*⟩

end

41 Nested-Environment: Nested environments

theory *Nested-Environment*
imports *List*
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.

41.1 The lookup operation

Lookup in nested environments works by following a given path of index elements, leading to an optional result (a terminal value or nested environment). A *defined position* within a nested environment is one where *lookup* at its path does not yield *None*.

consts
  $lookup :: ('a, 'b, 'c)\ env \Rightarrow 'c\ list \Rightarrow ('a, 'b, 'c)\ env\ option$
  $lookup\text{-}option :: ('a, 'b, 'c)\ env\ option \Rightarrow 'c\ list \Rightarrow ('a, 'b, 'c)\ env\ option$

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 ∃ *b′ es′ env′*.
    lookup env xs = *Some* (*Env b′ es′*) ∧
    es′ y = *Some env′* ∧
    lookup env′ ys = *Some e*
  ⟨*proof*⟩

41.2 The update operation

Update at a certain position in a nested environment may either delete an existing entry, or overwrite an existing one. Note that update at undefined positions is simple absorbed, i.e. the environment is left unchanged.

consts
  update :: *'c list* => (*'a*, *'b*, *'c*) *env option*
    => (*'a*, *'b*, *'c*) *env* => (*'a*, *'b*, *'c*) *env*
  update-option :: *'c list* => (*'a*, *'b*, *'c*) *env option*
    => (*'a*, *'b*, *'c*) *env option* => (*'a*, *'b*, *'c*) *env option*

primrec (*update*)
  update xs opt (*Val a*) =
    (*if xs* = [] *then* (*case opt of None* => *Val a* | *Some e* => *e*)
    else Val a)
  update xs opt (*Env b es*) =
    (*case xs of*
      [] => (*case opt of None* => *Env b es* | *Some e* => *e*)
    | *y* # *ys* => *Env b* (*es* (*y* := *update-option ys opt* (*es y*))))
  update-option xs opt None =
    (*if xs* = [] *then opt else None*)
  update-option xs opt (*Some e*) =
    (*if xs* = [] *then opt else Some* (*update xs opt e*))

hide *const update-option*

The characteristic cases of *update* are expressed by the following equalities.

theorem *update-nil-none*: *update* [] *None env* = *env*
  ⟨*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*⟩

Equality of environments for code generation

lemma [*code func*, *code func del*]:
fixes *e1 e2* :: *('b::eq, 'a::eq, 'c::eq) env*
shows *eq-class.eq e1 e2 ⟷ eq-class.eq e1 e2* ⟨*proof*⟩

lemma *eq-env-code* [*code func*]:
fixes *x y* :: *'a::eq*
and *f g* :: *'c::{eq, finite} ⇒ ('b::eq, 'a, 'c) env option*
shows *eq-class.eq (Env x f) (Env y g) ⟷*
eq-class.eq x y ∧ (∀ z∈UNIV. case f z
of None ⇒ (case g z
of None ⇒ True | Some - ⇒ False)
| Some a ⇒ (case g z
*of None ⇒ False | Some b ⇒ eq-class.eq a b)) (***is** *?env)*
and *eq-class.eq (Val a) (Val b) ⟷ eq-class.eq a b*
and *eq-class.eq (Val a) (Env y g) ⟷ False*
and *eq-class.eq (Env x f) (Val b) ⟷ False*
⟨*proof*⟩

end

42 Numeral-Type: Numeral Syntax for Types

theory *Numeral-Type*

imports *ATP-Linkup*
begin

42.1 Preliminary lemmas

lemma *inj-Inl* [*simp*]: *inj-on Inl A*
⟨*proof*⟩

lemma *inj-Inr* [*simp*]: *inj-on Inr A*
⟨*proof*⟩

lemma *inj-Some* [*simp*]: *inj-on Some A*
⟨*proof*⟩

lemma *card-Plus*:
[| *finite A*; *finite B* |] ==> *card* (*A* <+> *B*) = *card A* + *card B*
⟨*proof*⟩

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

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

42.2 Cardinalities of types

syntax -*type-card* :: *type* => *nat* ((*1CARD*/(*1'*(-')))))

translations *CARD*(*t*) => *CONST card* (*UNIV* :: *t set*)

⟨*ML*⟩

lemma *card-unit*: *CARD*(*unit*) = *1*
⟨*proof*⟩

lemma *card-bool*: *CARD*(*bool*) = *2*
⟨*proof*⟩

lemma *card-prod*: *CARD*('*a*::*finite* × '*b*::*finite*) = *CARD*('*a*) * *CARD*('*b*)
⟨*proof*⟩

lemma *card-sum*: *CARD*('*a*::*finite* + '*b*::*finite*) = *CARD*('*a*) + *CARD*('*b*)
⟨*proof*⟩

lemma *card-option*: *CARD*('*a*::*finite option*) = *Suc CARD*('*a*)
⟨*proof*⟩

lemma *card-set*: *CARD*('*a*::*finite set*) = *2* ^ *CARD*('*a*)
⟨*proof*⟩

42.3 Numeral Types

typedef (**open**) *num0* = *UNIV* :: *nat set* ⟨*proof*⟩
typedef (**open**) *num1* = *UNIV* :: *unit set* ⟨*proof*⟩
typedef (**open**) *'a bit0* = *UNIV* :: (*bool* ∗ *'a*) *set* ⟨*proof*⟩
typedef (**open**) *'a bit1* = *UNIV* :: (*bool* ∗ *'a*) *option set* ⟨*proof*⟩

instance *num1* :: *finite*
⟨*proof*⟩

instance *bit0* :: (*finite*) *finite*
⟨*proof*⟩

instance *bit1* :: (*finite*) *finite*
⟨*proof*⟩

lemma *card-num1*: *CARD*(*num1*) = *1*
⟨*proof*⟩

lemma *card-bit0*: *CARD*(*'a*::*finite bit0*) = *2* ∗ *CARD*(*'a*)
⟨*proof*⟩

lemma *card-bit1*: *CARD*(*'a*::*finite bit1*) = *Suc* (*2* ∗ *CARD*(*'a*))
⟨*proof*⟩

lemma *card-num0*: *CARD* (*num0*) = *0*
⟨*proof*⟩

lemmas *card-univ-simps* [*simp*] =
card-unit
card-bool
card-prod
card-sum
card-option
card-set
card-num1
card-bit0
card-bit1
card-num0

42.4 Syntax

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

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

⟨*ML*⟩

42.5 Classes with at least 1 and 2

Class finite already captures "at least 1"

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

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

Class for cardinality "at least 2"

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

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

instance *bit0* :: (*finite*) *card2*
⟨*proof*⟩

instance *bit1* :: (*finite*) *card2*
⟨*proof*⟩

42.6 Examples

lemma *CARD(0) = 0* ⟨*proof*⟩
lemma *CARD(17) = 17* ⟨*proof*⟩

end

43 Option-ord: Canonical order on option type

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

instantiation *option* :: (*order*) *order*
begin

definition *less-eq-option* **where**
[*code func del*]: $x \leq y \longleftrightarrow$ (*case x of None* ⇒ *True* | *Some x* ⇒ (*case y of None* ⇒ *False* | *Some y* ⇒ $x \leq y$))

definition *less-option* **where**
[*code func del*]: $x < y \longleftrightarrow$ (*case y of None* $\Rightarrow$ *False* | *Some y* $\Rightarrow$ (*case x of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ $x < y$))

lemma *less-eq-option-None* [*simp*]: *None* $\leq x$
⟨*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* :: (*linorder*) *linorder*
⟨*proof*⟩

end

44 Order-Relation: Orders as Relations

theory *Order-Relation*
imports *ATP-Linkup Hilbert-Choice*

begin

This prelude could be moved to theory Relation:

definition *irrefl r* ≡ ∀ *x*. (*x*,*x*) ∉ *r*

definition *total-on A r* ≡ ∀ *x*∈*A*.∀ *y*∈*A*. *x*≠*y* ⟶ (*x*,*y*)∈*r* ∨ (*y*,*x*)∈*r*

abbreviation *total* ≡ *total-on UNIV*

lemma *total-on-empty*[*simp*]: *total-on* {} *r*
⟨*proof*⟩

lemma *refl-on-converse*[*simp*]: *refl A* (*r*^−*1*) = *refl A r*
⟨*proof*⟩

lemma *total-on-converse*[*simp*]: *total-on A* (*r*^−*1*) = *total-on A r*
⟨*proof*⟩

lemma *irrefl-diff-Id*[*simp*]: *irrefl*(*r*−*Id*)
⟨*proof*⟩

declare [[*simp-depth-limit* = *2*]]
lemma *trans-diff-Id*: *trans r* ⟹ *antisym r* ⟹ *trans* (*r*−*Id*)
⟨*proof*⟩
declare [[*simp-depth-limit* = *50*]]

lemma *total-on-diff-Id*[*simp*]: *total-on A* (*r*−*Id*) = *total-on A r*
⟨*proof*⟩

44.1 Orders on a set

definition *preorder-on A r* ≡ *refl A r* ∧ *trans r*

definition *partial-order-on A r* ≡ *preorder-on A r* ∧ *antisym r*

definition *linear-order-on A r* ≡ *partial-order-on A r* ∧ *total-on A r*

definition *strict-linear-order-on A r* ≡ *trans r* ∧ *irrefl r* ∧ *total-on A r*

definition *well-order-on A r* ≡ *linear-order-on A r* ∧ *wf*(*r* − *Id*)

lemmas *order-on-defs* =
preorder-on-def partial-order-on-def linear-order-on-def
strict-linear-order-on-def well-order-on-def

lemma *preorder-on-empty*[*simp*]: *preorder-on* {} {}
⟨*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*ˆ−*1*) = *preorder-on A r*
⟨*proof*⟩

lemma *partial-order-on-converse*[*simp*]:
partial-order-on A (*r*ˆ−*1*) = *partial-order-on A r*
⟨*proof*⟩

lemma *linear-order-on-converse*[*simp*]:
linear-order-on A (*r*ˆ−*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*⟩

44.2 Orders on the field

abbreviation *Refl r* ≡ *refl* (*Field r*) *r*

abbreviation *Preorder r* ≡ *preorder-on* (*Field r*) *r*

abbreviation *Partial-order r* ≡ *partial-order-on* (*Field r*) *r*

abbreviation *Total r* ≡ *total-on* (*Field r*) *r*

abbreviation *Linear-order r* ≡ *linear-order-on* (*Field r*) *r*

abbreviation *Well-order r* ≡ *well-order-on* (*Field r*) *r*

lemma *subset-Image-Image-iff*:
⟦ *Preorder r*; *A* ⊆ *Field r*; *B* ⊆ *Field r*⟧ ⟹
r `` *A* ⊆ *r* `` *B* ⟷ (∀ *a*∈*A*.∃ *b*∈*B*. (*b*,*a*):*r*)
⟨*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*⟩

44.3 Orders on a type

abbreviation *strict-linear-order* ≡ *strict-linear-order-on UNIV*

abbreviation *linear-order* ≡ *linear-order-on UNIV*

abbreviation *well-order r* ≡ *well-order-on UNIV*

end

45 Permutation: Permutations

theory *Permutation*
imports *Multiset*
begin

inductive
perm :: *'a list* => *'a list* => *bool* (- <~~> - [*50*, *50*] *50*)
where
Nil [*intro*!]: [] <~~> []
| *swap* [*intro*!]: *y* # *x* # *l* <~~> *x* # *y* # *l*
| *Cons* [*intro*!]: *xs* <~~> *ys* ==> *z* # *xs* <~~> *z* # *ys*
| *trans* [*intro*]: *xs* <~~> *ys* ==> *ys* <~~> *zs* ==> *xs* <~~> *zs*

lemma *perm-refl* [*iff*]: *l* <~~> *l*
⟨*proof*⟩

45.1 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*⟩

45.2 Ways of making new permutations

We can insert the head anywhere in the list.

lemma *perm-append-Cons*: *a* # *xs* @ *ys* <~~> *xs* @ *a* # *ys*
⟨*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*⟩

45.3 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*⟩

45.4 Removing elements

consts
remove :: *'a* => *'a list* => *'a list*
primrec
remove x [] = []
remove x (*y* # *ys*) = (*if x* = *y then ys else y* # *remove x ys*)

lemma *perm-remove*: $x \in set\ ys \Longrightarrow ys <\sim\sim> x \# remove\ x\ ys$
⟨*proof*⟩

lemma *remove-commute*: $remove\ x\ (remove\ y\ l) = remove\ y\ (remove\ x\ l)$
⟨*proof*⟩

lemma *multiset-of-remove* [*simp*]:
$multiset\text{-}of\ (remove\ a\ x) = multiset\text{-}of\ x - \{\#a\#\}$
⟨*proof*⟩

Congruence rule

lemma *perm-remove-perm*: $xs <\sim\sim> ys \Longrightarrow remove\ z\ xs <\sim\sim> remove\ z\ ys$
⟨*proof*⟩

lemma *remove-hd* [*simp*]: $remove\ z\ (z \# xs) = xs$
⟨*proof*⟩

lemma *cons-perm-imp-perm*: $z \# xs <\sim\sim> z \# ys \Longrightarrow xs <\sim\sim> ys$
⟨*proof*⟩

lemma *cons-perm-eq* [*iff*]: $(z\#xs <\sim\sim> z\#ys) = (xs <\sim\sim> ys)$
⟨*proof*⟩

lemma *append-perm-imp-perm*: $zs @ xs <\sim\sim> zs @ ys \Longrightarrow xs <\sim\sim> ys$
⟨*proof*⟩

lemma *perm-append1-eq* [*iff*]: $(zs @ xs <\sim\sim> zs @ ys) = (xs <\sim\sim> ys)$
⟨*proof*⟩

lemma *perm-append2-eq* [*iff*]: $(xs @ zs <\sim\sim> ys @ zs) = (xs <\sim\sim> ys)$
⟨*proof*⟩

lemma *multiset-of-eq-perm*: $(multiset\text{-}of\ xs = multiset\text{-}of\ ys) = (xs <\sim\sim> ys)$
⟨*proof*⟩

lemma *multiset-of-le-perm-append*:
$(multiset\text{-}of\ xs \leq\# multiset\text{-}of\ ys) = (\exists zs.\ xs @ zs <\sim\sim> ys)$
⟨*proof*⟩

lemma *perm-set-eq*: $xs <\sim\sim> ys \Longrightarrow set\ xs = set\ ys$
⟨*proof*⟩

lemma *perm-distinct-iff*: $xs <\sim\sim> ys \Longrightarrow distinct\ xs = distinct\ ys$
⟨*proof*⟩

lemma *eq-set-perm-remdups*: $set\ xs = set\ ys \Longrightarrow remdups\ xs <\sim\sim> remdups\ ys$
⟨*proof*⟩

lemma *perm-remdups-iff-eq-set*: $remdups\ x <\sim\sim> remdups\ y = (set\ x = set\ y)$

⟨*proof*⟩

end

46 Primes: Primality on nat

theory *Primes*
imports *GCD Parity*
begin

definition
coprime :: *nat* => *nat* => *bool* **where**
coprime m n ⟷ (*gcd* (*m*, *n*) = *1*)

definition
prime :: *nat* ⇒ *bool* **where**
prime p ⟷ (*1* < *p* ∧ (∀ *m*. *m dvd p* −−> *m* = *1* ∨ *m* = *p*))

lemma *two-is-prime*: *prime 2*
⟨*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-le*: *m dvd n* ==> *m* <= *n* ∨ *n* = (*0*::*nat*) ⟨*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 $\Longrightarrow$ (*x*::*nat*) *dvd* (*y*^(*Suc n*)) ⟨*proof*⟩

The Bezout theorem is a bit ugly for N; it'd be easier for Z

lemma *ind-euclid*:
assumes *c*: $\forall a\ b.\ P\ (a::nat)\ b \longleftrightarrow P\ b\ a$ **and** *z*: $\forall a.\ P\ a\ 0$
and *add*: $\forall a\ b.\ P\ a\ b \longrightarrow P\ a\ (a + b)$
shows *P a b*
⟨*proof*⟩

lemma *bezout-lemma*:
assumes *ex*: $\exists (d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (a * x = b * y + d \vee b * x = a * y + d)$
shows $\exists d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ a + b \wedge (a * x = (a + b) * y + d \vee (a + b) * x = a * y + d)$
⟨*proof*⟩

lemma *bezout-add*: $\exists (d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (a * x = b * y + d \vee b * x = a * y + d)$
⟨*proof*⟩

lemma *bezout*: $\exists (d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (a * x - b * y = d \vee b * x - a * y = d)$
⟨*proof*⟩

We can get a stronger version with a nonzeroness assumption.

lemma *bezout-add-strong*: **assumes** *nz*: $a \neq (0::nat)$
shows $\exists d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge a * x = b * y + d$
⟨*proof*⟩

Greatest common divisor.

lemma *gcd-unique*: *d dvd a*$\wedge$*d dvd b* $\wedge$ ($\forall e.\ e\ dvd\ a \wedge e\ dvd\ b \longrightarrow e\ dvd\ d$) $\longleftrightarrow$ *d* = *gcd*(*a*,*b*)
⟨*proof*⟩

lemma *gcd-eq*: **assumes** *H*: $\forall d.\ d\ dvd\ x \wedge d\ dvd\ y \longleftrightarrow d\ dvd\ u \wedge d\ dvd\ v$
shows *gcd* (*x*,*y*) = *gcd*(*u*,*v*)
⟨*proof*⟩

lemma *bezout-gcd*: $\exists x\ y.\ a * x - b * y = gcd(a,b) \vee b * x - a * y = gcd(a,b)$
⟨*proof*⟩

lemma *bezout-gcd-strong*: **assumes** *a*: $a \neq 0$
shows $\exists x\ y.\ a * x = b * y + gcd(a,b)$
⟨*proof*⟩

lemma *gcd-mult-distrib*: $gcd(a * c,\ b * c) = c * gcd(a,b)$
⟨*proof*⟩

lemma *gcd-bezout*: $(\exists x\ y.\ a * x - b * y = d \vee b * x - a * y = d) \longleftrightarrow gcd(a,b)\ dvd\ d$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *gcd-bezout-sum*: **assumes** H:$a * x + b * y = d$ **shows** $gcd(a,b)\ dvd\ d$
⟨*proof*⟩

lemma *gcd-mult'*: $gcd(b,a * b) = b$
⟨*proof*⟩

lemma *gcd-add*: $gcd(a + b,b) = gcd(a,b)\ gcd(b + a,b) = gcd(a,b)\ gcd(a,a + b) = gcd(a,b)\ gcd(a,b + a) = gcd(a,b)$
⟨*proof*⟩

lemma *gcd-sub*: $b <= a ==> gcd(a - b,b) = gcd(a,b)\ a <= b ==> gcd(a,b - a) = gcd(a,b)$
⟨*proof*⟩

Coprimality

lemma *coprime*: $coprime\ a\ b \longleftrightarrow (\forall d.\ d\ dvd\ a \wedge d\ dvd\ b \longleftrightarrow d = 1)$
⟨*proof*⟩
lemma *coprime-commute*: $coprime\ a\ b \longleftrightarrow coprime\ b\ a$ ⟨*proof*⟩

lemma *coprime-bezout*: $coprime\ a\ b \longleftrightarrow (\exists x\ y.\ a * x - b * y = 1 \vee b * x - a * y = 1)$
⟨*proof*⟩

lemma *coprime-divprod*: $d\ dvd\ a * b \implies coprime\ d\ a \implies d\ dvd\ b$
⟨*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*: $gcd(a,b) \neq 0$ **and** *a*: $a = a' * gcd(a,b)$ **and** *b*: $b = b' * gcd(a,b)$
shows $coprime\ a'\ b'$
⟨*proof*⟩
lemma *coprime-0*: $coprime\ d\ 0 \longleftrightarrow d = 1$ ⟨*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*:

prime $(p*q) \Longrightarrow p = 1 \vee q = 1$ ⟨*proof*⟩

lemma *prime-exp*: *prime* $(p\hat{}n) \longleftrightarrow$ *prime* $p \wedge n = 1$
⟨*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

47 Quicksort: Quicksort

theory *Quicksort*
imports *Multiset*
begin

context *linorder*
begin

function *quicksort* :: *'a list* ⇒ *'a list* **where**
quicksort [] = [] |
quicksort $(x\#xs)$ = *quicksort*([$y \leftarrow xs.\ \sim x \leq y$]) @ [*x*] @ *quicksort*([$y \leftarrow xs.\ x \leq y$])
⟨*proof*⟩

termination
⟨*proof*⟩

end
context *linorder*
begin

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

48 Quotient: Quotient types

theory *Quotient*
imports *ATP-Linkup Hilbert-Choice*
begin

We introduce the notion of quotient types over equivalence relations via type classes.

48.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations ∼ :: *'a* => *'a* => *bool*.

class *eqv* = *type* +
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* ∼ *x*} | *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* => *'a quot* (⌊-⌋) **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*⟩

48.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $(\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim b)$
⟨*proof*⟩

48.3 Picking representing elements

definition
pick :: *'a*::*equiv quot* => *'a* **where**
$pick\ A = (SOME\ a.\ A = \lfloor a \rfloor)$

theorem *pick-equiv* [*intro*]: $pick\ \lfloor a \rfloor \sim a$
⟨*proof*⟩

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 !!$x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor ==> \lfloor y \rfloor = \lfloor y' \rfloor ==> g\ x\ y = g\ x'\ y'$
shows $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
⟨*proof*⟩

theorem *quot-function'*:
$(!!X\ Y.\ f\ X\ Y == g\ (pick\ X)\ (pick\ Y)) ==>$
$(!!x\ x'\ y\ y'.\ x \sim x' ==> y \sim y' ==> g\ x\ y = g\ x'\ y') ==>$
$f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
⟨*proof*⟩

end

49 Ramsey: Ramsey's Theorem

theory *Ramsey*
imports *ATP-Linkup Infinite-Set*
begin

49.1 Preliminaries

49.1.1 "Axiom" of Dependent Choice

consts *choice* :: $('a => bool) => ('a * 'a)\ set => nat => 'a$
— 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*: !!$x.\ P\ x ==> \exists y.\ P\ y\ \&\ (x,y) \in r$
shows $P\ (choice\ P\ r\ n)$
⟨*proof*⟩

lemma *dependent-choice*:
assumes *trans*: *trans r*
and *P0*: *P x0*
and *Pstep*: !!$x.\ P\ x ==> \exists y.\ P\ y\ \&\ (x,y) \in r$
obtains $f :: nat => 'a$ **where**
!!$n.\ P\ (f\ n)$ **and** !!$n\ m.\ n < m ==> (f\ n,\ f\ m) \in r$
⟨*proof*⟩

49.1.2 Partitions of a Set

definition
part :: *nat* => *nat* => *'a set* => (*'a set* => *nat*) => *bool*
— the function f partitions the r-subsets of the typically infinite set Y into s distinct categories.
where
part r s Y f = (∀ *X*. *X* ⊆ *Y* & *finite X* & *card X* = *r* --> *f X* < *s*)

For induction, we decrease the value of r in partitions.

lemma *part-Suc-imp-part*:
[| *infinite Y*; *part* (*Suc r*) *s Y f*; *y* ∈ *Y* |]
==> *part r s* (*Y* − {*y*}) (%*u*. *f* (*insert y u*))
⟨*proof*⟩

lemma *part-subset*: *part r s YY f* ==> *Y* ⊆ *YY* ==> *part r s Y f*
⟨*proof*⟩

49.2 Ramsey's Theorem: Infinitary Version

lemma *Ramsey-induction*:
fixes *s* **and** *r*::*nat*
shows
!!(*YY*::*'a set*) (*f*::*'a set* => *nat*).
[|*infinite YY*; *part r s YY f*|]
==> ∃ *Y' t'*. *Y'* ⊆ *YY* & *infinite Y'* & *t'* < *s* &
(∀ *X*. *X* ⊆ *Y'* & *finite X* & *card X* = *r* --> *f X* = *t'*)
⟨*proof*⟩

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*⟩

49.3 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [5].

definition
disj-wf :: $('a * 'a)set \Rightarrow bool$
where
$disj\text{-}wf\ r = (\exists\, T.\ \exists\, n{::}nat.\ (\forall\, i{<}n.\ wf(T\ i))\ \&\ r = (\bigcup i{<}n.\ T\ i))$

definition
transition-idx :: $[nat \Rightarrow 'a,\ nat \Rightarrow ('a{*}'a)set,\ nat\ set] \Rightarrow nat$
where
$transition\text{-}idx\ s\ T\ A =$
$(LEAST\ k.\ \exists\, i\ j.\ A = \{i,j\}\ \&\ i{<}j\ \&\ (s\ j,\ s\ i) \in T\ k)$

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

50 RBT: Red-Black Trees

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨p

This theory defines purely functional red-black trees which can be used as an efficient representation of finite maps.

50.1 Data type and invariant

The type $('k, 'v)$ *rbt* denotes red-black trees with keys of type $'k$ and values of type $'v$. To function properly, the key type must belong to the *linorder*

class.

A value t of this type is a valid red-black tree if it satisfies the invariant *isrbt t*. This theory provides lemmas to prove that the invariant is satisfied throughout the computation.

The interpretation function *RBT.map-of* returns the partial map represented by a red-black tree:

$RBT.map\text{-}of :: ('a, 'b)\ rbt \Rightarrow 'a \rightharpoonup 'b$

This function should be used for reasoning about the semantics of the RBT operations. Furthermore, it implements the lookup functionality for the data structure: It is executable and the lookup is performed in $O(\log n)$.

50.2 Operations

Currently, the following operations are supported:

$Empty :: ('a, 'b)\ rbt$

Returns the empty tree. $O(1)$

$insrt :: 'a \Rightarrow 'b \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

Updates the map at a given position. $O(\log n)$

$RBT.delete :: 'a \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

Deletes a map entry at a given position. $O(\log n)$

$union :: ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

Forms the union of two trees, preferring entries from the first one.

$RBT.map :: ('a \Rightarrow 'b) \Rightarrow ('c, 'a)\ rbt \Rightarrow ('c, 'b)\ rbt$

Maps a function over the values of a map. $O(n)$

50.3 Invariant preservation

$isrbt\ Empty$	(*Empty-isrbt*)
$isrbt\ ?t \Longrightarrow isrbt\ (insrt\ ?k\ ?v\ ?t)$	(*insrt-isrbt*)
$isrbt\ ?t \Longrightarrow isrbt\ (RBT.delete\ ?k\ ?t)$	(*delete-isrbt*)
$isrbt\ ?lt \Longrightarrow isrbt\ (union\ ?lt\ ?rt)$	(*union-isrbt*)
$isrbt\ (RBT.map\ ?f\ ?t) = isrbt\ ?t$	(*map-isrbt*)

50.4 Map Semantics

map-of-Empty

$RBT.map\text{-}of\ Empty = empty$

map-of-insert

$isrbt\ ?t \Longrightarrow RBT.map\text{-}of\ (insrt\ ?k\ ?v\ ?t) = RBT.map\text{-}of\ ?t(?k \mapsto ?v)$

map-of-delete

$isrbt\ ?t \Longrightarrow RBT.map\text{-}of\ (RBT.delete\ ?k\ ?t) = RBT.map\text{-}of\ ?t\restriction_{(-\ \{?k\})}$

map-of-union

$[\![isrbt\ ?s;\ st\ ?t]\!]$
$\Longrightarrow RBT.map\text{-}of\ (union\ ?s\ ?t) = RBT.map\text{-}of\ ?s\ ++\ RBT.map\text{-}of\ ?t$

map-of-map

$RBT.map\text{-}of\ (RBT.map\ ?f\ ?t) = option\text{-}map\ ?f \circ RBT.map\text{-}of\ ?t$

end

51 State-Monad: Combinators syntax for generic, open state monads (single threaded monads)

theory *State-Monad*
imports *List*
begin

51.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is transformed single-threaded).

To enter from the Haskell world, http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

51.2 State transformations and combinators

We classify functions operating on states into two categories:

transformations with type signature $\sigma \Rightarrow \sigma'$, transforming a state.

"yielding" transformations with type signature $\sigma \Rightarrow \alpha \times \sigma'$, "yielding" a side result while transforming a state.

queries with type signature $\sigma \Rightarrow \alpha$, computing a result dependent on a state.

By convention we write σ for types representing states and α, β, γ, ... for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type σ are used in a single-threaded way: after application of a transformation on a value of type σ, the former value should not be used again. To achieve this, we use a set of monad combinators:

notation *fcomp* (**infixl** $>>$ *60*)
notation (*xsymbols*) *fcomp* (**infixl** $\gg$ *60*)
notation *scomp* (**infixl** $>>=$ *60*)
notation (*xsymbols*) *scomp* (**infixl** $\gg=$ *60*)

abbreviation (*input*)
  return $\equiv$ *Pair*

definition
  run :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ **where**
  run $f = f$

⟨*ML*⟩

Given two transformations f and g, they may be directly composed using the *op* $\gg$ combinator, forming a forward composition: $(f \gg g)\ s = f\ (g\ s)$.

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the *op* $\gg=$ combinator: $(f \gg= (\lambda x.\ g))\ s = (let\ (x,\ s') = f\ s\ in\ g\ s')$.

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *Pair* for this purpose.

The *run* ist just a marker.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.
- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units ("open monad").
- The type of states may change due to a transformation.

51.3 Obsolete runs

run is just a doodle and should not occur nested:

lemma *run-simp* [*simp*]:
$\bigwedge f.\ run\ (run\ f) = run\ f$
$\bigwedge f\ g.\ run\ f \gg\!= g = f \gg\!= g$
$\bigwedge f\ g.\ run\ f \gg g = f \gg g$
$\bigwedge f\ g.\ f \gg\!= (\lambda x.\ run\ g) = f \gg\!= (\lambda x.\ g)$
$\bigwedge f\ g.\ f \gg run\ g = f \gg g$
$\bigwedge f.\ f = run\ f \longleftrightarrow True$
$\bigwedge f.\ run\ f = f \longleftrightarrow True$
⟨*proof*⟩

51.4 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

lemma
return-scomp [*simp*]: $return\ x \gg\!= f = f\ x$
⟨*proof*⟩

lemma
scomp-return [*simp*]: $x \gg\!= return = x$
⟨*proof*⟩

lemma
id-fcomp [*simp*]: $id \gg f = f$
⟨*proof*⟩

lemma
fcomp-id [*simp*]: $f \gg id = f$
⟨*proof*⟩

lemma
scomp-scomp [*simp*]: $(f \gg\!= g) \gg\!= h = f \gg\!= (\lambda x.\ g\ x \gg\!= h)$
⟨*proof*⟩

lemma
scomp-fcomp [*simp*]: $(f \gg\!= g) \gg h = f \gg\!= (\lambda x.\ g\ x \gg h)$
⟨*proof*⟩

lemma
fcomp-scomp [*simp*]: $(f \gg g) \gg\!= h = f \gg (g \gg\!= h)$
⟨*proof*⟩

lemma
fcomp-fcomp [*simp*]: $(f \gg g) \gg h = f \gg (g \gg h)$
⟨*proof*⟩

lemmas *monad-simp* = *run-simp return-scomp scomp-return id-fcomp fcomp-id scomp-scomp scomp-fcomp fcomp-scomp fcomp-fcomp*

Evaluation of monadic expressions by force:

lemmas *monad-collapse* = *monad-simp o-apply o-assoc split-Pair split-comp scomp-def fcomp-def run-def*

51.5 ML abstract operations

⟨*ML*⟩

51.6 Syntax

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

nonterminals *do-expr*

syntax
-do :: *do-expr* ⇒ *'a*
(*do - done* [*12*] *12*)
-scomp :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*- <− -;// -* [*1000*, *13*, *12*] *12*)
-fcomp :: *'a* ⇒ *do-expr* ⇒ *do-expr*
(*-;// -* [*13*, *12*] *12*)
-let :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*let - = -;// -* [*1000*, *13*, *12*] *12*)
-nil :: *'a* ⇒ *do-expr*
(*-* [*12*] *12*)

syntax (*xsymbols*)
-scomp :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*- ← -;// -* [*1000*, *13*, *12*] *12*)

translations
-do f => *CONST run f*
-scomp x f g => $f \gg\!= (\lambda x.\ g)$
-fcomp f g => $f \gg g$

-let x t f => CONST Let t (λx. f)
-nil f => f

⟨*ML*⟩

51.7 Combinators

definition
lift :: ('*a* ⇒ '*b*) ⇒ '*a* ⇒ '*c* ⇒ '*b* × '*c* **where**
lift f x = *return* (*f x*)

primrec
list :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*a list* ⇒ '*b* ⇒ '*b* **where**
list f [] = *id*
| *list f* (*x*#*xs*) = (*do f x*; *list f xs done*)

primrec
list-yield :: ('*a* ⇒ '*b* ⇒ '*c* × '*b*) ⇒ '*a list* ⇒ '*b* ⇒ '*c list* × '*b* **where**
list-yield f [] = *return* []
| *list-yield f* (*x*#*xs*) = (*do y* ← *f x*; *ys* ← *list-yield f xs*; *return* (*y*#*ys*) *done*)

definition
collapse :: ('*a* ⇒ ('*a* ⇒ '*b* × '*a*) × '*a*) ⇒ '*a* ⇒ '*b* × '*a* **where**
collapse f = (*do g* ← *f*; *g done*)

combinator properties

lemma *list-append* [*simp*]:
list f (*xs* @ *ys*) = *list f xs* ≫ *list f ys*
⟨*proof*⟩

lemma *list-cong* [*fundef-cong*, *recdef-cong*]:
⟦ ⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *g x*; *xs* = *ys* ⟧
⟹ *list f xs* = *list g ys*
⟨*proof*⟩

lemma *list-yield-cong* [*fundef-cong*, *recdef-cong*]:
⟦ ⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *g x*; *xs* = *ys* ⟧
⟹ *list-yield f xs* = *list-yield g ys*
⟨*proof*⟩

For an example, see HOL/ex/Random.thy.

end

52 Univ-Poly: Univariate Polynomials

theory *Univ-Poly*
imports *Main*
begin

Application of polynomial as a function.

```
primrec (in semiring-0) poly :: 'a list => 'a  => 'a where
  poly-Nil:  poly [] x = 0
| poly-Cons: poly (h#t) x = h + x * poly t x
```

52.1 Arithmetic Operations on Polynomials

addition

```
primrec (in semiring-0) padd :: 'a list ⇒ 'a list ⇒ 'a list  (infixl +++ 65)
where
  padd-Nil:  [] +++ l2 = l2
| padd-Cons: (h#t) +++ l2 = (if l2 = [] then h#t
                          else (h + hd l2)#(t +++ tl l2))
```

Multiplication by a constant

```
primrec (in semiring-0) cmult :: 'a ⇒ 'a list ⇒ 'a list  (infixl %* 70) where
   cmult-Nil:  c %* [] = []
|  cmult-Cons: c %* (h#t) = (c * h)#(c %* t)
```

Multiplication by a polynomial

```
primrec (in semiring-0) pmult :: 'a list ⇒ 'a list ⇒ 'a list  (infixl *** 70)
where
   pmult-Nil:  [] *** l2 = []
|  pmult-Cons: (h#t) *** l2 = (if t = [] then h %* l2
                            else (h %* l2) +++ ((0) # (t *** l2)))
```

Repeated multiplication by a polynomial

```
primrec (in semiring-0) mulexp :: nat ⇒ 'a list ⇒ 'a  list ⇒ 'a list where
   mulexp-zero:  mulexp 0 p q = q
|  mulexp-Suc:   mulexp (Suc n) p q = p *** mulexp n p q
```

Exponential

```
primrec (in semiring-1) pexp :: 'a list ⇒ nat ⇒ 'a list  (infixl %^ 80) where
   pexp-0:   p %^ 0 = [1]
|  pexp-Suc: p %^ (Suc n) = p *** (p %^ n)
```

Quotient related value of dividing a polynomial by x + a

```
primrec (in field) pquot :: 'a list ⇒ 'a ⇒ 'a list where
   pquot-Nil:  pquot [] a= []
|  pquot-Cons: pquot (h#t) a = (if t = [] then [h]
                 else (inverse(a) * (h - hd( pquot t a)))#(pquot t a))
```

normalization of polynomials (remove extra 0 coeff)

```
primrec (in semiring-0) pnormalize :: 'a list ⇒ 'a list where
  pnormalize-Nil:  pnormalize [] = []
| pnormalize-Cons: pnormalize (h#p) = (if ( (pnormalize p) = [])
                                  then (if (h = 0) then [] else [h])
                                  else (h#(pnormalize p)))
```

definition (**in** *semiring-0*) *pnormal p* = ((*pnormalize p* = *p*) ∧ *p* ≠ []) **definition** (**in** *semiring-0*) *nonconstant p* = (*pnormal p* ∧ (∀ *x*. *p* ≠ [*x*]))

Other definitions

definition (**in** *ring-1*)
poly-minus :: *'a list* => *'a list* (−− - [*80*] *80*) **where**
−− *p* = (− *1*) %∗ *p*

definition (**in** *semiring-0*)
divides :: *'a list* ⇒ *'a list* ⇒ *bool* (**infixl** *divides 70*) **where**
p1 divides p2 = (∃ *q*. *poly p2* = *poly*(*p1* ∗∗∗ *q*))

— order of a polynomial
definition (**in** *ring-1*) *order* :: *'a* => *'a list* => *nat* **where**
order a p = (*SOME n*. ([−*a*, *1*] %^ *n*) *divides p* &
~ (([−*a*, *1*] %^ (*Suc n*)) *divides p*))

— degree of a polynomial
definition (**in** *semiring-0*) *degree* :: *'a list* => *nat* **where**
degree p = *length* (*pnormalize p*) − *1*

— squarefree polynomials — NB with respect to real roots only.
definition (**in** *ring-1*)
rsquarefree :: *'a list* => *bool* **where**
rsquarefree p = (*poly p* ≠ *poly* [] &
(∀ *a*. (*order a p* = *0*) | (*order a p* = *1*)))

context *semiring-0*
begin

lemma *padd-Nil2*[*simp*]: *p* +++ [] = *p*
⟨*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*: $\forall b\ c.\ (a +++ b) +++ c = a +++ (b +++ c)$
⟨*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* $(p\ \%\hat{}\ n)\ x = (poly\ p\ x)\ \hat{}\ n$
⟨*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*⟩

52.2 Key Property: if *f a* = (*0*::'*a*) then *x* − *a* divides *p x*

lemma (**in** *comm-ring-1*) *lemma-poly-linear-rem*: ∀ *h*. ∃ *q r*. *h*#*t* = [*r*] +++ [−*a*, *1*] *** *q*
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-linear-rem*: ∃ *q r*. *h*#*t* = [*r*] +++ [−*a*, *1*] *** *q*
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-linear-divides*: (*poly p a* = *0*) = ((*p* = []) | (∃ *q*. *p* = [−*a*, *1*] *** *q*))
⟨*proof*⟩

lemma (**in** *semiring-0*) *lemma-poly-length-mult*[*simp*]: ∀ *h k a*. *length* (*k* %* *p* +++ (*h* # (*a* %* *p*))) = *Suc* (*length p*)
⟨*proof*⟩

lemma (**in** *semiring-0*) *lemma-poly-length-mult2*[*simp*]: ∀ *h k*. *length* (*k* %* *p* +++ (*h* # *p*)) = *Suc* (*length p*)
⟨*proof*⟩

lemma (**in** *ring-1*) *poly-length-mult*[*simp*]: *length*([−*a*,*1*] *** *q*) = *Suc* (*length q*)
⟨*proof*⟩

52.3 Polynomial length

lemma (**in** *semiring-0*) *poly-cmult-length*[*simp*]: *length* (*a* %* *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*] ***** *p*) = *Suc* (*length p*)
⟨*proof*⟩

lemma (**in** *idom*) *poly-mult-not-eq-poly-Nil*[*simp*]:
poly (*p *** q*) *x* ≠ *poly* [] *x* ⟷ *poly p x* ≠ *poly* [] *x* ∧ *poly q x* ≠ *poly* [] *x*
⟨*proof*⟩

lemma (**in** *idom*) *poly-mult-eq-zero-disj*: *poly* (*p *** q*) *x* = *0* ⟷ *poly p x* = *0* ∨ *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 *UNIV-nat-infinite*: ¬ *finite* (*UNIV* :: *nat set*)
⟨*proof*⟩

lemma (**in** *ring-char-0*) *UNIV-ring-char-0-infinte*:
¬ (*finite* (*UNIV*:: *'a set*))
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-roots-finite*: (*poly p* ≠ *poly* []) =
finite {*x*. *poly p x* = *0*}
⟨*proof*⟩

Entirety and Cancellation for polynomials

lemma (**in** *idom-char-0*) *poly-entire-lemma2*:
assumes *p0*: *poly p* ≠ *poly* [] **and** *q0*: *poly q* ≠ *poly* []
shows *poly* (*p****q*) ≠ *poly* []
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-entire*:
poly (*p* *** *q*) = *poly* [] ⟷ *poly p* = *poly* [] ∨ *poly q* = *poly* []
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-entire-neg*: (*poly* (*p* *** *q*) ≠ *poly* []) = ((*poly p* ≠ *poly* []) & (*poly q* ≠ *poly* []))
⟨*proof*⟩

lemma *fun-eq*: (*f* = *g*) = (∀ *x*. *f x* = *g x*)
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-add-minus-zero-iff*: (*poly* (*p* +++ −− *q*) = *poly* []) = (*poly p* = *poly q*)
⟨*proof*⟩

lemma (**in** *comm-ring-1*) *poly-add-minus-mult-eq*: *poly* (*p* *** *q* +++ −−(*p* *** *r*)) = *poly* (*p* *** (*q* +++ −− *r*))
⟨*proof*⟩

subclass (**in** *idom-char-0*) *comm-ring-1* ⟨*proof*⟩
lemma (**in** *idom-char-0*) *poly-mult-left-cancel*: (*poly* (*p* *** *q*) = *poly* (*p* *** *r*)) = (*poly p* = *poly* [] | *poly q* = *poly r*)
⟨*proof*⟩

lemma (**in** *recpower-idom*) *poly-exp-eq-zero*[*simp*]:
(*poly* (*p* %^ *n*) = *poly* []) = (*poly p* = *poly* [] & *n* ≠ *0*)
⟨*proof*⟩

lemma (**in** *semiring-1*) *one-neq-zero*[*simp*]: *1* ≠ *0* ⟨*proof*⟩
lemma (**in** *comm-ring-1*) *poly-prime-eq-zero*[*simp*]: *poly* [*a*,*1*] ≠ *poly* []
⟨*proof*⟩

lemma (**in** *recpower-idom*) *poly-exp-prime-eq-zero*: (*poly* ([*a*, *1*] %^ *n*) ≠ *poly* [])
⟨*proof*⟩

A more constructive notion of polynomials being trivial

lemma (**in** *idom-char-0*) *poly-zero-lemma'*: *poly* (*h* # *t*) = *poly* [] ==> *h* = *0* & *poly t* = *poly* []

⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-zero*: (*poly p* = *poly* []) = *list-all* (%*c*. *c* = *0*) *p*
⟨*proof*⟩

lemma (**in** *idom-char-0*) *poly-0*: *list-all* (λ*c*. *c* = *0*) *p* ⟹ *poly p x* = *0*
⟨*proof*⟩

Basics of divisibility.

lemma (**in** *idom*) *poly-primes*: ([*a*, *1*] *divides* (*p* *** *q*)) = ([*a*, *1*] *divides p* | [*a*, *1*] *divides q*)
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *poly-divides-refl*[*simp*]: *p divides p*
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *poly-divides-trans*: [| *p divides q*; *q divides r* |] ==> *p divides r*
⟨*proof*⟩

lemma (**in** *recpower-comm-semiring-1*) *poly-divides-exp*: *m* ≤ *n* ==> (*p* %^ *m*) *divides* (*p* %^ *n*)
⟨*proof*⟩

lemma (**in** *recpower-comm-semiring-1*) *poly-exp-divides*: [| (*p* %^ *n*) *divides q*; *m*≤*n* |] ==> (*p* %^ *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* ≠ *poly* []
shows ∃ *n q*. *p* = *mulexp n* [−*a*, *1*] *q* ∧ *poly q a* ≠ *0*
⟨*proof*⟩

lemma (**in** *recpower-comm-semiring-1*) *poly-mulexp*: *poly* (*mulexp n p q*) *x* = (*poly p x*) ^ *n* ∗ *poly q x*
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *divides-left-mult*:
assumes *d*:(*p*∗∗∗*q*) *divides r* **shows** *p divides r* ∧ *q divides r*
⟨*proof*⟩

lemma (**in** *recpower-semiring-1*)
zero-power-iff: *0* ^ *n* = (*if n* = *0 then 1 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*] %ˆ*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

53 While-Combinator: A general "while" combinator

theory *While-Combinator*
imports *Main*
begin

We define the while combinator as the "mother of all tail recursive functions".

function (*tailrec*) *while* :: ('*a* ⇒ *bool*) ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a*
where
while-unfold[*simp del*]: *while b c s* = (*if b s then while b c* (*c s*) *else s*)
⟨*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* (λN::*int set*. {*0*} ∪ {(*n* + *2*) *mod* *6* | *n*. *n* ∈ N})) =
P {*0*, *4*, *2*}
⟨*proof*⟩

end

54 Word: Binary Words

theory *Word*
imports *List*
begin

54.1 Auxilary Lemmas

lemma *max-le* [*intro*!]: [| x ≤ z; y ≤ z |] ==> *max* x y ≤ z
⟨*proof*⟩

lemma *max-mono*:
fixes x :: 'a::*linorder*
assumes *mf*: *mono* f
shows *max* (f x) (f y) ≤ 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*⟩

54.2 Bits

datatype *bit* =
Zero (**0**)
| *One* (**1**)

primrec
bitval :: *bit* => *nat*
where
bitval **0** = *0*
| *bitval* **1** = *1*

consts
bitnot :: *bit* => *bit*
bitand :: *bit* => *bit* => *bit* (**infixr** *bitand 35*)
bitor :: *bit* => *bit* => *bit* (**infixr** *bitor 30*)
bitxor :: *bit* => *bit* => *bit* (**infixr** *bitxor 30*)

notation (*xsymbols*)
bitnot ($\neg_b$ - [*40*] *40*) **and**
bitand (**infixr** $\wedge_b$ *35*) **and**
bitor (**infixr** $\vee_b$ *30*) **and**
bitxor (**infixr** $\oplus_b$ *30*)

notation (*HTML* **output**)
bitnot ($\neg_b$ - [*40*] *40*) **and**
bitand (**infixr** $\wedge_b$ *35*) **and**
bitor (**infixr** $\vee_b$ *30*) **and**
bitxor (**infixr** $\oplus_b$ *30*)

primrec
bitnot-zero: (*bitnot* **0**) = **1**
bitnot-one : (*bitnot* **1**) = **0**

primrec
bitand-zero: (**0** *bitand y*) = **0**
bitand-one: (**1** *bitand y*) = *y*

primrec
bitor-zero: (**0** *bitor y*) = *y*
bitor-one: (**1** *bitor y*) = **1**

primrec
bitxor-zero: (**0** *bitxor y*) = *y*
bitxor-one: (**1** *bitxor y*) = (*bitnot y*)

lemma *bitnot-bitnot* [*simp*]: (*bitnot* (*bitnot b*)) = *b*
⟨*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*⟩

54.3 Bit Vectors

First, a couple of theorems expressing case analysis and induction principles for bit vectors.

lemma *bit-list-cases*:
assumes *empty*: *w* = [] ==> *P w*
and *zero*: !!*bs*. *w* = **0** # *bs* ==> *P w*
and *one*: !!*bs*. *w* = **1** # *bs* ==> *P w*
shows *P w*
⟨*proof*⟩

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*⟩

54.4 Unsigned Arithmetic Operations

definition
bv-add :: [*bit list*, *bit list*] => *bit list* **where**
bv-add w1 w2 = *nat-to-bv* (*bv-to-nat w1* + *bv-to-nat w2*)

lemma *bv-add-type1* [*simp*]: *bv-add* (*norm-unsigned w1*) *w2* = *bv-add w1 w2*
⟨*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*⟩

54.5 Signed Vectors

consts
norm-signed :: *bit list* => *bit list*
primrec
norm-signed-Nil: *norm-signed* [] = []
norm-signed-Cons: *norm-signed* (*b*#*bs*) =
(*case b of*
0 => *if norm-unsigned bs* = [] *then* [] *else b*#*norm-unsigned bs*
| **1** => *b*#*rem-initial b bs*)

lemma *norm-signed0* [*simp*]: *norm-signed* [**0**] = []
⟨*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* ^ (*length w* − *1*)) ≤ *bv-to-int w*
⟨*proof*⟩

lemma *int-bv-int* [*simp*]: *int-to-bv* (*bv-to-int w*) = *norm-signed w*
⟨*proof*⟩

lemma *bv-int-bv* [*simp*]: *bv-to-int* (*int-to-bv i*) = *i*
⟨*proof*⟩

lemma *bv-msb-norm* [*simp*]: *bv-msb* (*norm-signed w*) = *bv-msb w*
⟨*proof*⟩

lemma *norm-signed-length*: *length* (*norm-signed w*) ≤ *length w*
⟨*proof*⟩

lemma *norm-signed-equal*: *length* (*norm-signed w*) = *length w* ==> *norm-signed w* = *w*
⟨*proof*⟩

lemma *bv-extend-norm-signed*: *bv-msb w* = *b* ==> *bv-extend* (*length w*) *b* (*norm-signed w*) = *w*
⟨*proof*⟩

lemma *bv-to-int-qinj*:
assumes *one*: *bv-to-int xs* = *bv-to-int ys*
and *len*: *length xs* = *length ys*

shows $xs = ys$
⟨*proof*⟩

lemma *int-to-bv-returntype* [*simp*]: *norm-signed* (*int-to-bv w*) = *int-to-bv w*
⟨*proof*⟩

lemma *bv-to-int-msb0*: $0 \leq$ *bv-to-int w1* ==> *bv-msb w1* = **0**
⟨*proof*⟩

lemma *bv-to-int-msb1*: *bv-to-int w1* < *0* ==> *bv-msb w1* = **1**
⟨*proof*⟩

lemma *bv-to-int-lower-limit-gt0*:
assumes *w0*: *0* < *bv-to-int w*
shows $2 \hat{} (length\ (norm\text{-}signed\ w) - 2) \leq$ *bv-to-int w*
⟨*proof*⟩

lemma *norm-signed-result*: *norm-signed w* = [] ∨ *norm-signed w* = [**1**] ∨ *bv-msb* (*norm-signed w*) ≠ *bv-msb* (*tl* (*norm-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*: $i \leq 2 \hat{} (k - 1) - 1$
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*: $2 \hat{} (k - 1) \leq i$
shows *k* < *length* (*int-to-bv i*)
⟨*proof*⟩

lemma *length-int-to-bv-upper-limit-lem1*:
assumes *w1*: *i* < *−1*

and *wk*: $-(2\ \hat{}\ (k-1)) \leq i$
shows *length* (*int-to-bv i*) $\leq k$
⟨*proof*⟩

lemma *length-int-to-bv-lower-limit-lem1*:
assumes *wk*: $i < -(2\ \hat{}\ (k-1))$
shows $k <$ *length* (*int-to-bv i*)
⟨*proof*⟩

54.6 Signed Arithmetic Operations

54.6.1 Conversion from unsigned to signed

definition
utos :: *bit list* => *bit list* **where**
utos w = *norm-signed* (**0** # *w*)

lemma *utos-type* [*simp*]: *utos* (*norm-unsigned w*) = *utos w*
⟨*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*⟩

54.6.2 Unary minus

definition
bv-uminus :: *bit list* => *bit list* **where**
bv-uminus w = *int-to-bv* (− *bv-to-int w*)

lemma *bv-uminus-type* [*simp*]: *bv-uminus* (*norm-signed w*) = *bv-uminus w*
⟨*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*⟩

54.7 Structural operations

definition
bv-select :: [*bit list*,*nat*] => *bit* **where**
bv-select w i = *w* ! (*length w* − *1* − *i*)

definition
bv-chop :: [*bit list*,*nat*] => *bit list* ∗ *bit list* **where**
bv-chop w i = (*let len* = *length w in* (*take* (*len* − *i*) *w*,*drop* (*len* − *i*) *w*))

definition
bv-slice :: [*bit list*,*nat*∗*nat*] => *bit list* **where**
bv-slice w = (λ(*b*,*e*). *fst* (*bv-chop* (*snd* (*bv-chop w* (*b*+*1*))) *e*))

lemma *bv-select-rev*:
assumes *notnull*: *n* < *length w*
shows *bv-select w n* = *rev w* ! *n*
⟨*proof*⟩

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**
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* ≤ *y* ; *y* ≤ *0* |] ==> *length-int y* ≤ *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*)

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

55 Zorn: Zorn's Lemma

theory *Zorn*
imports *Order-Relation*
begin

definition *chain-subset* :: *'a set set* ⇒ *bool* (*chain*$_{\subseteq}$) **where**
chain$_{\subseteq}$ *C* ≡ ∀ *A*∈*C*.∀ *B*∈*C*. *A* ⊆ *B* ∨ *B* ⊆ *A*

The lemma and section numbers refer to an unpublished article [1].

definition
chain :: *'a set set* => *'a set set set* **where**
chain S = {*F*. *F* ⊆ *S* & *chain*$_{\subseteq}$ *F*}

definition
super :: [*'a set set*,*'a set set*] => *'a set set set* **where**
super S c = {*d*. *d* ∈ *chain S* & *c* ⊂ *d*}

definition
maxchain :: *'a set set => 'a set set set* **where**
maxchain S = {*c*. *c* ∈ *chain S* & *super S c* = {}}

definition
succ :: [*'a set set*,*'a set set*] => *'a set set* **where**
succ S c =
(*if c* ∉ *chain S* | *c* ∈ *maxchain S*
then c else SOME c'. *c'* ∈ *super S c*)

inductive-set
TFin :: *'a set set => 'a set set set*
for *S* :: *'a set set*
where
succI: *x* ∈ *TFin S* ==> *succ S x* ∈ *TFin S*
| *Pow-UnionI*: *Y* ∈ *Pow*(*TFin S*) ==> *Union*(*Y*) ∈ *TFin S*

55.1 Mathematical Preamble

lemma *Union-lemma0*:
(∀ *x* ∈ *C*. *x* ⊆ *A* | *B* ⊆ *x*) ==> *Union*(*C*) ⊆ *A* | *B* ⊆ *Union*(*C*)
⟨*proof*⟩

This is theorem *increasingD2* of ZF/Zorn.thy

lemma *Abrial-axiom1*: *x* ⊆ *succ S x*
⟨*proof*⟩

lemmas *TFin-UnionI* = *TFin.Pow-UnionI* [*OF PowI*]

lemma *TFin-induct*:
assumes *H*: *n* ∈ *TFin S*
and *I*: !!*x*. *x* ∈ *TFin S* ==> *P x* ==> *P* (*succ S x*)
!!*Y*. *Y* ⊆ *TFin S* ==> *Ball Y P* ==> *P*(*Union Y*)
shows *P n* ⟨*proof*⟩

lemma *succ-trans*: *x* ⊆ *y* ==> *x* ⊆ *succ S y*
⟨*proof*⟩

Lemma 1 of section 3.1

lemma *TFin-linear-lemma1*:
[| *n* ∈ *TFin S*; *m* ∈ *TFin S*;
∀ *x* ∈ *TFin S*. *x* ⊆ *m* --> *x* = *m* | *succ S x* ⊆ *m*
|] ==> *n* ⊆ *m* | *succ S m* ⊆ *n*
⟨*proof*⟩

Lemma 2 of section 3.2

lemma *TFin-linear-lemma2*:
m ∈ *TFin S* ==> ∀ *n* ∈ *TFin S*. *n* ⊆ *m* --> *n*=*m* | *succ S n* ⊆ *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*⟩

55.2 Hausdorff's Theorem: Every Set Contains a Maximal Chain.

NB: We assume the partial ordering is $\subseteq$, the subset relation!

lemma *empty-set-mem-chain*: $(\{\} :: 'a\ set\ set) \in chain\ S$
⟨*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*: $c \in chain\ S - maxchain\ S$ ==> $EX\ d.\ d \in super\ S\ c$
⟨*proof*⟩

lemma *select-super*:
$c \in chain\ S - maxchain\ S$ ==> $(\epsilon\ c'.\ c' : super\ S\ c) : super\ S\ c$
⟨*proof*⟩

lemma *select-not-equals*:
$c \in chain\ S - maxchain\ S$ ==> $(\epsilon\ c'.\ c' : super\ S\ c) \neq c$
⟨*proof*⟩

lemma *succI3*: $c \in chain\ S - maxchain\ S$ ==> $succ\ S\ c = (\epsilon\ c'.\ c' : super\ S\ c)$
⟨*proof*⟩

lemma *succ-not-equals*: $c \in chain\ S - maxchain\ S \Longrightarrow succ\ S\ c \neq c$
⟨*proof*⟩

lemma *TFin-chain-lemma4*: $c \in TFin\ S \Longrightarrow (c :: 'a\ set\ set)$: *chain S*
⟨*proof*⟩

theorem *Hausdorff*: $\exists c.\ (c :: 'a\ set\ set)$: *maxchain S*
⟨*proof*⟩

55.3 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*:
$\forall c \in chain\ S.\ Union(c)$: $S \Longrightarrow \exists y \in S.\ \forall z \in S.\ y \subseteq z \longrightarrow y = z$
⟨*proof*⟩

55.4 Alternative version of Zorn's Lemma

lemma *Zorn-Lemma2*:
$\forall c \in chain\ S.\ \exists y \in S.\ \forall x \in c.\ x \subseteq y$
$\Longrightarrow \exists y \in S.\ \forall x \in S.\ (y :: 'a\ set) \subseteq x \longrightarrow y = x$
⟨*proof*⟩

Various other lemmas

lemma *chainD*: $[|\ c \in chain\ S;\ x \in c;\ y \in c\ |] \Longrightarrow x \subseteq y \mid y \subseteq x$
⟨*proof*⟩

lemma *chainD2*: $!!(c :: 'a\ set\ set).\ c \in chain\ S \Longrightarrow c \subseteq S$
⟨*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-order r* **and** *u*: $\forall\, C{\in}Chain\ r.\ \exists\, u{\in}Field\ r.\ \forall\, a{\in}C.\ (a,u){:}r$
shows $\exists\, m{\in}Field\ r.\ \forall\, a{\in}Field\ r.\ (m,a){:}r \longrightarrow a{=}m$
⟨*proof*⟩

definition *init-seg-of* :: $(('a{*}'a)set * ('a{*}'a)set)set$ **where**
init-seg-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-segment-of s == (r,s)*:init-seg-of*

lemma *refl-init-seg-of*[*simp*]: *r initial-segment-of r*
⟨*proof*⟩

lemma *trans-init-seg-of*:
r initial-segment-of s $\Longrightarrow$ *s initial-segment-of t* $\Longrightarrow$ *r initial-segment-of t*
⟨*proof*⟩

lemma *antisym-init-seg-of*:
r initial-segment-of s $\Longrightarrow$ *s initial-segment-of r* $\Longrightarrow$ $r{=}s$
⟨*proof*⟩

lemma *Chain-init-seg-of-Union*:
$R \in Chain\ init\text{-}seg\text{-}of \Longrightarrow r{\in}R \Longrightarrow r\ initial\text{-}segment\text{-}of\ \bigcup R$
⟨*proof*⟩

lemma *chain-subset-trans-Union*:
$chain_{\subseteq}\ R \Longrightarrow \forall\, r{\in}R.\ trans\ r \Longrightarrow trans(\bigcup R)$
⟨*proof*⟩

lemma *chain-subset-antisym-Union*:
$chain_{\subseteq}\ R \Longrightarrow \forall\, r{\in}R.\ antisym\ r \Longrightarrow antisym(\bigcup R)$
⟨*proof*⟩

lemma *chain-subset-Total-Union*:
assumes $chain_{\subseteq}\ R\ \forall\, r{\in}R.\ Total\ r$
shows $Total\ (\bigcup R)$
⟨*proof*⟩

lemma *wf-Union-wf-init-segs*:
assumes $R \in Chain\ init\text{-}seg\text{-}of$ **and** $\forall\, r{\in}R.\ wf\ r$ **shows** $wf(\bigcup R)$
⟨*proof*⟩

lemma *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

56 List-Prefix: List prefixes and postfixes

theory *List-Prefix*
imports *List*
begin

56.1 Prefix order on lists

instantiation *list* :: (*type*) *order*
begin

definition
prefix-def [*code func del*]: $xs \leq ys = (\exists zs.\ ys = xs\ @\ zs)$

definition
strict-prefix-def [*code func del*]: $xs < ys = (xs \leq ys \wedge xs \neq (ys::'a\ list))$

instance
⟨*proof*⟩

end

lemma *prefixI* [*intro?*]: $ys = xs\ @\ zs$ ==> $xs \leq ys$
⟨*proof*⟩

lemma *prefixE* [*elim?*]:
assumes $xs \leq ys$
obtains zs **where** $ys = xs\ @\ zs$
⟨*proof*⟩

lemma *strict-prefixI'* [*intro?*]: $ys = xs\ @\ z\ \#\ zs$ ==> $xs < ys$
⟨*proof*⟩

lemma *strict-prefixE'* [*elim?*]:
assumes $xs < ys$
obtains $z\ zs$ **where** $ys = xs\ @\ z\ \#\ zs$

⟨*proof*⟩

lemma *strict-prefixI* [*intro?*]: $xs \le ys \Longrightarrow xs \ne ys \Longrightarrow xs < (ys::'a\ list)$
⟨*proof*⟩

lemma *strict-prefixE* [*elim?*]:
fixes $xs\ ys :: 'a\ list$
assumes $xs < ys$
obtains $xs \le ys$ **and** $xs \ne ys$
⟨*proof*⟩

56.2 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*]: $(xs\ @\ ys \le xs\ @\ zs) = (ys \le zs)$
⟨*proof*⟩

lemma *same-prefix-nil* [*iff*]: $(xs\ @\ ys \le xs) = (ys = [])$
⟨*proof*⟩

lemma *prefix-prefix* [*simp*]: $xs \le ys \Longrightarrow xs \le ys\ @\ zs$
⟨*proof*⟩

lemma *append-prefixD*: $xs\ @\ ys \le zs \Longrightarrow xs \le zs$
⟨*proof*⟩

theorem *prefix-Cons*: $(xs \le y\ \#\ ys) = (xs = [] \vee (\exists\, zs.\ xs = y\ \#\ zs \wedge zs \le ys))$
⟨*proof*⟩

theorem *prefix-append*:
$(xs \le ys\ @\ zs) = (xs \le ys \vee (\exists\, us.\ xs = ys\ @\ us \wedge us \le zs))$
⟨*proof*⟩

lemma *append-one-prefix*:
$xs \le ys \Longrightarrow length\ xs < length\ ys \Longrightarrow xs\ @\ [ys\ !\ length\ xs] \le ys$
⟨*proof*⟩

theorem *prefix-length-le*: $xs \le ys \Longrightarrow length\ xs \le length\ ys$

⟨*proof*⟩

lemma *prefix-same-cases*:
(*xs*$_1$::'*a list*) ≤ *ys* ⟹ *xs*$_2$ ≤ *ys* ⟹ *xs*$_1$ ≤ *xs*$_2$ ∨ *xs*$_2$ ≤ *xs*$_1$
⟨*proof*⟩

lemma *set-mono-prefix*: *xs* ≤ *ys* ⟹ *set xs* ⊆ *set ys*
⟨*proof*⟩

lemma *take-is-prefix*: *take n xs* ≤ *xs*
⟨*proof*⟩

lemma *map-prefixI*: *xs* ≤ *ys* ⟹ *map f xs* ≤ *map f ys*
⟨*proof*⟩

lemma *prefix-length-less*: *xs* < *ys* ⟹ *length xs* < *length ys*
⟨*proof*⟩

lemma *strict-prefix-simps* [*simp*]:
xs < [] = *False*
[] < (*x* # *xs*) = *True*
(*x* # *xs*) < (*y* # *ys*) = (*x* = *y* ∧ *xs* < *ys*)
⟨*proof*⟩

lemma *take-strict-prefix*: *xs* < *ys* ⟹ *take n xs* < *ys*
⟨*proof*⟩

lemma *not-prefix-cases*:
assumes *pfx*: ¬ *ps* ≤ *ls*
obtains
(*c1*) *ps* ≠ [] **and** *ls* = []
| (*c2*) *a as x xs* **where** *ps* = *a*#*as* **and** *ls* = *x*#*xs* **and** *x* = *a* **and** ¬ *as* ≤ *xs*
| (*c3*) *a as x xs* **where** *ps* = *a*#*as* **and** *ls* = *x*#*xs* **and** *x* ≠ *a*
⟨*proof*⟩

lemma *not-prefix-induct* [*consumes 1*, *case-names Nil Neq Eq*]:
assumes *np*: ¬ *ps* ≤ *ls*
and *base*: ⋀*x xs*. *P* (*x*#*xs*) []
and *r1*: ⋀*x xs y ys*. *x* ≠ *y* ⟹ *P* (*x*#*xs*) (*y*#*ys*)
and *r2*: ⋀*x xs y ys*. ⟦ *x* = *y*; ¬ *xs* ≤ *ys*; *P xs ys* ⟧ ⟹ *P* (*x*#*xs*) (*y*#*ys*)
shows *P ps ls* ⟨*proof*⟩

56.3 Parallel lists

definition
parallel :: '*a list* => '*a list* => *bool* (**infixl** ∥ 50) **where**
(*xs* ∥ *ys*) = (¬ *xs* ≤ *ys* ∧ ¬ *ys* ≤ *xs*)

lemma *parallelI* [*intro*]: ¬ *xs* ≤ *ys* ==> ¬ *ys* ≤ *xs* ==> *xs* ∥ *ys*

⟨*proof*⟩

lemma *parallelE* [*elim*]:
assumes $xs \parallel ys$
obtains $\neg\ xs \le ys \wedge \neg\ ys \le xs$
⟨*proof*⟩

theorem *prefix-cases*:
obtains $xs \le ys \mid ys < xs \mid xs \parallel ys$
⟨*proof*⟩

theorem *parallel-decomp*:
$xs \parallel ys \Longrightarrow \exists\, as\ b\ bs\ c\ cs.\ b \neq c \wedge xs = as\ @\ b\ \#\ bs \wedge ys = as\ @\ c\ \#\ cs$
⟨*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*⟩

56.4 Postfix order on lists

definition
postfix :: *'a list* => *'a list* => *bool* ((-/ >>= -) [*51*, *50*] *50*) **where**
$(xs >>= ys) = (\exists\, zs.\ xs = zs\ @\ ys)$

lemma *postfixI* [*intro?*]: $xs = zs\ @\ ys \Longrightarrow xs >>= ys$
⟨*proof*⟩

lemma *postfixE* [*elim?*]:
assumes $xs >>= ys$
obtains *zs* **where** $xs = zs\ @\ ys$
⟨*proof*⟩

lemma *postfix-refl* [*iff*]: $xs >>= xs$
⟨*proof*⟩
lemma *postfix-trans*: $[\![xs >>= ys;\ ys >>= zs]\!] \Longrightarrow xs >>= zs$
⟨*proof*⟩
lemma *postfix-antisym*: $[\![xs >>= ys;\ ys >>= xs]\!] \Longrightarrow xs = ys$
⟨*proof*⟩

lemma *Nil-postfix* [*iff*]: $xs >>= []$
⟨*proof*⟩
lemma *postfix-Nil* [*simp*]: $([] >>= xs) = (xs = [])$
⟨*proof*⟩

lemma *postfix-ConsI*: $xs >>= ys \Longrightarrow x\#xs >>= ys$
⟨*proof*⟩
lemma *postfix-ConsD*: $xs >>= y\#ys \Longrightarrow xs >>= ys$
⟨*proof*⟩

lemma *postfix-appendI*: $xs >>= ys \Longrightarrow zs @ xs >>= ys$
⟨*proof*⟩
lemma *postfix-appendD*: $xs >>= zs @ ys \Longrightarrow xs >>= ys$
⟨*proof*⟩

lemma *postfix-is-subset*: $xs >>= ys ==> set\ ys \subseteq set\ xs$
⟨*proof*⟩

lemma *postfix-ConsD2*: $x\#xs >>= y\#ys ==> xs >>= ys$
⟨*proof*⟩

lemma *postfix-to-prefix*: $xs >>= ys \longleftrightarrow rev\ ys \leq rev\ xs$
⟨*proof*⟩

lemma *distinct-postfix*: $distinct\ xs \Longrightarrow xs >>= ys \Longrightarrow distinct\ ys$
⟨*proof*⟩

lemma *postfix-map*: $xs >>= ys \Longrightarrow map\ f\ xs >>= map\ f\ ys$
⟨*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*⟩

56.5 Executable code

lemma *less-eq-code* [*code func*]:
$([]::'a::\{eq, ord\}\ list) \leq xs \longleftrightarrow True$
$(x::'a::\{eq, ord\})\ \#\ xs \leq [] \longleftrightarrow False$
$(x::'a::\{eq, ord\})\ \#\ xs \leq y\ \#\ ys \longleftrightarrow x = y \wedge xs \leq ys$
⟨*proof*⟩

lemma *less-code* [*code func*]:
$xs < ([]::'a::\{eq, ord\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{eq, ord\})\#\ xs \longleftrightarrow True$
$(x::'a::\{eq, ord\})\ \#\ xs < y\ \#\ ys \longleftrightarrow x = y \wedge xs < ys$
⟨*proof*⟩

lemmas [*code func*] = *postfix-to-prefix*

end

57 List-lexord: Lexicographic order on lists

theory *List-lexord*
imports *List*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def [*code func del*]: $(xs::('a::ord)\ list) < ys \longleftrightarrow (xs, ys) \in lexord\ \{(u,v).\ u < v\}$

definition
list-le-def [*code func del*]: $(xs::('a::ord)\ list) \leq ys \longleftrightarrow (xs < ys \vee xs = ys)$

instance ⟨*proof*⟩

end

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

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

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition
[*code func del*]: (*inf* :: *'a list* ⇒ -) = *min*

definition
[*code func del*]: (*sup* :: *'a list* ⇒ -) = *max*

instance
⟨*proof*⟩

end

lemma *not-less-Nil* [*simp*]: ¬ (*x* < [])
⟨*proof*⟩

lemma *Nil-less-Cons* [*simp*]: [] < *a* # *x*
⟨*proof*⟩

lemma *Cons-less-Cons* [*simp*]: *a* # *x* < *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* < *y*
⟨*proof*⟩

lemma *le-Nil* [*simp*]: *x* ≤ [] ⟷ *x* = []
⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: [] ≤ *x*
⟨*proof*⟩

lemma *Cons-le-Cons* [*simp*]: *a* # *x* ≤ *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* ≤ *y*
⟨*proof*⟩

lemma *less-code* [*code func*]:
xs < ([]::*'a*::{*eq*, *order*} *list*) ⟷ *False*
[] < (*x*::*'a*::{*eq*, *order*}) # *xs* ⟷ *True*
(*x*::*'a*::{*eq*, *order*}) # *xs* < *y* # *ys* ⟷ *x* < *y* ∨ *x* = *y* ∧ *xs* < *ys*
⟨*proof*⟩

lemma *less-eq-code* [*code func*]:
x # *xs* ≤ ([]::*'a*::{*eq*, *order*} *list*) ⟷ *False*
[] ≤ (*xs*::*'a*::{*eq*, *order*} *list*) ⟷ *True*
(*x*::*'a*::{*eq*, *order*}) # *xs* ≤ *y* # *ys* ⟷ *x* < *y* ∨ *x* = *y* ∧ *xs* ≤ *ys*
⟨*proof*⟩

end

58 Sublist-Order: Sublist Ordering

theory *Sublist-Order*
imports *Main*
begin

This theory defines sublist ordering on lists. A list *ys* is a sublist of a list *xs*, iff one obtains *ys* by erasing some elements from *xs*.

58.1 Definitions and basic lemmas

instantiation *list* :: (*type*) *order*
begin

inductive *less-eq-list* **where**
  empty [*simp*, *intro*!]: $[] \leq xs$
  | *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 func del*]: $(xs :: 'a\ list) < ys \longleftrightarrow xs \leq ys \wedge xs \neq ys$

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*: $xs \leq ys \Longrightarrow xs \leq zs \mathbin{@} ys$
⟨*proof*⟩
lemma *ileq-take-many*: $xs \leq ys \Longrightarrow zs \mathbin{@} xs \leq zs \mathbin{@} ys$
⟨*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*: $xs < ys \Longrightarrow xs < zs \mathbin{@} ys$
⟨*proof*⟩
lemma *ilt-take-many*: $xs < ys \Longrightarrow zs \mathbin{@} xs < zs \mathbin{@} ys$
⟨*proof*⟩

58.2 Appending elements

lemma *ileq-rev-take*: $xs \leq ys \Longrightarrow xs \mathbin{@} [x] \leq ys \mathbin{@} [x]$
⟨*proof*⟩
lemma *ilt-rev-take*: $xs < ys \Longrightarrow xs \mathbin{@} [x] < ys \mathbin{@} [x]$
⟨*proof*⟩
lemma *ileq-rev-drop*: $xs \leq ys \Longrightarrow xs \leq ys \mathbin{@} [x]$
⟨*proof*⟩
lemma *ileq-rev-drop-many*: $xs \leq ys \Longrightarrow xs \leq ys \mathbin{@} zs$
⟨*proof*⟩

58.3 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] H. Davenport. *The Higher Arithmetic.* Cambridge University Press, 1992.

[4] A. Oberschelp. *Rekursionstheorie.* BI-Wissenschafts-Verlag, 1993.

[5] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.