The Supplemental Isabelle/HOL Library

November 22, 2007

Contents

1 GCD: The Greatest Common Divisor

theory *GCD*
imports *Main*
begin

See [3].

1.1 Specification of GCD on nats

definition
$is\text{-}gcd :: nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where** — *gcd* as a relation
$is\text{-}gcd\ p\ m\ n \longleftrightarrow p\ dvd\ m \wedge p\ dvd\ n \wedge$
$(\forall d.\ d\ dvd\ m \longrightarrow d\ dvd\ n \longrightarrow d\ dvd\ p)$

Uniqueness

lemma *is-gcd-unique*: $is\text{-}gcd\ m\ a\ b \Longrightarrow is\text{-}gcd\ n\ a\ b \Longrightarrow m = n$
⟨*proof*⟩

Connection to divides relation

lemma *is-gcd-dvd*: $is\text{-}gcd\ m\ a\ b \Longrightarrow k\ dvd\ a \Longrightarrow k\ dvd\ b \Longrightarrow k\ dvd\ m$
⟨*proof*⟩

Commutativity

lemma *is-gcd-commute*: $is\text{-}gcd\ k\ m\ n = is\text{-}gcd\ k\ n\ m$
⟨*proof*⟩

1.2 GCD on nat by Euclid's algorithm

fun
$gcd :: nat \times nat => nat$
where
$gcd\ (m,\ n) = (if\ n = 0\ then\ m\ else\ gcd\ (n,\ m\ mod\ n))$

lemma *gcd-induct*:
fixes $m\ n :: nat$
assumes $\bigwedge m.\ P\ m\ 0$
and $\bigwedge m\ n.\ 0 < n \Longrightarrow P\ n\ (m\ mod\ n) \Longrightarrow P\ m\ n$
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*: $n > 0 \Longrightarrow gcd\ (m,\ n) = gcd\ (n,\ m\ mod\ n)$
⟨*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*: $k\ dvd\ m \Longrightarrow k\ dvd\ n \Longrightarrow k\ dvd\ gcd\ (m,\ n)$
⟨*proof*⟩

Function gcd yields the Greatest Common Divisor.

lemma *is-gcd*: $is\text{-}gcd\ (gcd\ (m,\ n))\ m\ n$
⟨*proof*⟩

1.3 Derived laws for GCD

lemma *gcd-greatest-iff* [*iff*]: $k\ dvd\ gcd\ (m,\ n) \longleftrightarrow k\ dvd\ m \land k\ dvd\ n$
⟨*proof*⟩

lemma *gcd-zero*: $gcd\ (m,\ n) = 0 \longleftrightarrow m = 0 \land n = 0$
⟨*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*: $gcd\ (k,\ n) = 1 \Longrightarrow k\ dvd\ m * n \Longrightarrow k\ dvd\ m$
⟨*proof*⟩

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

lemma *gcd-mult-cancel*: $gcd\ (k,\ n) = 1 \Longrightarrow gcd\ (k * m,\ n) = gcd\ (m,\ n)$
⟨*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*: $a \neq 0 \vee b \neq 0$
shows $gcd\ (a\ div\ gcd(a,b),\ b\ div\ gcd(a,b)) = 1$
⟨*proof*⟩

1.4 LCM defined by GCD

definition
$lcm :: nat \times nat \Rightarrow nat$
where
$lcm = (\lambda(m,\ n).\ m * n\ div\ gcd\ (m,\ n))$

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*: $igcd\ i\ j = 1 \Longrightarrow i\ dvd\ k * j \Longrightarrow i\ dvd\ k$
⟨*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*: $a \neq 0 \vee b \neq 0$
shows $igcd\ (a\ div\ (igcd\ a\ b))\ (b\ div\ (igcd\ a\ b)) = 1$
⟨*proof*⟩

definition $ilcm = (\lambda i\ j.\ int\ (lcm(nat(abs\ i),nat(abs\ j))))$

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 :: $int \Rightarrow Num$ (-_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
$Nadd = (\lambda(a,b)\ (a',b').\ if\ a = 0 \vee b = 0\ then\ normNum(a',b')$
$else\ if\ a'=0 \vee b' = 0\ then\ normNum(a,b)$
$else\ normNum(a*b' + b*a',\ b*b'))$

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

definition
$Nneg :: Num \Rightarrow Num\ (\sim_N)$
where
$Nneg \equiv (\lambda(a,b).\ (-a,b))$

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

definition
$Ninv :: Num \Rightarrow Num$
where
$Ninv \equiv \lambda(a,b).\ if\ a < 0\ then\ (-b,\ |a|)\ else\ (b,a)$

definition
$Ndiv :: Num \Rightarrow Num \Rightarrow Num$ (**infixl** $\div_N\ 60$)
where
$Ndiv \equiv \lambda a\ b.\ a *_N Ninv\ b$

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

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

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

Relations over Num

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

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

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

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

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

definition
Nle :: *Num* ⇒ *Num* ⇒ *bool* (**infix** $\leq_N$ *55*)
where
$Nle = (\lambda a\ b.\ 0\geq_N (a -_N b))$

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

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

lemma *isnormNum-unique*[*simp*]:
assumes *na*: *isnormNum x* **and** *nb*: *isnormNum y*
shows $((INum\ x :: 'a::\{ring\text{-}char\text{-}0, field,\ division\text{-}by\text{-}zero\}) = INum\ y) = (x = y)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *isnormNum0*[*simp*]: *isnormNum x* $\Longrightarrow$ $(INum\ x = (0::'a::\{ring\text{-}char\text{-}0, field, division\text{-}by\text{-}zero\})) = (x = 0_N)$
⟨*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* [*code*]: (*INum x* ::*'a*::{*field*, *division-by-zero*, *ring-char-0*})
= *INum y* ⟷ *normNum x* = *normNum y* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

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

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

termination ⟨*proof*⟩

fun
  $merge :: ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
    $merge\ qs\ [] = qs$
  $|\ merge\ qs\ (p\#ps) = update\ (fst\ p)\ (snd\ p)\ (merge\ qs\ ps)$

lemma *length-delete-le*: $length\ (delete\ k\ al) \leq length\ al$
⟨*proof*⟩

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

function
  $compose :: ('key \times 'a)\ list \Rightarrow ('a \times 'b)\ list \Rightarrow ('key \times 'b)\ list$
where
    $compose\ []\ ys = []$
  $|\ compose\ (x\#xs)\ ys = (case\ map\text{-}of\ ys\ (snd\ x)$
      $of\ None \Rightarrow compose\ (delete\ (fst\ x)\ xs)\ ys$
       $|\ Some\ v \Rightarrow (fst\ x,\ v)\ \#\ compose\ xs\ ys)$
⟨*proof*⟩
termination ⟨*proof*⟩

fun
  $restrict :: 'key\ set \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
    $restrict\ A\ [] = []$
  $|\ restrict\ A\ (p\#ps) = (if\ fst\ p \in A\ then\ p\#restrict\ A\ ps\ else\ restrict\ A\ ps)$

fun
  $map\text{-}ran :: ('key \Rightarrow 'val \Rightarrow 'val) \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
    $map\text{-}ran\ f\ [] = []$
  $|\ map\text{-}ran\ f\ (p\#ps) = (fst\ p,\ f\ (fst\ p)\ (snd\ p))\ \#\ map\text{-}ran\ f\ ps$

fun
  $clearjunk :: ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
    $clearjunk\ [] = []$
  $|\ clearjunk\ (p\#ps) = p\ \#\ clearjunk\ (delete\ (fst\ p)\ ps)$

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

3.1 Lookup

lemma *lookup-simps* [*code func*]:
  $map\text{-}of\ []\ k = None$

$map\text{-}of\ (p\#ps)\ k = (if\ fst\ p = k\ then\ Some\ (snd\ p)\ else\ map\text{-}of\ ps\ k)$
⟨*proof*⟩

3.2 *delete*

lemma *delete-def*:
$delete\ k\ xs = filter\ (\lambda p.\ fst\ p \neq k)\ xs$
⟨*proof*⟩

lemma *delete-id* [*simp*]: $k \notin fst\ `\ set\ al \Longrightarrow delete\ k\ al = al$
⟨*proof*⟩

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

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

3.3 *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-of* [$q \leftarrow ps$. *fst* $q \neq a$] k = *map-of ps k*
⟨*proof*⟩

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

lemma *length-clearjunk*: *length* (*clearjunk al*) $\leq$ *length al*
⟨*proof*⟩

lemma *notin-fst-filter*: $a \notin fst$ ' *set ps* $\Longrightarrow$ [$q \leftarrow ps$. *fst* $q \neq a$] = *ps*
⟨*proof*⟩

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

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

3.4 *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*⟩

3.5 *update*

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

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

lemma *dom-update*: fst ‘ set $(update\ k\ v\ al) = \{k\} \cup fst$ ‘ $set\ al$
⟨*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*⟩

3.6 *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*]: $size\ ks < size\ vs \Longrightarrow$
$updates\ (ks@[k])\ vs\ al = update\ k\ (vs!size\ ks)\ (updates\ ks\ vs\ al)$
⟨*proof*⟩

lemma *updates-list-update-drop*[*simp*]:
$[\![size\ ks \le i;\ i < size\ vs]\!]$
$\Longrightarrow updates\ ks\ (vs[i:=v])\ al = updates\ ks\ vs\ al$
⟨*proof*⟩

lemma *update-updates-conv-if*:
$map\text{-}of\ (updates\ xs\ ys\ (update\ x\ y\ al)) =$
$map\text{-}of\ (if\ x \in\ set(take\ (length\ ys)\ xs)\ then\ updates\ xs\ ys\ al$
$else\ (update\ x\ y\ (updates\ xs\ ys\ al)))$
⟨*proof*⟩

lemma *updates-twist* [*simp*]:
$k \notin set\ ks \Longrightarrow$
$map\text{-}of\ (updates\ ks\ vs\ (update\ k\ v\ al)) = map\text{-}of\ (update\ k\ v\ (updates\ ks\ vs\ al))$
⟨*proof*⟩

lemma *updates-apply-notin*[*simp*]:
$k \notin set\ ks ==> map\text{-}of\ (updates\ ks\ vs\ al)\ k = map\text{-}of\ al\ k$
⟨*proof*⟩

lemma *updates-append-drop*[*simp*]:
$size\ xs = size\ ys \Longrightarrow updates\ (xs@zs)\ ys\ al = updates\ xs\ ys\ al$
⟨*proof*⟩

lemma *updates-append2-drop*[*simp*]:
$size\ xs = size\ ys \Longrightarrow updates\ xs\ (ys@zs)\ al = updates\ xs\ ys\ al$
⟨*proof*⟩

3.7 *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*) $\Longrightarrow$ *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*⟩

3.8 *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* $\Longrightarrow$ *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* $\wedge$ *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*⟩

3.9 *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* $\circ_m$ *map-of xs*) *k*
⟨*proof*⟩

lemma *compose-conv'*:
shows *map-of* (*compose xs ys*) = (*map-of ys* $\circ_m$ *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*) $\subseteq$ *fst* ‘ *set xs*
⟨*proof*⟩

lemma *distinct-compose*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*compose xs ys*))

$\langle proof \rangle$

lemma *compose-delete-twist*: $(compose\ (delete\ k\ xs)\ ys) = delete\ k\ (compose\ xs\ ys)$
$\langle proof \rangle$

lemma *compose-clearjunk*: $compose\ xs\ (clearjunk\ ys) = compose\ xs\ ys$
$\langle proof \rangle$

lemma *clearjunk-compose*: $clearjunk\ (compose\ xs\ ys) = compose\ (clearjunk\ xs)\ ys$
$\langle proof \rangle$

lemma *compose-empty* [*simp*]:
$compose\ xs\ [] = []$
$\langle proof \rangle$

lemma *compose-Some-iff*:
$(map\text{-}of\ (compose\ xs\ ys)\ k = Some\ v) =$
$(\exists\, k'.\ map\text{-}of\ xs\ k = Some\ k' \land map\text{-}of\ ys\ k' = Some\ v)$
$\langle proof \rangle$

lemma *map-comp-None-iff*:
$(map\text{-}of\ (compose\ xs\ ys)\ k = None) =$
$(map\text{-}of\ xs\ k = None \lor (\exists\, k'.\ map\text{-}of\ xs\ k = Some\ k' \land map\text{-}of\ ys\ k' = None))$

$\langle proof \rangle$

3.10 *restrict*

lemma *restrict-def*:
$restrict\ A = filter\ (\lambda p.\ fst\ p \in A)$
$\langle proof \rangle$

lemma *distinct-restr*: $distinct\ (map\ fst\ al) \Longrightarrow distinct\ (map\ fst\ (restrict\ A\ al))$
$\langle proof \rangle$

lemma *restr-conv*: $map\text{-}of\ (restrict\ A\ al)\ k = ((map\text{-}of\ al)|\text{`}\ A)\ k$
$\langle proof \rangle$

lemma *restr-conv'*: $map\text{-}of\ (restrict\ A\ al) = ((map\text{-}of\ al)|\text{`}\ A)$
$\langle proof \rangle$

lemma *restr-empty* [*simp*]:
$restrict\ \{\}\ al = []$
$restrict\ A\ [] = []$
$\langle proof \rangle$

lemma *restr-in* [*simp*]: $x \in A \Longrightarrow map\text{-}of\ (restrict\ A\ al)\ x = map\text{-}of\ al\ x$
$\langle proof \rangle$

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

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

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

lemma *restr-delete* [*simp*]:
(*delete x* (*restrict D al*)) =
(*if* $x \in D$ *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*]:
$x \in D \Longrightarrow$
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* $\subseteq D$ ⟧
$\Longrightarrow$ *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 *Main*
begin

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

4.1 Basic definitions

instance *set* :: (*plus*) *plus* ⟨*proof*⟩
instance *fun* :: (*type*, *plus*) *plus* ⟨*proof*⟩

defs (**overloaded**)
func-plus: *f* + *g* == (%*x*. *f x* + *g x*)
set-plus: *A* + *B* == {*c*. *EX a*:*A*. *EX b*:*B*. *c* = *a* + *b*}

instance *set* :: (*times*) *times* ⟨*proof*⟩
instance *fun* :: (*type*, *times*) *times* ⟨*proof*⟩

defs (**overloaded**)
func-times: *f* * *g* == (%*x*. *f x* * *g x*)
set-times:*A* * *B* == {*c*. *EX a*:*A*. *EX b*:*B*. *c* = *a* * *b*}

instance *fun* :: (*type*, *minus*) *minus* ⟨*proof*⟩

defs (**overloaded**)
func-minus: − *f* == (%*x*. − *f x*)
func-diff: *f* − *g* == %*x*. *f x* − *g x*

instance *fun* :: (*type*, *zero*) *zero* ⟨*proof*⟩
instance *set* :: (*zero*) *zero* ⟨*proof*⟩

defs (**overloaded**)
func-zero: *0*::((*'a*::*type*) => (*'b*::*zero*)) == %*x*. *0*
set-zero: *0*::(*'a*::*zero*)*set* == {*0*}

instance *fun* :: (*type*, *one*) *one* ⟨*proof*⟩
instance *set* :: (*one*) *one* ⟨*proof*⟩

defs (**overloaded**)
func-one: *1*::((*'a*::*type*) => (*'b*::*one*)) == %*x*. *1*
set-one: *1*::(*'a*::*one*)*set* == {*1*}

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*⟩

instance *set* :: (*semigroup-add*)*semigroup-add*
⟨*proof*⟩

instance *set* :: (*semigroup-mult*)*semigroup-mult*
⟨*proof*⟩

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

instance *set* :: (*comm-monoid-mult*)*comm-monoid-mult*
⟨*proof*⟩

4.2 Basic properties

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

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

lemma *set-plus-rearrange*: $((a{::}'a{::}comm\text{-}monoid\text{-}add) +o\ C) +$

$(b +o\ D) = (a + b) +o\ (C + D)$
⟨proof⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *set-times-mono3* [*intro*]: $a : C ==> a\ {*}o\ D <= C * D$

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

end

5 BigO: Big O notation

theory *BigO*
imports *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*: $ALL\ x.\ f\,x\ {}^{\sim}{=}\ 0 ==>$
$O(f * g) <= (f{::}'a => ('b{::}ordered\text{-}field)) *o\ O(g)$
⟨*proof*⟩

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

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

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

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

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

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

lemma *bigo-const-mult* [*simp*]: (*c*::*'a*::*ordered-field*) ~= *0* ==>
  O(%*x*. *c* ∗ *f x*) = *O*(*f*)
  ⟨*proof*⟩

lemma *bigo-const-mult5* [*simp*]: (*c*::*'a*::*ordered-field*) ~= *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 + 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-field) ~= 0 ==>
(%x. c) * f =o O(h) ==> f =o O(h)
⟨proof⟩

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

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

5.4 Less than or equal to

definition
lesso :: ('a => 'b::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 *Main*
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*: $k \le n ==> (n \ choose \ k) > 0$
⟨*proof*⟩

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

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

lemma *Suc-times-binomial-eq*:
  $!!k. \ k \le n ==> Suc \ n * (n \ choose \ k) = (Suc \ n \ choose \ Suc \ k) * Suc \ k$
⟨*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*:
  $k \le n ==> (Suc \ n \ choose \ Suc \ k) = (Suc \ n * (n \ choose \ k)) \ div \ Suc \ k$
⟨*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 *Main*
begin

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

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

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

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

7.1 Complement

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

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

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

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

7.2 Conjunction

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

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

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

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

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

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

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

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

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

7.3 Disjunction

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

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

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

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

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

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

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

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

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

7.4 De Morgan's Laws

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

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

end

7.5 Symmetric Difference

locale *boolean-xor* = *boolean* +
fixes *xor* :: *'a => 'a => '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 *Main*
begin

instance $*$ :: (*ord*, *ord*) *ord*
prod-le-def: $(x \leq y) \equiv (\mathit{fst}\ x < \mathit{fst}\ y) \vee (\mathit{fst}\ x = \mathit{fst}\ y \wedge \mathit{snd}\ x \leq \mathit{snd}\ y)$
prod-less-def: $(x < y) \equiv (\mathit{fst}\ x < \mathit{fst}\ y) \vee (\mathit{fst}\ x = \mathit{fst}\ y \wedge \mathit{snd}\ x < \mathit{snd}\ y)$ ⟨*proof*⟩

lemmas *prod-ord-defs* [*code func del*] = *prod-less-def prod-le-def*

lemma [*code func*]:

$(x1::'a::\{ord,\ eq\},\ y1) \leq (x2,\ y2) \longleftrightarrow x1 < x2 \lor x1 = x2 \land y1 \leq y2$
$(x1::'a::\{ord,\ eq\},\ y1) < (x2,\ y2) \longleftrightarrow x1 < x2 \lor x1 = x2 \land y1 < y2$
⟨*proof*⟩

lemma [*code*]:
$(x1,\ y1) \leq (x2,\ y2) \longleftrightarrow x1 < x2 \lor x1 = x2 \land y1 \leq y2$
$(x1,\ y1) < (x2,\ y2) \longleftrightarrow x1 < x2 \lor x1 = x2 \land y1 < y2$
⟨*proof*⟩

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

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

instance $*$ :: (*linorder*, *linorder*) *distrib-lattice*
inf-prod-def: *inf* ≡ *min*
sup-prod-def: *sup* ≡ *max*
⟨*proof*⟩

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].

fun
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-BIT 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*⟩

fun
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

instance *nibble* :: *linorder*
nibble-less-eq-def: $n \leq m \equiv$ *nat-of-nibble* $n \leq$ *nat-of-nibble* m
nibble-less-def: $n < m \equiv$ *nat-of-nibble* $n <$ *nat-of-nibble* m
⟨*proof*⟩

instance *nibble* :: *distrib-lattice*
inf ≡ *min*
sup ≡ *max*
⟨*proof*⟩

instance *char* :: *linorder*
char-less-eq-def: *c1* ≤ *c2* ≡ *case c1 of Char n1 m1* ⇒ *case c2 of Char n2 m2* ⇒
n1 < *n2* ∨ *n1* = *n2* ∧ *m1* ≤ *m2*
char-less-def: *c1* < *c2* ≡ *case c1 of Char n1 m1* ⇒ *case c2 of Char n2 m2* ⇒
n1 < *n2* ∨ *n1* = *n2* ∧ *m1* < *m2*
⟨*proof*⟩

lemmas [*code func del*] = *char-less-eq-def char-less-def*

instance *char* :: *distrib-lattice*
inf ≡ *min*
sup ≡ *max*
⟨*proof*⟩

lemma [*simp*, *code func*]:

shows *char-less-eq-simp*: $Char\ n1\ m1 \leq Char\ n2\ m2 \longleftrightarrow n1 < n2 \vee n1 = n2 \wedge m1 \leq m2$
and *char-less-simp*: $Char\ n1\ m1 < Char\ n2\ m2 \longleftrightarrow n1 < n2 \vee n1 = n2 \wedge m1 < m2$
⟨*proof*⟩

end

11 Code-Index: Type of indices

theory *Code-Index*
imports *PreList*
begin

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

11.1 Datatype of indices

datatype *index* = *index-of-int int*

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

fun
int-of-index :: *index* ⇒ *int*
where
int-of-index (*index-of-int k*) = *k*
lemmas [*code func del*] = *int-of-index.simps*

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

lemma *index*:
$(\bigwedge k::index.\ PROP\ P\ k) \equiv (\bigwedge k::int.\ PROP\ P\ (index\text{-}of\text{-}int\ k))$
⟨*proof*⟩

lemma [*code func*]: *size* (*k*::*index*) = *0*
⟨*proof*⟩

11.2 Built-in integers as datatype on numerals

instance *index* :: *number*
number-of ≡ *index-of-int* ⟨*proof*⟩

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

lemma *number-of-index-id* [*simp*]:

number-of (*int-of-index k*) = *k*
⟨*proof*⟩

lemma *number-of-index-shift*:
number-of k = *index-of-int* (*number-of k*)
⟨*proof*⟩

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

11.3 Basic arithmetic

instance *index* :: *zero*
[*simp*]: *0* ≡ *index-of-int 0* ⟨*proof*⟩
lemmas [*code func del*] = *zero-index-def*

instance *index* :: *one*
[*simp*]: *1* ≡ *index-of-int 1* ⟨*proof*⟩
lemmas [*code func del*] = *one-index-def*

instance *index* :: *plus*
[*simp*]: *k* + *l* ≡ *index-of-int* (*int-of-index k* + *int-of-index l*) ⟨*proof*⟩
lemmas [*code func del*] = *plus-index-def*
lemma *plus-index-code* [*code func*]:
index-of-int k + *index-of-int l* = *index-of-int* (*k* + *l*)
⟨*proof*⟩

instance *index* :: *minus*
[*simp*]: − *k* ≡ *index-of-int* (− *int-of-index k*)
[*simp*]: *k* − *l* ≡ *index-of-int* (*int-of-index k* − *int-of-index l*) ⟨*proof*⟩
lemmas [*code func del*] = *uminus-index-def minus-index-def*
lemma *uminus-index-code* [*code func*]:
− *index-of-int k* ≡ *index-of-int* (− *k*)
⟨*proof*⟩
lemma *minus-index-code* [*code func*]:
index-of-int k − *index-of-int l* = *index-of-int* (*k* − *l*)
⟨*proof*⟩

instance *index* :: *times*
[*simp*]: *k* ∗ *l* ≡ *index-of-int* (*int-of-index k* ∗ *int-of-index l*) ⟨*proof*⟩
lemmas [*code func del*] = *times-index-def*
lemma *times-index-code* [*code func*]:
index-of-int k ∗ *index-of-int l* = *index-of-int* (*k* ∗ *l*)
⟨*proof*⟩

instance *index* :: *ord*
[*simp*]: *k* ≤ *l* ≡ *int-of-index k* ≤ *int-of-index l*
[*simp*]: *k* < *l* ≡ *int-of-index k* < *int-of-index l* ⟨*proof*⟩

lemmas [*code func del*] = *less-eq-index-def less-index-def*
lemma *less-eq-index-code* [*code func*]:
  index-of-int k ≤ *index-of-int l* ⟷ *k* ≤ *l*
  ⟨*proof*⟩
lemma *less-index-code* [*code func*]:
  index-of-int k < *index-of-int l* ⟷ *k* < *l*
  ⟨*proof*⟩

instance *index* :: *Divides.div*
  [*simp*]: *k div l* ≡ *index-of-int* (*int-of-index k div int-of-index l*)
  [*simp*]: *k mod l* ≡ *index-of-int* (*int-of-index k mod int-of-index l*) ⟨*proof*⟩

instance *index* :: *ring-1*
  ⟨*proof*⟩

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

instance *index* :: *number-ring*
  ⟨*proof*⟩

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

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

instance *index* :: *abs*
  |*k*| ≡ *if k* < *0 then* −*k else k* ⟨*proof*⟩

lemma *index-of-int* [*code func*]:
  index-of-int k = (*if k* = *0 then 0*
    else if k = −*1 then* −*1*
    else let (*l*, *m*) = *divAlg* (*k*, *2*) *in 2* ∗ *index-of-int l* +
      (*if m* = *0 then 0 else 1*))
  ⟨*proof*⟩

lemma *int-of-index* [*code func*]:
  int-of-index k = (*if k* = *0 then 0*
    else if k = −*1 then* −*1*
    else let l = *k div 2*; *m* = *k mod 2 in 2* ∗ *int-of-index l* +
      (*if m* = *0 then 0 else 1*))
  ⟨*proof*⟩

11.4 Conversion to and from *nat*

definition

nat-of-index :: *index* ⇒ *nat*
where
[*code func del*]: *nat-of-index* = *nat o int-of-index*

definition
nat-of-index-aux :: *index* ⇒ *nat* ⇒ *nat* **where**
[*code func del*]: *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*⟩

definition
index-of-nat :: *nat* ⇒ *index*
where
[*code func del*]: *index-of-nat* = *index-of-int o of-nat*

lemma *index-of-nat* [*code func*]:
index-of-nat 0 = *0*
index-of-nat (*Suc n*) = *index-of-nat n* + *1*
⟨*proof*⟩

lemma *index-nat-id* [*simp*]:
nat-of-index (*index-of-nat n*) = *n*
index-of-nat (*nat-of-index i*) = (*if i* ≤ *0 then 0 else i*)
⟨*proof*⟩

11.5 ML interface

⟨*ML*⟩

11.6 Code serialization

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

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

⟨*ML*⟩

code-reserved *SML int*
code-reserved *OCaml int*

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

code-const *uminus* :: *index* ⇒ *index*
 (*SML Int*.~)
 (*OCaml Pervasives*.~−)
 (*Haskell negate*)

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

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

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

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

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

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

end

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

theory *Code-Message*
imports *List*
begin

12.1 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*⟩

12.2 ML interface

⟨*ML*⟩

12.3 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

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

theory *Coinductive-List*
imports *Main*
begin

13.1 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*]: $NIL \neq CONS\ M\ N$
and *CONS-inject* [*iff*]: $(CONS\ K\ M) = (CONS\ L\ N) = (K = L \wedge M = N)$
⟨*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* $c\ h = Datatype.Case\ (\lambda\text{-}.\ c)\ (Datatype.Split\ h)$

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

13.2 Corecursive lists

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

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

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

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

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

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

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

13.3 Abstract type definition

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

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

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

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

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

lemma *Rep-llist-UNIV*: $Rep\text{-}llist\ x \in LList\ UNIV$
⟨*proof*⟩

definition $LNil = Abs\text{-}llist\ NIL$
definition $LCons\ x\ xs = Abs\text{-}llist\ (CONS\ (Datatype.Leaf\ x)\ (Rep\text{-}llist\ xs))$

lemma *LCons-not-LNil* [*iff*]: $LCons\ x\ xs \neq LNil$
⟨*proof*⟩

lemma *LNil-not-LCons* [*iff*]: $LNil \neq LCons\ x\ xs$
⟨*proof*⟩

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

lemma *Rep-llist-LNil*: $Rep\text{-}llist\ LNil = NIL$
⟨*proof*⟩

lemma *Rep-llist-LCons*: $Rep\text{-}llist\ (LCons\ x\ l) =$
$CONS\ (Datatype.Leaf\ x)\ (Rep\text{-}llist\ l)$
⟨*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 (λ*x y. d* (*inv Datatype.Leaf x*) (*Abs-llist y*)) (*Rep-llist l*)

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

lemma *llist-case-LNil* [*simp*]: *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*⟩

13.4 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*:
  $(M, N) \in EqLList\ (diag\ A) \Longrightarrow ntrunc\ k\ M = ntrunc\ k\ N$
  ⟨*proof*⟩

lemma *Domain-EqLList*: $Domain\ (EqLList\ (diag\ A)) \subseteq LList\ A$
  ⟨*proof*⟩

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

lemma *EqLList-diag-iff* [*iff*]: $(p \in EqLList\ (diag\ A)) = (p \in diag\ (LList\ A))$
  ⟨*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: $(M, N) \in r$
    and *step*: $\bigwedge M\ N.\ (M, N) \in r \Longrightarrow$
      $M = NIL \wedge N = NIL \vee$
        $(\exists\, a\ b\ M'\ N'.$
          $M = CONS\ a\ M' \wedge N = CONS\ b\ N' \wedge (a, b) \in diag\ A \wedge$
            $((M', N') \in r \vee (M', N') \in EqLList\ (diag\ A)))$
  shows $M = N$
⟨*proof*⟩

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

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

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

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

13.5 Derived operations – both on the set and abstract type

13.5.1 *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*⟩

13.5.2 *Lmap* and *lmap*

definition
$Lmap\ f\ M = LList\text{-}corec\ M\ (List\text{-}case\ None\ (\lambda x\ M'.\ Some\ (f\ x,\ M')))$
definition
$lmap\ f\ l = llist\text{-}corec\ l$
$(\lambda z.$
$case\ z\ of\ LNil \Rightarrow None$
$|\ LCons\ y\ z \Rightarrow Some\ (f\ y,\ z))$

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

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

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

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

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

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

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

13.5.3 *Lappend*

definition
$Lappend\ M\ N = LList\text{-}corec\ (M,\ N)$
$(split\ (List\text{-}case$
$(List\text{-}case\ None\ (\lambda N1\ N2.\ Some\ (N1,\ (NIL,\ N2))))$
$(\lambda M1\ M2\ N.\ Some\ (M1,\ (M2,\ N)))))$
definition
$lappend\ l\ n = llist\text{-}corec\ (l,\ n)$
$(split\ (llist\text{-}case$

$(llist\text{-}case\ None\ (\lambda n1\ n2.\ Some\ (n1,\ (LNil,\ n2))))$
$(\lambda l1\ l2\ n.\ Some\ (l1,\ (l2,\ n)))))$

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

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

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

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

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

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

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

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

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

13.6 iterates

llist-fun-equalityI cannot be used here!

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

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

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

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

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

lemma *funpow-lmap*:
fixes $f :: 'a \Rightarrow 'a$
shows (*lmap f* ^ *n*) (*LCons b l*) = *LCons* ((*f* ^ *n*) *b*) ((*lmap f* ^ *n*) *l*)
⟨*proof*⟩

lemma *iterates-equality*:
assumes *h*: $\bigwedge x.$ *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

14 Parity: Even and Odd for int and nat

theory *Parity*
imports *Main*
begin

class *even-odd* = *type* +
fixes *even* :: $'a \Rightarrow$ *bool*

abbreviation
odd :: $'a$::*even-odd* $\Rightarrow$ *bool* **where**
odd x ≡ ¬ *even x*

instance *int* :: *even-odd*
even-def [*presburger*]: *even x* ≡ *x mod 2* = *0* ⟨*proof*⟩

instance *nat* :: *even-odd*
even-nat-def [*presburger*]: *even x* ≡ *even* (*int x*) ⟨*proof*⟩

14.1 Even and odd are mutually exclusive

lemma *int-pos-lt-two-imp-zero-or-one*:
0 <= *x* ==> (*x*::*int*) < *2* ==> *x* = *0* | *x* = *1*

⟨*proof*⟩

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

14.2 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*: $odd\ x ==> odd((x::int)\hat{}n)$ ⟨*proof*⟩

lemma *even-power*[*presburger*]: $even\ ((x::int)\hat{}n) = (even\ x\ \&\ 0 < n)$

⟨*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*

14.3 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*⟩

14.4 even and odd for nats

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

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

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

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

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

lemma *even-nat-power[presburger]*: *even* $((x::nat)\hat{\ }y) = (even \ x \ \& \ 0 < y)$
⟨*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*

14.5 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*⟩

14.6 Parity and powers

lemma *minus-one-even-odd-power*:
(*even x* −−> (− *1*::′*a*::{*comm-ring-1*,*recpower*}) ˆ*x* = *1*) &
(*odd x* −−> (− *1*::′*a*) ˆ*x* = − *1*)
⟨*proof*⟩

lemma *minus-one-even-power* [*simp*]:
even x ==> (− *1*::′*a*::{*comm-ring-1*,*recpower*}) ˆ*x* = *1*
⟨*proof*⟩

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

lemma *neg-one-even-odd-power*:
(*even x* −−> (−*1*::′*a*::{*number-ring*,*recpower*}) ˆ*x* = *1*) &
(*odd x* −−> (−*1*::′*a*) ˆ*x* = −*1*)
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

lemma *power-minus-odd* [*simp*]: *odd n ==>*

$(- x)\hat{\ }n = - (x\hat{\ }n::'a::\{recpower,comm\text{-}ring\text{-}1\})$
⟨*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*]

14.7 An Equivalence for $0 \le a\hat{\ }n$

lemma *even-power-le-0-imp-0*:
$a \ \hat{\ } \ (2*k) \le (0::'a::\{ordered\text{-}idom,recpower\}) ==> a=0$
⟨*proof*⟩

lemma *zero-le-power-iff* [*presburger*]:
$(0 \le a\hat{\ }n) = (0 \le (a::'a::\{ordered\text{-}idom,recpower\}) \mid even\ n)$
⟨*proof*⟩

14.8 Miscellaneous

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

lemma [*presburger*]: (*Suc x*) *div Suc* (*Suc 0*) = *x div Suc* (*Suc 0*) ⟷ *even x*
⟨*proof*⟩
lemma [*presburger*]: (*Suc x*) *div Suc* (*Suc 0*) = *x div Suc* (*Suc 0*) ⟷ *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

15 Commutative-Ring: Proving equalities in commutative rings

theory *Commutative-Ring*
imports *Main 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*)ˆ*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 ˆ 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* $\Rightarrow$ $'a$ *pol* $\Rightarrow$ $'a$ *pol* (**infixl** $\otimes$ *70*)
where
$Pc\ a \otimes Pc\ b = Pc\ (a * b)$
$|\ Pc\ c \otimes Pinj\ i\ P =$
$(if\ c = 0\ then\ Pc\ 0\ else\ mkPinj\ i\ (P \otimes Pc\ c))$
$|\ Pinj\ i\ P \otimes Pc\ c =$
$(if\ c = 0\ then\ Pc\ 0\ else\ mkPinj\ i\ (P \otimes Pc\ c))$
$|\ Pc\ c \otimes PX\ P\ i\ Q =$
$(if\ c = 0\ then\ Pc\ 0\ else\ mkPX\ (P \otimes Pc\ c)\ i\ (Q \otimes Pc\ c))$
$|\ PX\ P\ i\ Q \otimes Pc\ c =$
$(if\ c = 0\ then\ Pc\ 0\ else\ mkPX\ (P \otimes Pc\ c)\ i\ (Q \otimes Pc\ c))$
$|\ Pinj\ x\ P \otimes Pinj\ y\ Q =$
$(if\ x = y\ then\ mkPinj\ x\ (P \otimes Q)\ else$
$(if\ x > y\ then\ mkPinj\ y\ (Pinj\ (x-y)\ P \otimes Q)$
$else\ mkPinj\ x\ (Pinj\ (y - x)\ Q \otimes P)))$
$|\ Pinj\ x\ P \otimes PX\ Q\ y\ R =$
$(if\ x = 0\ then\ P \otimes PX\ Q\ y\ R\ else$
$(if\ x = 1\ then\ mkPX\ (Pinj\ x\ P \otimes Q)\ y\ (R \otimes P)$
$else\ mkPX\ (Pinj\ x\ P \otimes Q)\ y\ (R \otimes Pinj\ (x - 1)\ P)))$
$|\ PX\ P\ x\ R \otimes Pinj\ y\ Q =$
$(if\ y = 0\ then\ PX\ P\ x\ R \otimes Q\ else$
$(if\ y = 1\ then\ mkPX\ (Pinj\ y\ Q \otimes P)\ x\ (R \otimes Q)$
$else\ mkPX\ (Pinj\ y\ Q \otimes P)\ x\ (R \otimes Pinj\ (y - 1)\ Q)))$
$|\ PX\ P1\ x\ P2 \otimes PX\ Q1\ y\ Q2 =$
$mkPX\ (P1 \otimes Q1)\ (x + y)\ (P2 \otimes Q2) \oplus$
$(mkPX\ (P1 \otimes mkPinj\ 1\ Q2)\ x\ (Pc\ 0) \oplus$
$(mkPX\ (Q1 \otimes mkPinj\ 1\ P2)\ y\ (Pc\ 0)))$
⟨*proof*⟩
termination ⟨*proof*⟩

Negation

fun
neg :: $'a$::{*comm-ring*,*recpower*} *pol* $\Rightarrow$ $'a$ *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* $\Rightarrow$ $'a$ *pol* $\Rightarrow$ $'a$ *pol* (**infixl** $\ominus$ *65*)
where
$sub\ P\ Q = P \oplus neg\ Q$

Square for Fast Exponentation

fun
sqr :: $'a$::{*comm-ring*,*recpower*} *pol* $\Rightarrow$ $'a$ *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*: $Ipol\ l\ (P \otimes Q) = Ipol\ l\ P * Ipol\ l\ Q$
⟨*proof*⟩

Substraction

lemma *sub-ci*: $Ipol\ l\ (P \ominus Q) = Ipol\ l\ P - Ipol\ l\ Q$
⟨*proof*⟩

Square

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

Power

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

lemma *pow-ci*: $Ipol\ ls\ (pow\ n\ P) = Ipol\ ls\ P\ \hat{}\ n$
⟨*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

16 Continuity: Continuity and iterations (of set transformers)

theory *Continuity*
imports *Main*
begin

16.1 Continuity for complete lattices

definition
$chain :: (nat \Rightarrow 'a::complete\text{-}lattice) \Rightarrow bool$ **where**
$chain\ M \longleftrightarrow (\forall i.\ M\ i \le M\ (Suc\ i))$

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*.

16.2 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 $y \subseteq F$ x
⟨*proof*⟩

lemma *down-chain-mono*: *down-chain* F ==> $x \leq y$ ==> F $y \subseteq F$ x
⟨*proof*⟩

16.3 Continuity

definition
up-cont :: ('*a set* => '*a set*) => *bool* **where**
up-cont f = ($\forall$ F. *up-chain* F --> f ($\bigcup$(*range* F)) = $\bigcup$(f ‘ *range* F))

lemma *up-contI*:
(!!F. *up-chain* F ==> f ($\bigcup$(*range* F)) = $\bigcup$(f ‘ *range* F)) ==> *up-cont f*
⟨*proof*⟩

lemma *up-contD*:
up-cont f ==> *up-chain* F ==> f ($\bigcup$(*range* F)) = $\bigcup$(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 =
($\forall$ 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*⟩

16.4 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

17 Code-Integer: Pretty integer literals for code generation

theory *Code-Integer*
imports *IntArith Code-Index*
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 *Numeral.Pls* **and** *Numeral.Min* **and** *Numeral.Bit*
(*SML raise/ Fail/ Pls*
and *raise/ Fail/ Min*
and !((-);/ (-);/ *raise/ Fail/ Bit*))
(*OCaml failwith/ Pls*
and *failwith/ Min*
and !((-);/ (-);/ *failwith/ Bit*))
(*Haskell error/ Pls*
and *error/ Min*
and *error/ Bit*)

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

code-const *Numeral.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-const *index-of-int* **and** *int-of-index*
(*SML IntInf.toInt* **and** *IntInf.fromInt*)
(*OCaml Big'-int.int'-of'-big'-int* **and** *Big'-int.big'-int'-of'-int*)
(*Haskell* - **and** -)

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

end

18 Efficient-Nat: Implementation of natural numbers by integers

theory *Efficient-Nat*
imports *Main Code-Integer*
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 integers. To do this, just include this theory.

18.1 Logical rewrites

An int-to-nat conversion restricted to non-negative ints (in contrast to *nat*). Note that this restriction has no logical relevance and is just a kind of proof hint – nothing prevents you from writing nonsense like *nat-of-int* $(-4::'a)$

definition
  nat-of-int :: *int* ⇒ *nat* **where**
  $k \geq 0 \Longrightarrow$ *nat-of-int* k = *nat* k

definition
  int-of-nat :: *nat* ⇒ *int* **where**
  int-of-nat n = *of-nat* n

lemma *int-of-nat-Suc* [*simp*]:
  int-of-nat (*Suc* n) = 1 + *int-of-nat* n
  ⟨*proof*⟩

lemma *int-of-nat-add*:
  int-of-nat $(m + n)$ = *int-of-nat* m + *int-of-nat* n
  ⟨*proof*⟩

lemma *int-of-nat-mult*:
  int-of-nat $(m * n)$ = *int-of-nat* m $*$ *int-of-nat* n
  ⟨*proof*⟩

lemma *nat-of-int-of-number-of*:
  fixes k
  assumes $k \geq 0$
  shows *number-of* k = *nat-of-int* (*number-of* k)
  ⟨*proof*⟩

lemma *nat-of-int-of-number-of-aux*:
  fixes k
  assumes *Numeral.Pls* $\leq k \equiv$ *True*
  shows $k \geq 0$
  ⟨*proof*⟩

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

lemma *eq-nat-of-int*: *int-of-nat* $n = x \Longrightarrow n$ = *nat-of-int* x
⟨*proof*⟩

code-datatype *nat-of-int*

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

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

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

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

lemma [*code func*]: $0 = nat\text{-}of\text{-}int\ 0$
⟨*proof*⟩

lemma [*code func*, *code inline*]: $1 = nat\text{-}of\text{-}int\ 1$
⟨*proof*⟩

lemma [*code func*]: $Suc\ n = nat\text{-}of\text{-}int\ (int\text{-}of\text{-}nat\ n + 1)$
⟨*proof*⟩

lemma [*code*]: $m + n = nat\ (int\text{-}of\text{-}nat\ m + int\text{-}of\text{-}nat\ n)$
⟨*proof*⟩

lemma [*code func*, *code inline*]: $m + n = nat\text{-}of\text{-}int\ (int\text{-}of\text{-}nat\ m + int\text{-}of\text{-}nat\ n)$
⟨*proof*⟩

lemma [*code*, *code inline*]: $m - n = nat\ (int\text{-}of\text{-}nat\ m - int\text{-}of\text{-}nat\ n)$
⟨*proof*⟩

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

lemma [*code func*, *code inline*]: $m * n = nat\text{-}of\text{-}int\ (int\text{-}of\text{-}nat\ m * int\text{-}of\text{-}nat\ n)$
⟨*proof*⟩

lemma [*code*]: $m\ div\ n = nat\ (int\text{-}of\text{-}nat\ m\ div\ int\text{-}of\text{-}nat\ n)$
⟨*proof*⟩

lemma *div-nat-code* [*code func*]:
$m\ div\ k = nat\text{-}of\text{-}int\ (fst\ (divAlg\ (int\text{-}of\text{-}nat\ m,\ int\text{-}of\text{-}nat\ k)))$
⟨*proof*⟩

lemma [*code*]: $m\ mod\ n = nat\ (int\text{-}of\text{-}nat\ m\ mod\ int\text{-}of\text{-}nat\ n)$
⟨*proof*⟩

lemma *mod-nat-code* [*code func*]:

$m\ mod\ k = nat\text{-}of\text{-}int\ (snd\ (divAlg\ (int\text{-}of\text{-}nat\ m,\ int\text{-}of\text{-}nat\ k)))$
⟨*proof*⟩

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

lemma [*code func*, *code inline*]: $(m \leq n) \longleftrightarrow (int\text{-}of\text{-}nat\ m \leq int\text{-}of\text{-}nat\ n)$
⟨*proof*⟩

lemma [*code func*, *code inline*]: $m = n \longleftrightarrow int\text{-}of\text{-}nat\ m = int\text{-}of\text{-}nat\ n$
⟨*proof*⟩

lemma [*code func*]: $nat\ k = (if\ k < 0\ then\ 0\ else\ nat\text{-}of\text{-}int\ k)$
⟨*proof*⟩

lemma [*code func*]:
$int\text{-}aux\ n\ i = (if\ int\text{-}of\text{-}nat\ n = 0\ then\ i\ else\ int\text{-}aux\ (nat\text{-}of\text{-}int\ (int\text{-}of\text{-}nat\ n - 1))\ (i + 1))$
⟨*proof*⟩

lemma *index-of-nat-code* [*code func*, *code inline*]:
$index\text{-}of\text{-}nat\ n = index\text{-}of\text{-}int\ (int\text{-}of\text{-}nat\ n)$
⟨*proof*⟩

lemma *nat-of-index-code* [*code func*, *code inline*]:
$nat\text{-}of\text{-}index\ k = nat\ (int\text{-}of\text{-}index\ k)$
⟨*proof*⟩

18.2 Code generator setup for basic functions

nat is no longer a datatype but embedded into the integers.

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

types-code
nat (*int*)
attach (*term-of*) ⟪
val term-of-nat = HOLogic.mk-number HOLogic.natT;
⟫
attach (*test*) ⟪
fun gen-nat i = random-range 0 i;
⟫

consts-code
0 :: nat (*0*)
Suc ((- + *1*))

Since natural numbers are implemented using integers, the coercion function *int* of type $nat \Rightarrow int$ is simply implemented by the identity function, likewise *nat-of-int* of type $int \Rightarrow nat$. 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*.

consts-code
  int-of-nat ((-))
  nat (⟨**module**⟩*nat*)
attach ⟪
fun nat i = if i < 0 then 0 else i;
⟫

code-const *int-of-nat*
  (*SML* -)
  (*OCaml* -)
  (*Haskell* -)

code-const *nat-of-int*
  (*SML* -)
  (*OCaml* -)
  (*Haskell* -)

18.3 Preprocessors

Natural numerals should be expressed using *nat-of-int*.

lemmas [*code inline del*] = *nat-number-of-def*

⟨*ML*⟩

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:

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

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

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

⟨*ML*⟩

18.4 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 nat-of-int int-of-nat*

end

19 Eval-Witness: Evaluation Oracle with ML witnesses

theory *Eval-Witness*
imports *Main*
begin

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

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

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

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

class *ml-equiv* = *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*⟩

19.1 Toy Examples

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

lemma $\exists\, n{::}nat.\ n = 1$
⟨*proof*⟩

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

lemma $\exists\, l.\ length\ (l{::}bool\ list) = 3$
⟨*proof*⟩

Multiple witnesses

lemma $\exists\, k\ l.\ length\ (k{::}bool\ list) = length\ (l{::}bool\ list)$
⟨*proof*⟩

19.2 Discussion

19.2.1 Conflicts

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

19.2.2 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

20 Executable-Set: Implementation of finite sets by lists

theory *Executable-Set*
imports *Main*
begin

20.1 Definitional rewrites

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

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

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

20.2 Operations on lists

20.2.1 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* $(x\#xs)\ y \longleftrightarrow y = x \vee member\ xs\ y$
$\langle proof \rangle$

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* (λy. $y = x$) *xs else xs*)
⟨*proof*⟩

lemma *member-nil* [*simp*]:
member [] = (λx. *False*)
⟨*proof*⟩

lemma *member-insertl* [*simp*]:
$x \in$ *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*: $\neg$ (*member xs x*)
shows *insertl x xs* = *x* # *xs*
⟨*proof*⟩

lemma *foldr-remove1-empty* [*simp*]:
foldr remove1 xs [] = []
⟨*proof*⟩

20.2.2 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-def* = *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-def* = *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-def* = *subtract.simps*(*3*)

function *map-distinct* :: ($'a \Rightarrow 'b$) $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'b$ *list*
where
  map-distinct f [] = []
| *map-distinct f xs* = *foldr* (*insertl o f*) *xs* []
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *map-distinct-def* = *map-distinct.simps*(*2*)

function *unions* :: $'a$ *list list* $\Rightarrow$ $'a$ *list*
where
  unions [] = []
| *unions xs* = *foldr unionl xs* []
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *unions-def* = *unions.simps*(*2*)

consts *intersects* :: $'a$ *list list* $\Rightarrow$ $'a$ *list*
primrec
  intersects (*x*#*xs*) = *foldr intersect xs x*

definition
  map-union :: $'a$ *list* $\Rightarrow$ ($'a \Rightarrow 'b$ *list*) $\Rightarrow$ $'b$ *list* **where**
  map-union xs f = *unions* (*map f xs*)

definition
  map-inter :: $'a$ *list* $\Rightarrow$ ($'a \Rightarrow 'b$ *list*) $\Rightarrow$ $'b$ *list* **where**
  map-inter xs f = *intersects* (*map f xs*)

20.3 Isomorphism proofs

lemma *iso-member*:
  member xs x $\longleftrightarrow$ $x \in$ *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 $\cup$ *set ys*
⟨*proof*⟩

lemma *iso-intersect*:
set (intersect xs ys) = set xs $\cap$ *set ys*
⟨*proof*⟩

definition
subtract' :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'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) = $\bigcup$ *set (map set xss)*
⟨*proof*⟩

lemma *iso-intersects*:
set (intersects (xs#xss)) = $\bigcap$ *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* $\Rightarrow$ *('a* $\Rightarrow$ *bool)* $\Rightarrow$ *bool* **where**
Blall = flip list-all
definition
Blex :: *'a list* $\Rightarrow$ *('a* $\Rightarrow$ *bool)* $\Rightarrow$ *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*⟩

20.4 code generator setup

⟨*ML*⟩

20.4.1 type serializations

types-code
set (- *list*)
attach (*term-of*) ⟪
fun term-of-set f T [] = *Const* ({}, *Type* (*set*, [*T*]))
| *term-of-set f T* (*x* :: *xs*) = *Const* (*insert*,
T --> *Type* (*set*, [*T*]) --> *Type* (*set*, [*T*])) \$ *f x* \$ *term-of-set f T xs*;
⟫
attach (*test*) ⟪
fun gen-set' aG i j = *frequency*
[(*i*, *fn* () => *aG j* :: *gen-set' aG* (*i*−*1*) *j*), (*1*, *fn* () => [])] ()
and gen-set aG i = *gen-set' aG i i*;
⟫

20.4.2 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

21 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 ((3Π -∈-./ -) 10)
-lam :: [pttrn, 'a set, 'a => 'b] => ('a=>'b) ((3λ-∈-./ -) [0,0,3] 3)

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

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

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

21.1 Basic Properties of *Pi*

lemma *Pi-I*: $(!!x.\ x \in A ==> f\,x \in B\ x) ==> f \in Pi\ A\ B$
⟨*proof*⟩

lemma *funcsetI*: $(!!x.\ x \in A ==> f\,x \in B) ==> f \in A -> B$
⟨*proof*⟩

lemma *Pi-mem*: $[|f\colon Pi\ A\ B;\ x \in A|] ==> f\,x \in B\ x$
⟨*proof*⟩

lemma *funcset-mem*: $[|f \in A -> B;\ x \in A|] ==> f\,x \in B$
⟨*proof*⟩

lemma *funcset-image*: $f \in A \rightarrow B ==> f\ \text{`}\ A \subseteq B$
⟨*proof*⟩

lemma *Pi-eq-empty*: $((PI\ x\colon A.\ B\ x) = \{\}) = (\exists\, x{\in}A.\ B(x) = \{\})$
⟨*proof*⟩

lemma *Pi-empty* [*simp*]: $Pi\ \{\}\ B = UNIV$
⟨*proof*⟩

lemma *Pi-UNIV* [*simp*]: $A -> UNIV = UNIV$
⟨*proof*⟩

Covariance of Pi-sets in their second argument

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

Contravariance of Pi-sets in their first argument

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

21.2 Composition With a Restricted Domain: *compose*

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

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

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

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

21.3 Bounded Abstraction: *restrict*

lemma *restrict-in-funcset*: $(!!x.\ x \in A ==> f\,x \in B) ==> (\lambda x{\in}A.\ f\,x) \in A\ {-}{>}\ B$
$\langle proof \rangle$

lemma *restrictI*: $(!!x.\ x \in A ==> f\,x \in B\ x) ==> (\lambda x{\in}A.\ f\,x) \in Pi\ A\ B$
$\langle proof \rangle$

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

lemma *restrict-ext*:
$(!!x.\ x \in A ==> f\,x = g\ x) ==> (\lambda x{\in}A.\ f\,x) = (\lambda x{\in}A.\ g\ x)$
$\langle proof \rangle$

lemma *inj-on-restrict-eq* [*simp*]: $inj\text{-}on\ (restrict\ f\ A)\ A = inj\text{-}on\ f\ A$
$\langle proof \rangle$

lemma *Id-compose*:
$[|f \in A\ {-}{>}\ B;\ \ f \in extensional\ A|] ==> compose\ A\ (\lambda y{\in}B.\ y)\ f = f$
$\langle proof \rangle$

lemma *compose-Id*:
$[|g \in A\ {-}{>}\ B;\ \ g \in extensional\ A|] ==> compose\ A\ g\ (\lambda x{\in}A.\ x) = g$
$\langle proof \rangle$

lemma *image-restrict-eq* [*simp*]: $(restrict\ f\ A)\ `\ A = f\ `\ A$
$\langle proof \rangle$

21.4 Bijections Between Sets

The basic definition could be moved to *Fun.thy*, but most of the theorems belong here, or need at least *Hilbert-Choice*.

definition
$bij\text{-}betw :: [{'}a => {'}b,\ {'}a\ set,\ {'}b\ set] => bool$ **where** — bijective
$bij\text{-}betw\ f\ A\ B = (inj\text{-}on\ f\ A\ \&\ f\ `\ A = B)$

lemma *bij-betw-imp-inj-on*: $bij\text{-}betw\ f\ A\ B \Longrightarrow inj\text{-}on\ f\ A$
$\langle proof \rangle$

lemma *bij-betw-imp-funcset*: $bij\text{-}betw\ f\ A\ B \Longrightarrow f \in A \to B$
$\langle proof \rangle$

lemma *bij-betw-Inv*: $bij\text{-}betw\ f\ A\ B \Longrightarrow bij\text{-}betw\ (Inv\ A\ f)\ B\ A$
$\langle proof \rangle$

lemma *inj-on-compose*:

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

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

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

21.5 Extensionality

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

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

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

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

lemma *Inv-funcset*: $f \; ` \; A = B \; ==> \; (\lambda x \in B. \; \textit{Inv} \; A \; f \; x) : B \; -> \; A$
⟨*proof*⟩

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

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

21.6 Cardinality

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

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

22 Infinite-Set: Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

22.1 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* $\Rightarrow$ *bool* **where**
  infinite S == $\neg$ *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 $(S::nat\ set) = (\forall\, m.\ \exists\, n.\ m{\le}n \wedge n{\in}S)$
(**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** $y \in f`A$ **and** *infinite* $(f -`\ \{y\})$
⟨*proof*⟩

22.2 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
$Inf\text{-}many :: ('a \Rightarrow bool) \Rightarrow bool$ (**binder** *INFM 10*) **where**
$Inf\text{-}many\ P = infinite\ \{x.\ P\ x\}$

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

notation (*xsymbols*)
Inf-many (**binder** $\exists_\infty$ *10*) **and**
Alm-all (**binder** $\forall_\infty$ *10*)

notation (*HTML* **output**)
Inf-many (**binder** $\exists_\infty$ *10*) **and**
Alm-all (**binder** $\forall_\infty$ *10*)

lemma *INF-EX*:
$(\exists_\infty x.\ P\ x) \Longrightarrow (\exists x.\ P\ x)$
$\langle proof \rangle$

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

lemma *ALL-MOST*: $\forall x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$
$\langle proof \rangle$

lemma *INF-mono*:
assumes *inf*: $\exists_\infty x.\ P\ x$ **and** *q*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\exists_\infty x.\ Q\ x$
$\langle proof \rangle$

lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$
$\langle proof \rangle$

lemma *INF-nat*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall m.\ \exists n.\ m<n \wedge P\ n)$
$\langle proof \rangle$

lemma *INF-nat-le*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall m.\ \exists n.\ m\leq n \wedge P\ n)$
$\langle proof \rangle$

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

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

22.3 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*⟩

22.4 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* $\Rightarrow$ *bool* **where**
atmost-one S = ($\forall$ *x y*. *x*$\in$*S* $\wedge$ *y*$\in$*S* $\longrightarrow$ *x*=*y*)

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

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

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

end

23 Multiset: Multisets

theory *Multiset*
imports *Main*
begin

23.1 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**
{#*a*#} = *Abs-multiset* (λ*b*. *if b* = *a then 1 else 0*)

definition
count :: *'a multiset* => *'a* => *nat* **where**
count = *Rep-multiset*

definition
MCollect :: *'a multiset* => (*'a* => *bool*) => *'a multiset* **where**
MCollect M P = *Abs-multiset* (λ*x*. *if P x then Rep-multiset M x else 0*)

abbreviation
Melem :: *'a* => *'a multiset* => *bool* ((-/ :# -) [*50*, *51*] *50*) **where**
a :# *M* == *count M a* > *0*

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*}

instance *multiset* :: (*type*) {*plus*, *minus*, *zero*, *size*}
union-def: *M* + *N* == *Abs-multiset* (λ*a*. *Rep-multiset M a* + *Rep-multiset N a*)
diff-def: *M* − *N* == *Abs-multiset* (λ*a*. *Rep-multiset M a* − *Rep-multiset N a*)
Zero-multiset-def [*simp*]: *0* == {#}
size-def: *size M* == *setsum* (*count M*) (*set-of M*) ⟨*proof*⟩

definition
multiset-inter :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* (**infixl** #∩ *70*) **where**
multiset-inter A B = *A* − (*A* − *B*)

Preservation of the representing set *multiset*.

lemma *const0-in-multiset* [*simp*]: (λ*a*. *0*) ∈ *multiset*
⟨*proof*⟩

lemma *only1-in-multiset* [*simp*]: (λ*b*. *if b* = *a then 1 else 0*) ∈ *multiset*
⟨*proof*⟩

lemma *union-preserves-multiset* [*simp*]:
M ∈ *multiset* ==> *N* ∈ *multiset* ==> (λ*a*. *M a* + *N a*) ∈ *multiset*

⟨*proof*⟩

lemma *diff-preserves-multiset* [*simp*]:
$M \in multiset ==> (\lambda a.\ M\ a - N\ a) \in multiset$
⟨*proof*⟩

23.2 Algebraic properties of multisets

23.2.1 Union

lemma *union-empty* [*simp*]: $M + \{\#\} = M \wedge \{\#\} + M = M$
⟨*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*⟩

23.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
⟨*proof*⟩

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
⟨*proof*⟩

23.2.3 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*⟩

23.2.4 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* $M \cup$ *set-of* N
⟨*proof*⟩

lemma *set-of-eq-empty-iff* [*simp*]: $($*set-of* $M = \{\}) = (M = \{\#\})$
⟨*proof*⟩

lemma *mem-set-of-iff* [*simp*]: $(x \in$ *set-of* $M) = (x :\# M)$
⟨*proof*⟩

23.2.5 Size

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

lemma *size-single* [*simp*]: *size* $\{\#b\#\} = 1$
⟨*proof*⟩

lemma *finite-set-of* [*iff*]: *finite* (*set-of* M)
⟨*proof*⟩

lemma *setsum-count-Int*:
finite A ==> *setsum* (*count* N) $(A \cap$ *set-of* $N)$ = *setsum* (*count* N) A
⟨*proof*⟩

lemma *size-union* [*simp*]: *size* $(M + N::'a$ *multiset*$)$ = *size* M + *size* N
⟨*proof*⟩

lemma *size-eq-0-iff-empty* [*iff*]: $($*size* $M = 0) = (M = \{\#\})$
⟨*proof*⟩

lemma *size-eq-Suc-imp-elem*: *size* M = *Suc* n ==> $\exists\, a.\ a :\# M$
⟨*proof*⟩

23.2.6 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*:
$(\{\#a\#\} = M + N) = (\{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N)$
⟨*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*⟩

23.2.7 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-union-diff-commute*: $B \ \#\cap\ C = \{\#\} \Longrightarrow A + B - C = A - C + B$
⟨*proof*⟩

23.3 Induction over multisets

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 *MCollect-preserves-multiset*:
$M \in multiset ==> (\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
⟨*proof*⟩

lemma *count-MCollect* [*simp*]:
$count\ \{\#\ x{:}M.\ P\ x\ \#\}\ a = (if\ P\ a\ then\ count\ M\ a\ else\ 0)$
⟨*proof*⟩

lemma *set-of-MCollect* [*simp*]: $set\text{-}of\ \{\#\ x{:}M.\ P\ x\ \#\} = set\text{-}of\ M \cap \{x.\ P\ x\}$
⟨*proof*⟩

lemma *multiset-partition*: $M = \{\#\ x{:}M.\ P\ x\ \#\} + \{\#\ x{:}M.\ \neg\ P\ x\ \#\}$
⟨*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*⟩

declare *multiset-typedef* [*simp del*]

23.4 Multiset orderings

23.4.1 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*⟩

23.4.2 Closure-free presentation

lemma *diff-union-single-conv*: $a :\#\ J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

One direction.

lemma *mult-implies-one-step*:
$trans\ r \Longrightarrow (M, N) \in mult\ r \Longrightarrow$
$\exists\, I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge$
$(\forall\, k \in set\text{-}of\ K.\ \exists\, j \in set\text{-}of\ J.\ (k, j) \in r)$
⟨*proof*⟩

lemma *elem-imp-eq-diff-union*: $a :\#\ M \Longrightarrow M = M - \{\#a\#\} + \{\#a\#\}$

$\langle proof \rangle$

lemma *size-eq-Suc-imp-eq-union*: $size\ M = Suc\ n ==> \exists\, a\ N.\ M = N + \{\#a\#\}$
$\langle proof \rangle$

lemma *one-step-implies-mult-aux*:
$trans\ r ==>$
$\forall\, I\ J\ K.\ (size\ J = n \wedge J \neq \{\#\} \wedge (\forall\, k \in set\text{-}of\ K.\ \exists\, j \in set\text{-}of\ J.\ (k, j) \in r))$
$--> (I + K, I + J) \in mult\ r$
$\langle proof \rangle$

lemma *one-step-implies-mult*:
$trans\ r ==> J \neq \{\#\} ==> \forall\, k \in set\text{-}of\ K.\ \exists\, j \in set\text{-}of\ J.\ (k, j) \in r$
$==> (I + K, I + J) \in mult\ r$
$\langle proof \rangle$

23.4.3 Partial-order properties

instance *multiset* :: (*type*) *ord* $\langle proof \rangle$

defs (overloaded)
less-multiset-def: $M' < M == (M', M) \in mult\ \{(x', x).\ x' < x\}$
le-multiset-def: $M' <= M == M' = M \vee M' < (M::'a\ multiset)$

lemma *trans-base-order*: $trans\ \{(x', x).\ x' < (x::'a::order)\}$
$\langle proof \rangle$

Irreflexivity.

lemma *mult-irrefl-aux*:
$finite\ A ==> (\forall\, x \in A.\ \exists\, y \in A.\ x < (y::'a::order)) \Longrightarrow A = \{\}$
$\langle proof \rangle$

lemma *mult-less-not-refl*: $\neg\ M < (M::'a::order\ multiset)$
$\langle proof \rangle$

lemma *mult-less-irrefl* [*elim*!]: $M < (M::'a::order\ multiset) ==> R$
$\langle proof \rangle$

Transitivity.

theorem *mult-less-trans*: $K < M ==> M < N ==> K < (N::'a::order\ multiset)$
$\langle proof \rangle$

Asymmetry.

theorem *mult-less-not-sym*: $M < N ==> \neg\ N < (M::'a::order\ multiset)$
$\langle proof \rangle$

theorem *mult-less-asym*:
$M < N ==> (\neg\ P ==> N < (M::'a::order\ multiset)) ==> P$
$\langle proof \rangle$

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*⟩

Partial order.

instance *multiset* :: (*order*) *order*
⟨*proof*⟩

23.4.4 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*⟩

23.5 Link with lists

consts
  multiset-of :: *'a list* ⇒ *'a multiset*
primrec
  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*: $x \in$ *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*]:
$multiset\text{-}of\ [x \leftarrow xs.\ P\ x] + multiset\text{-}of\ [x \leftarrow xs.\ \neg P\ x] = multiset\text{-}of\ xs$
⟨*proof*⟩

lemma *count-filter*:
$count\ (multiset\text{-}of\ xs)\ x = length\ [y \leftarrow xs.\ y = x]$
⟨*proof*⟩

23.6 Pointwise ordering induced by count

definition
$mset\text{-}le :: 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ (**infix** $\leq\#$ *50*) **where**
$(A \leq\# B) = (\forall a.\ count\ A\ a \leq count\ B\ a)$
definition
$mset\text{-}less :: 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ (**infix** $<\#$ *50*) **where**
$(A <\# B) = (A \leq\# B \wedge A \neq B)$

lemma *mset-le-refl*[*simp*]: $A \leq\# A$
⟨*proof*⟩

lemma *mset-le-trans*: $[\![A \leq\# B;\ B \leq\# C]\!] \Longrightarrow A \leq\# C$
⟨*proof*⟩

lemma *mset-le-antisym*: $[\![A \leq\# B;\ B \leq\# A]\!] \Longrightarrow A = B$
⟨*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 *multiset-of-remdups-le*: $multiset\text{-}of\ (remdups\ xs) \leq\# multiset\text{-}of\ xs$
⟨*proof*⟩

interpretation *mset-order*:
order [*op* $\leq$# *op* <#]
⟨*proof*⟩

interpretation *mset-order-cancel-semigroup*:
pordered-cancel-ab-semigroup-add [*op* $\leq$# *op* <# *op* +]
⟨*proof*⟩

interpretation *mset-order-semigroup-cancel*:
pordered-ab-semigroup-add-imp-le [*op* $\leq$# *op* <# *op* +]
⟨*proof*⟩

end

24 NatPair: Pairs of Natural Numbers

theory *NatPair*
imports *Main*
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*) $\Rightarrow$ *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* $\leq$ *x*+*y*
⟨*proof*⟩

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

end

25 Nat-Infinity: Natural numbers with infinity

theory *Nat-Infinity*
imports *Main*
begin

25.1 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 (∞)

instance *inat* :: {*ord*, *zero*} ⟨*proof*⟩

definition
iSuc :: *inat* => *inat* **where**
iSuc i = (*case i of Fin n* => *Fin* (*Suc n*) | ∞ => ∞)

defs (**overloaded**)
Zero-inat-def: *0* == *Fin 0*
iless-def: *m* < *n* ==
case m of Fin m1 => (*case n of Fin n1* => *m1* < *n1* | ∞ => *True*)
| ∞ => *False*
ile-def: (*m*::*inat*) $\leq$ *n* == $\neg$ (*n* < *m*)

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.

25.2 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*⟩

25.3 Ordering relations

lemma *Infty-ilessE* [*elim*!]: $\infty < Fin\ m ==> R$
⟨*proof*⟩

lemma *iless-linear*: $m < n \vee m = n \vee n < (m::inat)$
⟨*proof*⟩

lemma *iless-not-refl* [*simp*]: $\neg\ n < (n::inat)$
⟨*proof*⟩

lemma *iless-trans*: $i < j ==> j < k ==> i < (k::inat)$
⟨*proof*⟩

lemma *iless-not-sym*: $n < m ==> \neg\ m < (n::inat)$
⟨*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*: $n < Fin\ m ==> \exists\, k.\ n = Fin\ k$
⟨*proof*⟩

lemma *iSuc-mono* [*simp*]: $(iSuc\ n < iSuc\ m) = (n < m)$
⟨*proof*⟩

lemma *ile-def2*: $(m \le n) = (m < n \vee m = (n{::}inat))$
$\langle proof \rangle$

lemma *ile-refl* [*simp*]: $n \le (n{::}inat)$
$\langle proof \rangle$

lemma *ile-trans*: $i \le j$ ==> $j \le k$ ==> $i \le (k{::}inat)$
$\langle proof \rangle$

lemma *ile-iless-trans*: $i \le j$ ==> $j < k$ ==> $i < (k{::}inat)$
$\langle proof \rangle$

lemma *iless-ile-trans*: $i < j$ ==> $j \le k$ ==> $i < (k{::}inat)$
$\langle proof \rangle$

lemma *Infty-ub* [*simp*]: $n \le \infty$
$\langle proof \rangle$

lemma *i0-lb* [*simp*]: $(0{::}inat) \le n$
$\langle proof \rangle$

lemma *Infty-ileE* [*elim*!]: $\infty \le Fin\ m$ ==> R
$\langle proof \rangle$

lemma *Fin-ile-mono* [*simp*]: $(Fin\ n \le Fin\ m) = (n \le m)$
$\langle proof \rangle$

lemma *ilessI1*: $n \le m$ ==> $n \neq m$ ==> $n < (m{::}inat)$
$\langle proof \rangle$

lemma *ileI1*: $m < n$ ==> $iSuc\ m \le n$
$\langle proof \rangle$

lemma *Suc-ile-eq*: $(Fin\ (Suc\ m) \le n) = (Fin\ m < n)$
$\langle proof \rangle$

lemma *iSuc-ile-mono* [*simp*]: $(iSuc\ n \le iSuc\ m) = (n \le m)$
$\langle proof \rangle$

lemma *iless-Suc-eq* [*simp*]: $(Fin\ m < iSuc\ n) = (Fin\ m \le n)$
$\langle proof \rangle$

lemma *not-iSuc-ilei0* [*simp*]: $\neg\ iSuc\ n \le 0$
$\langle proof \rangle$

lemma *ile-iSuc* [*simp*]: $n \le iSuc\ n$
$\langle proof \rangle$

lemma *Fin-ile*: $n \le Fin\ m$ ==> $\exists\, k.\ n = Fin\ k$

⟨*proof*⟩

lemma *chain-incr*: $\forall\, i.\ \exists\, j.\ Y\ i < Y\ j \implies \exists\, j.\ Fin\ k < Y\ j$
⟨*proof*⟩

end

26 Nested-Environment: Nested environments

theory *Nested-Environment*
imports *Main*
begin

Consider a partial function $e :: {'a} => {'b}\ option$; this may be understood as an *environment* mapping indexes $'a$ to optional entry values $'b$ (cf. the basic theory *Map* of Isabelle/HOL). This basic idea is easily generalized to that of a *nested environment*, where entries may be either basic values or again proper environments. Then each entry is accessed by a *path*, i.e. a list of indexes leading to its position within the structure.

datatype ('a, 'b, 'c) *env* =
  Val 'a
  | *Env 'b 'c => ('a, 'b, 'c) env option*

In the type ('a, 'b, 'c) *env* the parameter 'a refers to basic values (occurring in terminal positions), type 'b to values associated with proper (inner) environments, and type 'c with the index type for branching. Note that there is no restriction on any of these types. In particular, arbitrary branching may yield rather large (transfinite) tree structures.

26.1 The lookup operation

Lookup in nested environments works by following a given path of index elements, leading to an optional result (a terminal value or nested environment). A *defined position* within a nested environment is one where *lookup* at its path does not yield *None*.

consts
  lookup :: ('a, 'b, 'c) *env* => 'c *list* => ('a, 'b, 'c) *env option*
  lookup-option :: ('a, 'b, 'c) *env option* => 'c *list* => ('a, 'b, 'c) *env option*

primrec (*lookup*)
  lookup (*Val a*) *xs* = (*if xs* = [] *then Some* (*Val a*) *else None*)
  lookup (*Env b es*) *xs* =
    (*case xs of*
      [] => *Some* (*Env b es*)
    | *y* # *ys* => *lookup-option* (*es y*) *ys*)

  lookup-option None xs = None
  lookup-option (Some e) xs = lookup e xs

hide *const lookup-option*

The characteristic cases of *lookup* are expressed by the following equalities.

theorem *lookup-nil*: *lookup e [] = Some e*
  ⟨*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 $\exists\, e.\ lookup\ env\ xs = Some\ e$
$\langle proof \rangle$

The subsequent statement describes in more detail how a successful *lookup* with a non-empty path results in a certain situation at any upper position.

theorem *lookup-some-upper*:
  assumes $lookup\ env\ (xs\ @\ y\ \#\ ys) = Some\ e$
  shows $\exists\, b'\ es'\ env'.$
    $lookup\ env\ xs = Some\ (Env\ b'\ es') \wedge$
    $es'\ y = Some\ env' \wedge$
    $lookup\ env'\ ys = Some\ e$
  $\langle proof \rangle$

26.2 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
  $update :: 'c\ list => ('a, 'b, 'c)\ env\ option$
    $=> ('a, 'b, 'c)\ env => ('a, 'b, 'c)\ env$
  $update\text{-}option :: 'c\ list => ('a, 'b, 'c)\ env\ option$
    $=> ('a, 'b, 'c)\ env\ option => ('a, 'b, 'c)\ env\ option$

primrec (*update*)
  $update\ xs\ opt\ (Val\ a) =$
    $(if\ xs = []\ then\ (case\ opt\ of\ None => Val\ a\ |\ Some\ e => e)$
    $else\ Val\ a)$
  $update\ xs\ opt\ (Env\ b\ es) =$
    $(case\ xs\ of$
      $[] => (case\ opt\ of\ None => Env\ b\ es\ |\ Some\ e => e)$
    $|\ y\ \#\ ys => Env\ b\ (es\ (y := update\text{-}option\ ys\ opt\ (es\ y))))$
  $update\text{-}option\ xs\ opt\ None =$
    $(if\ xs = []\ then\ opt\ else\ None)$
  $update\text{-}option\ xs\ opt\ (Some\ e) =$
    $(if\ xs = []\ then\ opt\ else\ Some\ (update\ xs\ opt\ e))$

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*: $y \neq (z{::}'c)$
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 $e1 = e2 \longleftrightarrow e1 = e2$ ⟨*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 $Env\ x\ f = Env\ y\ g \longleftrightarrow$
$x = y \land (\forall z \in UNIV.$ *case f z*
of None ⇒ (*case g z*
of None ⇒ *True* | *Some* - ⇒ *False*)
| *Some a* ⇒ (*case g z*
of None ⇒ *False* | *Some b* ⇒ *a* = *b*)) (**is** *?env*)
and $Val\ a = Val\ b \longleftrightarrow a = b$
and $Val\ a = Env\ y\ g \longleftrightarrow False$
and $Env\ x\ f = Val\ b \longleftrightarrow False$
⟨*proof*⟩

end

27 Numeral-Type: Numeral Syntax for Types

theory *Numeral-Type*
  imports *Infinite-Set*
begin

27.1 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*⟩

27.2 Cardinalities of types

syntax *-type-card* :: *type* => *nat* ((*1CARD*/(*1*′(-′))))

translations *CARD*(*t*) => *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*: $CARD('a::finite\ set) = 2\ \hat{}\ CARD('a)$
⟨*proof*⟩

27.3 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

27.4 Syntax

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

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

⟨*ML*⟩

27.5 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*⟩

27.6 Examples

term *TYPE*(*10*)

lemma *CARD*(*0*) = *0* ⟨*proof*⟩
lemma *CARD*(*17*) = *17* ⟨*proof*⟩

end

28 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*⟩

28.1 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*⟩

28.2 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*⟩

28.3 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*⟩

28.4 Removing elements

consts
remove :: *'a* => *'a list* => *'a list*
primrec
remove x [] = []
remove x (*y* # *ys*) = (*if x* = *y then ys else y* # *remove x ys*)

lemma *perm-remove*: *x* ∈ *set ys* ==> *ys* <~~> *x* # *remove x ys*
⟨*proof*⟩

lemma *remove-commute*: *remove x* (*remove y l*) = *remove y* (*remove x l*)
⟨*proof*⟩

lemma *multiset-of-remove* [*simp*]:
multiset-of (*remove a x*) = *multiset-of x* − {#*a*#}
⟨*proof*⟩

Congruence rule

lemma *perm-remove-perm*: *xs* <~~> *ys* ==> *remove z xs* <~~> *remove z ys*
⟨*proof*⟩

lemma *remove-hd* [*simp*]: *remove z* (*z* # *xs*) = *xs*
⟨*proof*⟩

lemma *cons-perm-imp-perm*: *z* # *xs* <~~> *z* # *ys* ==> *xs* <~~> *ys*
⟨*proof*⟩

lemma *cons-perm-eq* [*iff*]: (*z*#*xs* <~~> *z*#*ys*) = (*xs* <~~> *ys*)
⟨*proof*⟩

lemma *append-perm-imp-perm*: *zs* @ *xs* <~~> *zs* @ *ys* ==> *xs* <~~> *ys*
⟨*proof*⟩

lemma *perm-append1-eq* [*iff*]: (*zs* @ *xs* <~~> *zs* @ *ys*) = (*xs* <~~> *ys*)
⟨*proof*⟩

lemma *perm-append2-eq* [*iff*]: (*xs* @ *zs* <~~> *ys* @ *zs*) = (*xs* <~~> *ys*)
⟨*proof*⟩

lemma *multiset-of-eq-perm*: (*multiset-of xs* = *multiset-of ys*) = (*xs* <~~> *ys*)
⟨*proof*⟩

lemma *multiset-of-le-perm-append*:
(*multiset-of xs* ≤# *multiset-of ys*) = (∃ *zs*. *xs* @ *zs* <~~> *ys*)
⟨*proof*⟩

lemma *perm-set-eq*: *xs* <~~> *ys* ==> *set xs* = *set ys*
⟨*proof*⟩

lemma *perm-distinct-iff*: *xs* <~~> *ys* ==> *distinct xs* = *distinct ys*
⟨*proof*⟩

lemma *eq-set-perm-remdups*: *set xs* = *set ys* ==> *remdups xs* <~~> *remdups ys*
⟨*proof*⟩

lemma *perm-remdups-iff-eq-set*: *remdups x* <~~> *remdups y* = (*set x* = *set y*)
⟨*proof*⟩

end

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

theory *Code-Char*
imports *List*
begin

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

⟨*ML*⟩

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

code-reserved *SML*
char

code-reserved *OCaml*
char

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

end

30 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

31 Primes: Primality on nat

theory *Primes*
imports *GCD*
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*ˆ*Suc* (*Suc 0*) ==> *p dvd m*
⟨*proof*⟩

lemma *prime-dvd-power-two*: *prime p* ==> *p dvd* m^2 ==> *p dvd m*
⟨*proof*⟩

end

32 Quicksort: Quicksort

theory *Quicksort*
imports *Multiset*
begin

context *linorder*
begin

function *quicksort* :: *'a list* ⇒ *'a list* **where**
quicksort [] = [] |
quicksort (*x*#*xs*) = *quicksort*([*y*←*xs*. ~ *x*≤*y*]) @ [*x*] @ *quicksort*([*y*←*xs*. *x*≤*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

33 Quotient: Quotient types

theory *Quotient*
imports *Main*
begin

We introduce the notion of quotient types over equivalence relations via type classes.

33.1 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*]: $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$
assumes *equiv-sym* [*sym*]: $x \sim y \Longrightarrow y \sim x$

lemma *equiv-not-sym* [*sym*]: $\neg\ (x \sim y) ==> \neg\ (y \sim (x::'a::equiv))$
⟨*proof*⟩

lemma *not-equiv-trans1* [*trans*]: $\neg\ (x \sim y) ==> y \sim z ==> \neg\ (x \sim (z::'a::equiv))$
⟨*proof*⟩

lemma *not-equiv-trans2* [*trans*]: $x \sim y ==> \neg\ (y \sim z) ==> \neg\ (x \sim (z::'a::equiv))$
⟨*proof*⟩

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

typedef $'a\ quot = \{\{x.\ a \sim x\} \mid a::'a::eqv.\ True\}$
⟨*proof*⟩

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

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

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

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

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

lemma *quot-cases* [*cases type*: *quot*]: $(!!a.\ A = \lfloor a \rfloor ==> C) ==> C$
⟨*proof*⟩

33.2 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*⟩

33.3 Picking representing elements

definition
pick :: $'a::equiv\ quot => 'a$ **where**
$pick\ A = (SOME\ a.\ A = \lfloor a \rfloor)$

theorem *pick-equiv* [*intro*]: $pick\ \lfloor a \rfloor \sim a$

⟨*proof*⟩

theorem *pick-inverse* [*intro*]: $\lfloor pick\ A\rfloor = A$
⟨*proof*⟩

The following rules support canonical function definitions on quotient types (with up to two arguments). Note that the stripped-down version without additional conditions is sufficient most of the time.

theorem *quot-cond-function*:
  assumes *eq*: $!!X\ Y.\ P\ X\ Y ==> f\ X\ Y == g\ (pick\ X)\ (pick\ Y)$
    and *cong*: $!!x\ x'\ y\ y'.\ \lfloor x\rfloor = \lfloor x'\rfloor ==> \lfloor y\rfloor = \lfloor y'\rfloor$
      $==> P\ \lfloor x\rfloor\ \lfloor y\rfloor ==> P\ \lfloor x'\rfloor\ \lfloor y'\rfloor ==> g\ x\ y = g\ x'\ y'$
    and *P*: $P\ \lfloor a\rfloor\ \lfloor b\rfloor$
  shows $f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ a\ b$
⟨*proof*⟩

theorem *quot-function*:
  assumes $!!X\ Y.\ f\ X\ Y == g\ (pick\ X)\ (pick\ Y)$
    and $!!x\ x'\ y\ y'.\ \lfloor x\rfloor = \lfloor x'\rfloor ==> \lfloor y\rfloor = \lfloor y'\rfloor ==> g\ x\ y = g\ x'\ y'$
  shows $f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ a\ b$
  ⟨*proof*⟩

theorem *quot-function'*:
  $(!!X\ Y.\ f\ X\ Y == g\ (pick\ X)\ (pick\ Y)) ==>$
    $(!!x\ x'\ y\ y'.\ x \sim x' ==> y \sim y' ==> g\ x\ y = g\ x'\ y') ==>$
    $f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ a\ b$
  ⟨*proof*⟩

end

34 Ramsey: Ramsey's Theorem

theory *Ramsey* **imports** *Main Infinite-Set* **begin**

34.1 Preliminaries

34.1.1 "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 $P\ (choice\ P\ r\ n)$
$\langle proof \rangle$

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** $!!n\ m.\ n < m ==> (f\ n,\ f\ m) \in r$
$\langle proof \rangle$

34.1.2 Partitions of a Set

definition
$part :: nat => nat => {}'a\ set => ({}'a\ set => nat) => bool$
— the function f partitions the r-subsets of the typically infinite set Y into s distinct categories.
where
$part\ r\ s\ Y\ f = (\forall\, X.\ X \subseteq Y\ \&\ finite\ X\ \&\ card\ X = r \dashrightarrow f\ X < s)$

For induction, we decrease the value of r in partitions.

lemma *part-Suc-imp-part*:
$[|\ infinite\ Y;\ part\ (Suc\ r)\ s\ Y\ f;\ y \in Y\ |]$
$==> part\ r\ s\ (Y - \{y\})\ (\%u.\ f\ (insert\ y\ u))$
$\langle proof \rangle$

lemma *part-subset*: $part\ r\ s\ YY\ f ==> Y \subseteq YY ==> part\ r\ s\ Y\ f$
$\langle proof \rangle$

34.2 Ramsey's Theorem: Infinitary Version

lemma *Ramsey-induction*:
fixes s **and** $r::nat$
shows
$!!(YY::{}'a\ set)\ (f::{}'a\ set => nat).$
$[|infinite\ YY;\ part\ r\ s\ YY\ f|]$
$==> \exists\, Y'\ t'.\ Y' \subseteq YY\ \&\ infinite\ Y'\ \&\ t' < s\ \&$
$(\forall\, X.\ X \subseteq Y'\ \&\ finite\ X\ \&\ card\ X = r \dashrightarrow f\ X = t')$
$\langle proof \rangle$

theorem *Ramsey*:
fixes $s\ r :: nat$ **and** $Z::{}'a\ set$ **and** $f::{}'a\ set => nat$
shows
$[|infinite\ Z;$
$\forall\, X.\ X \subseteq Z\ \&\ finite\ X\ \&\ card\ X = r \dashrightarrow f\ X < s|]$
$==> \exists\, Y\ t.\ Y \subseteq Z\ \&\ infinite\ Y\ \&\ t < s$
$\&\ (\forall\, X.\ X \subseteq Y\ \&\ finite\ X\ \&\ card\ X = r \dashrightarrow f\ X = t)$
$\langle proof \rangle$

corollary *Ramsey2*:
fixes s::*nat* **and** Z::*'a set* **and** f::*'a set => nat*
assumes *infZ*: *infinite Z*
and *part*: $\forall\, x{\in}Z.\ \forall\, y{\in}Z.\ x{\neq}y \dashrightarrow f\{x,y\} < s$
shows
$\exists\, Y\ t.\ Y \subseteq Z\ \&\ infinite\ Y\ \&\ t < s\ \&\ (\forall\, x{\in}Y.\ \forall\, y{\in}Y.\ x{\neq}y \dashrightarrow f\{x,y\} = t)$
⟨*proof*⟩

34.3 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [5].

definition
disj-wf :: *('a * 'a)set => 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 => 'a, nat => ('a*'a)set, nat set] => nat*
where
transition-idx s T A =
$(LEAST\ k.\ \exists\, i\ j.\ A = \{i,j\}\ \&\ i{<}j\ \&\ (s\ j,\ s\ i) \in T\ k)$

lemma *transition-idx-less*:
$[\![i{<}j;\ (s\ j,\ s\ i) \in T\ k;\ k{<}n]\!] \Longrightarrow transition\text{-}idx\ s\ T\ \{i,j\} < n$
⟨*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

35 State-Monad: Combinators syntax for generic, open state monads (single threaded monads)

theory *State-Monad*
imports *Main*
begin

35.1 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.

35.2 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:

definition
$mbind :: ('a \Rightarrow 'b \times 'c) \Rightarrow ('b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$
(**infixl** $>>=$ *60*) **where**
$f >>= g = split\ g \circ f$

definition
$fcomp :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'c$
(**infixl** $>>$ *60*) **where**
$f >> g = g \circ f$

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

syntax (*xsymbols*)
  $mbind :: ('a \Rightarrow 'b \times 'c) \Rightarrow ('b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$
    (**infixl** $\gg=$ *60*)
  $fcomp :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'c$
    (**infixl** $\gg$ *60*)

abbreviation (*input*)
  $return \equiv Pair$

⟨*ML*⟩

Given two transformations f and g, they may be directly composed using the *op* >> combinator, forming a forward composition: $(f >> g)\ s = f\ (g\ s)$.

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the *op* >>= combinator: $(f >>= (\lambda x.\ g))\ s = (let\ (x,\ s') = f\ s\ in\ g\ s')$.

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 has to (or may) be specified completely independent.
- 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.

35.3 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*⟩

35.4 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

lemma
return-mbind [*simp*]: $return\ x \gg= f = f\ x$
⟨*proof*⟩

lemma
mbind-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
mbind-mbind [*simp*]: $(f \gg= g) \gg= h = f \gg= (\lambda x.\ g\ x \gg= h)$
⟨*proof*⟩

lemma
mbind-fcomp [*simp*]: $(f \gg= g) \gg h = f \gg= (\lambda x.\ g\ x \gg h)$
⟨*proof*⟩

lemma
fcomp-mbind [*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-mbind mbind-return id-fcomp fcomp-id mbind-mbind mbind-fcomp fcomp-mbind fcomp-fcomp*

Evaluation of monadic expressions by force:

lemmas *monad-collapse* = *monad-simp o-apply o-assoc split-Pair split-comp* *mbind-def fcomp-def run-def*

35.5 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*)
-mbind :: *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*)
-mbind :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(- ← -;// - [*1000*, *13*, *12*] *12*)

translations
-do f => *CONST run f*
-mbind x f g => *f* ≫= (λ*x*. *g*)
-fcomp f g => *f* ≫ *g*
-let x t f => *CONST Let t* (λ*x*. *f*)
-nil f => *f*

⟨*ML*⟩

35.6 Combinators

definition
lift :: (*'a* ⇒ *'b*) ⇒ *'a* ⇒ *'c* ⇒ *'b* × *'c* **where**
lift f x = *return* (*f x*)

fun
list :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a list* ⇒ *'b* ⇒ *'b* **where**
list f [] = *id*
| *list f* (*x*#*xs*) = (*do f x*; *list f xs done*)

fun *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*)

combinator properties

lemma *list-append* [*simp*]:
  $list\ f\ (xs\ @\ ys) = list\ f\ xs \gg list\ f\ ys$
  ⟨*proof*⟩

lemma *list-cong* [*fundef-cong*, *recdef-cong*]:
  $[\![\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x;\ xs = ys]\!]$
    $\Longrightarrow list\ f\ xs = list\ g\ ys$
⟨*proof*⟩

lemma *list-yield-cong* [*fundef-cong*, *recdef-cong*]:
  $[\![\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x;\ xs = ys]\!]$
    $\Longrightarrow list\text{-}yield\ f\ xs = list\text{-}yield\ g\ ys$
⟨*proof*⟩

still waiting for extensions. . .

For an example, see HOL/ex/Random.thy.

end

36 While-Combinator: A general "while" combinator

theory *While-Combinator*
imports *Main*
begin

We define the while combinator as the "mother of all tail recursive functions".

function (*tailrec*) *while* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
where
  while-unfold[*simp del*]: $while\ b\ c\ s = (if\ b\ s\ then\ while\ b\ c\ (c\ s)\ else\ s)$
⟨*proof*⟩

declare *while-unfold*[*code*]

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

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

theorem *while-rule-lemma*:
  assumes *invariant*: $!!s.\ P\ s ==> b\ s ==> P\ (c\ s)$
    and *terminate*: $!!s.\ P\ s ==> \neg\ b\ s ==> Q\ s$
    and *wf*: $wf\ \{(t,\ s).\ P\ s \wedge b\ s \wedge t = c\ s\}$
  shows $P\ s \Longrightarrow Q\ (while\ b\ c\ s)$
  ⟨*proof*⟩

theorem *while-rule*:
[| $P\ s$;
!!s. [| $P\ s$; $b\ s$ |] ==> $P\ (c\ s)$;
!!s. [| $P\ s$; $\neg\ b\ s$ |] ==> $Q\ s$;
wf r;
!!s. [| $P\ s$; $b\ s$ |] ==> $(c\ s,\ s) \in r$ |] ==>
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* $(\lambda(A, fA).\ A \neq fA)$ $(\lambda(A, fA).\ (fA, f\ fA))$ $(\{\}, f\ \{\})$)
⟨*proof*⟩

An example of using the *while* combinator.

Cannot use *set-eq-subset* because it leads to looping because the antisymmetry simproc turns the subset relationship back into equality.

theorem P (*lfp* $(\lambda N{::}int\ set.\ \{0\} \cup \{(n + 2)\ mod\ 6 \mid n.\ n \in N\})) =$
$P\ \{0, 4, 2\}$
⟨*proof*⟩

end

37 Word: Binary Words

theory *Word*
imports *Main*
begin

37.1 Auxilary Lemmas

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

lemma *max-mono*:
fixes $x :: {'}a{::}linorder$
assumes *mf*: *mono* f
shows $max\ (f\ x)\ (f\ y) \le f\ (max\ x\ y)$
⟨*proof*⟩

declare *zero-le-power* [*intro*]
and *zero-less-power* [*intro*]

lemma *int-nat-two-exp*: $2\ \hat{}\ k = int\ (2\ \hat{}\ k)$
⟨*proof*⟩

37.2 Bits

datatype *bit* =
  Zero (**0**)
| *One* (**1**)

consts
bitval :: *bit* => *nat*
primrec
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*⟩

37.3 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* ˆ *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* ˆ (*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*: $n \leq 2 \mathbin{\hat{}} k - 1$
shows *length* (*nat-to-bv n*) $\leq k$
⟨*proof*⟩

lemma *length-nat-to-bv-lower-limit*:
assumes *nk*: $2 \mathbin{\hat{}} k \leq n$
shows $k <$ *length* (*nat-to-bv n*)
⟨*proof*⟩

37.4 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*) $\leq$ *length w1* + *length w2*
⟨*proof*⟩

37.5 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* $0 \leq n$
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*]: $0 \leq n$ ==> *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*: $bv\text{-}to\text{-}int\ xs = bv\text{-}to\text{-}int\ ys$
and *len*: $length\ xs = length\ ys$
shows $xs = ys$
⟨*proof*⟩

lemma *int-to-bv-returntype* [*simp*]: $norm\text{-}signed\ (int\text{-}to\text{-}bv\ w) = int\text{-}to\text{-}bv\ w$
⟨*proof*⟩

lemma *bv-to-int-msb0*: $0 \leq bv\text{-}to\text{-}int\ w1 \implies bv\text{-}msb\ w1 = \mathbf{0}$
⟨*proof*⟩

lemma *bv-to-int-msb1*: $bv\text{-}to\text{-}int\ w1 < 0 \implies bv\text{-}msb\ w1 = \mathbf{1}$
⟨*proof*⟩

lemma *bv-to-int-lower-limit-gt0*:
assumes *w0*: $0 < bv\text{-}to\text{-}int\ w$
shows $2\ \hat{}\ (length\ (norm\text{-}signed\ w) - 2) \leq bv\text{-}to\text{-}int\ w$
⟨*proof*⟩

lemma *norm-signed-result*: $norm\text{-}signed\ w = [] \vee norm\text{-}signed\ w = [\mathbf{1}] \vee bv\text{-}msb\ (norm\text{-}signed\ w) \neq bv\text{-}msb\ (tl\ (norm\text{-}signed\ w))$
⟨*proof*⟩

lemma *bv-to-int-upper-limit-lem1*:
assumes *w0*: $bv\text{-}to\text{-}int\ w < -1$
shows $bv\text{-}to\text{-}int\ w < -\ (2\ \hat{}\ (length\ (norm\text{-}signed\ w) - 2))$
⟨*proof*⟩

lemma *length-int-to-bv-upper-limit-gt0*:
assumes *w0*: $0 < i$
and *wk*: $i \leq 2\ \hat{}\ (k - 1) - 1$
shows $length\ (int\text{-}to\text{-}bv\ i) \leq k$
⟨*proof*⟩

lemma *pos-length-pos*:
assumes *i0*: $0 < bv\text{-}to\text{-}int\ w$
shows $0 < length\ w$
⟨*proof*⟩

lemma *neg-length-pos*:
assumes *i0*: $bv\text{-}to\text{-}int\ w < -1$
shows $0 < length\ w$
⟨*proof*⟩

lemma *length-int-to-bv-lower-limit-gt0*:
assumes *wk*: $2\ \hat{}\ (k - 1) \leq i$
shows $k < length\ (int\text{-}to\text{-}bv\ i)$

⟨*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*⟩

37.6 Signed Arithmetic Operations

37.6.1 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*⟩

37.6.2 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 $((2::int)\ \hat{}\ (length\ w1 - 1)) + (2\ \hat{}\ (length\ w2 - 1)) \leq 2\ \hat{}\ max\ (length\ w1)\ (length\ w2)$
⟨*proof*⟩

lemma *bv-sadd-length*: $length\ (bv\text{-}sadd\ w1\ w2) \leq Suc\ (max\ (length\ w1)\ (length\ w2))$
⟨*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*: $length\ (bv\text{-}sub\ w1\ w2) \leq Suc\ (max\ (length\ w1)\ (length\ w2))$
⟨*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*⟩

37.7 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*]: [| $j \leq i$; $i < \mathit{length}\ w$ |] ==> *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*: $n \neq 0$
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*: [| $j \leq i$; $i < \mathit{length}\ w$; *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*: $k \leq i$
and *ij*: $i \leq j$
and *jl*: $j \leq l$

and *lw*: $l < length\ w$
shows *bv-slice w (j,i) = bv-slice (bv-slice w (l,k)) (j−k,i−k)*
⟨*proof*⟩

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*]: $x \leq y$ ==> *length-nat x* $\leq$ *length-nat y*
⟨*proof*⟩

lemma *length-nat-mono-int* [*simp*]: $x \leq y$ ==> *length-nat x* $\leq$ *length-nat y*
⟨*proof*⟩

lemma *length-nat-pos* [*simp*,*intro*!]: *0 < x ==> 0 < length-nat x*
⟨*proof*⟩

lemma *length-int-mono-gt0*: [| *0* $\leq$ *x* ; *x* $\leq$ *y* |] ==> *length-int x* $\leq$ *length-int y*
⟨*proof*⟩

lemma *length-int-mono-lt0*: [| *x* $\leq$ *y* ; *y* $\leq$ *0* |] ==> *length-int y* $\leq$ *length-int x*
⟨*proof*⟩

lemmas [*simp*] = *length-nat-non0*

lemma *nat-to-bv (number-of Numeral.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 (k BIT (bit-case bit.B0 bit.B1 b))

lemma *fast-bv-to-nat-Cons0*: *fast-bv-to-nat-helper* (**0**#*bs*) *bin =*
fast-bv-to-nat-helper bs (bin BIT bit.B0)
⟨*proof*⟩

lemma *fast-bv-to-nat-Cons1*: *fast-bv-to-nat-helper* (**1**#*bs*) *bin =*
fast-bv-to-nat-helper bs (bin BIT bit.B1)

⟨*proof*⟩

lemma *fast-bv-to-nat-def*:
bv-to-nat bs == *number-of* (*fast-bv-to-nat-helper bs Numeral.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

38 Zorn: Zorn's Lemma

theory *Zorn*
imports *Main*
begin

The lemma and section numbers refer to an unpublished article [1].

definition
chain :: *'a set set* => *'a set set set* **where**
chain S = $\{F.\ F \subseteq S\ \&\ (\forall x \in F.\ \forall y \in F.\ x \subseteq y \mid y \subseteq x)\}$

definition
super :: [*'a set set*,*'a set set*] => *'a set set set* **where**
super S c = $\{d.\ d \in$ *chain S* $\&\ c \subset d\}$

definition
$maxchain$ :: $'a\ set\ set => 'a\ set\ set\ set$ **where**
$maxchain\ S = \{c.\ c \in chain\ S\ \&\ super\ S\ c = \{\}\}$

definition
$succ$ :: $['a\ set\ set, 'a\ set\ set] => 'a\ set\ set$ **where**
$succ\ S\ c =$
$(if\ c \notin chain\ S \mid c \in maxchain\ S$
$then\ c\ else\ SOME\ c'.\ c' \in super\ S\ c)$

inductive-set
$TFin$ :: $'a\ set\ set => 'a\ set\ set\ set$
for S :: $'a\ set\ set$
where
$succI$: $x \in TFin\ S ==> succ\ S\ x \in TFin\ S$
| $Pow\text{-}UnionI$: $Y \in Pow(TFin\ S) ==> Union(Y) \in TFin\ S$
monos $Pow\text{-}mono$

38.1 Mathematical Preamble

lemma *Union-lemma0*:
$(\forall x \in C.\ x \subseteq A \mid B \subseteq x) ==> Union(C) \subseteq A \mid B \subseteq Union(C)$
⟨*proof*⟩

This is theorem *increasingD2* of ZF/Zorn.thy

lemma *Abrial-axiom1*: $x \subseteq succ\ S\ x$
⟨*proof*⟩

lemmas $TFin\text{-}UnionI = TFin.Pow\text{-}UnionI\ [OF\ PowI]$

lemma *TFin-induct*:
$[|\ n \in TFin\ S;$
$!!x.\ [|\ x \in TFin\ S;\ P(x)\ |] ==> P(succ\ S\ x);$
$!!Y.\ [|\ Y \subseteq TFin\ S;\ Ball\ Y\ P\ |] ==> P(Union\ Y)\ |]$
$==> P(n)$
⟨*proof*⟩

lemma *succ-trans*: $x \subseteq y ==> x \subseteq succ\ S\ y$
⟨*proof*⟩

Lemma 1 of section 3.1

lemma *TFin-linear-lemma1*:
$[|\ n \in TFin\ S;\ \ m \in TFin\ S;$
$\forall x \in TFin\ S.\ x \subseteq m --> x = m \mid succ\ S\ x \subseteq m$
$|] ==> n \subseteq m \mid succ\ S\ m \subseteq n$
⟨*proof*⟩

Lemma 2 of section 3.2

lemma *TFin-linear-lemma2*:

$m \in TFin\ S ==> \forall\, n \in TFin\ S.\ n \subseteq m --> n{=}m \mid succ\ S\ n \subseteq m$
⟨*proof*⟩

Re-ordering the premises of Lemma 2

lemma *TFin-subsetD*:
$[\mid n \subseteq m;\ \ m \in TFin\ S;\ \ n \in TFin\ S\ \mid] ==> n{=}m \mid succ\ S\ n \subseteq m$
⟨*proof*⟩

Consequences from section 3.3 – Property 3.2, the ordering is total

lemma *TFin-subset-linear*: $[\mid m \in TFin\ S;\ \ n \in TFin\ S\mid] ==> n \subseteq m \mid m \subseteq n$
⟨*proof*⟩

Lemma 3 of section 3.3

lemma *eq-succ-upper*: $[\mid n \in TFin\ S;\ \ m \in TFin\ S;\ \ m = succ\ S\ m\ \mid] ==> n \subseteq m$
⟨*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*⟩

38.2 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*: $c \in chain\ S - maxchain\ S ==>\ ?\ d.\ d \in super\ S\ c$
⟨*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*: $c \in chain\ S - maxchain\ S ==> succ\ S\ c \neq c$
⟨*proof*⟩

lemma *TFin-chain-lemma4*: $c \in TFin\ S ==> (c :: 'a\ set\ set): chain\ S$
⟨*proof*⟩

theorem *Hausdorff*: $\exists\, c.\ (c :: 'a\ set\ set): maxchain\ S$
⟨*proof*⟩

38.3 Zorn's Lemma: If All Chains Have Upper Bounds Then There Is a Maximal Element

lemma *chain-extend*:
$[|\ c \in chain\ S;\ z \in S;$
$\forall\, x \in c.\ x \subseteq (z{::}\ 'a\ set)\ |] ==> \{z\}\ Un\ c \in chain\ S$
⟨*proof*⟩

lemma *chain-Union-upper*: $[|\ c \in chain\ S;\ x \in c\ |] ==> x \subseteq Union(c)$
⟨*proof*⟩

lemma *chain-ball-Union-upper*: $c \in chain\ S ==> \forall\, x \in c.\ x \subseteq Union(c)$
⟨*proof*⟩

lemma *maxchain-Zorn*:
$[|\ c \in maxchain\ S;\ u \in S;\ Union(c) \subseteq u\ |] ==> Union(c) = u$
⟨*proof*⟩

theorem *Zorn-Lemma*:
$\forall\, c \in chain\ S.\ Union(c): S ==> \exists\, y \in S.\ \forall\, z \in S.\ y \subseteq z\ {-}{-}{>}\ y = z$
⟨*proof*⟩

38.4 Alternative version of Zorn's Lemma

lemma *Zorn-Lemma2*:
$\forall\, c \in chain\ S.\ \exists\, y \in S.\ \forall\, x \in c.\ x \subseteq y$
$==> \exists\, y \in S.\ \forall\, x \in S.\ (y :: 'a\ set) \subseteq x\ {-}{-}{>}\ y = x$
⟨*proof*⟩

Various other lemmas

lemma *chainD*: $[|\ c \in chain\ S;\ x \in c;\ y \in c\ |] ==> x \subseteq y\ |\ y \subseteq x$
⟨*proof*⟩

lemma *chainD2*: $!!(c :: 'a\ set\ set).\ c \in chain\ S ==> c \subseteq S$
⟨*proof*⟩

end

39 List-Prefix: List prefixes and postfixes

theory *List-Prefix*
imports *Main*
begin

39.1 Prefix order on lists

instance *list* :: (*type*) *ord* ⟨*proof*⟩

defs (overloaded)
  prefix-def: $xs \le ys == \exists\, zs.\ ys = xs\ @\ zs$
  strict-prefix-def: $xs < ys == xs \le ys \wedge xs \ne (ys{::}'a\ list)$

instance *list* :: (*type*) *order*
  ⟨*proof*⟩

lemma *prefixI* [*intro?*]: $ys = xs\ @\ zs ==> xs \le ys$
  ⟨*proof*⟩

lemma *prefixE* [*elim?*]:
  assumes $xs \le ys$
  obtains *zs* **where** $ys = xs\ @\ zs$
  ⟨*proof*⟩

lemma *strict-prefixI'* [*intro?*]: $ys = xs\ @\ z\ \#\ zs ==> xs < ys$
  ⟨*proof*⟩

lemma *strict-prefixE'* [*elim?*]:
  assumes $xs < ys$
  obtains *z zs* **where** $ys = xs\ @\ z\ \#\ zs$
⟨*proof*⟩

lemma *strict-prefixI* [*intro?*]: $xs \le ys ==> xs \ne ys ==> xs < (ys{::}'a\ list)$
  ⟨*proof*⟩

lemma *strict-prefixE* [*elim?*]:
  fixes *xs ys* :: *'a list*
  assumes $xs < ys$
  obtains $xs \le ys$ **and** $xs \ne ys$
  ⟨*proof*⟩

39.2 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*]: $xs \le ys ==> xs \le ys\ @\ zs$
⟨*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*:
$xs \le ys ==> length\ xs < length\ ys ==> xs\ @\ [ys\ !\ length\ xs] \le ys$
⟨*proof*⟩

theorem *prefix-length-le*: $xs \le ys ==> length\ xs \le length\ ys$
⟨*proof*⟩

lemma *prefix-same-cases*:
$(xs_1::'a\ list) \le ys \Longrightarrow xs_2 \le ys \Longrightarrow xs_1 \le xs_2 \vee xs_2 \le xs_1$
⟨*proof*⟩

lemma *set-mono-prefix*:
$xs \le ys \Longrightarrow set\ xs \subseteq set\ ys$
⟨*proof*⟩

lemma *take-is-prefix*:
$take\ n\ xs \le xs$
⟨*proof*⟩

lemma *map-prefixI*:
$xs \le ys \Longrightarrow map\ f\ xs \le map\ f\ ys$
⟨*proof*⟩

lemma *prefix-length-less*:

$xs < ys \Longrightarrow length\ xs < length\ ys$
$\langle proof \rangle$

lemma *strict-prefix-simps* [*simp*]:
$xs < [] = False$
$[] < (x \# xs) = True$
$(x \# xs) < (y \# ys) = (x = y \wedge xs < ys)$
$\langle proof \rangle$

lemma *take-strict-prefix*:
$xs < ys \Longrightarrow take\ n\ xs < ys$
$\langle proof \rangle$

lemma *not-prefix-cases*:
assumes *pfx*: $\neg\ ps \leq ls$
obtains
$(c1)$ $ps \neq []$ **and** $ls = []$
| $(c2)$ $a\ as\ x\ xs$ **where** $ps = a\#as$ **and** $ls = x\#xs$ **and** $x = a$ **and** $\neg\ as \leq xs$
| $(c3)$ $a\ as\ x\ xs$ **where** $ps = a\#as$ **and** $ls = x\#xs$ **and** $x \neq a$
$\langle proof \rangle$

lemma *not-prefix-induct* [*consumes 1*, *case-names Nil Neq Eq*]:
assumes *np*: $\neg\ ps \leq ls$
and *base*: $\bigwedge x\ xs.\ P\ (x\#xs)\ []$
and *r1*: $\bigwedge x\ xs\ y\ ys.\ x \neq y \Longrightarrow P\ (x\#xs)\ (y\#ys)$
and *r2*: $\bigwedge x\ xs\ y\ ys.\ [\![\ x = y;\ \neg\ xs \leq ys;\ P\ xs\ ys\ ]\!] \Longrightarrow P\ (x\#xs)\ (y\#ys)$
shows $P\ ps\ ls$ $\langle proof \rangle$

39.3 Parallel lists

definition
parallel :: *'a list => 'a list => bool* (**infixl** $\|$ *50*) **where**
$(xs \parallel ys) = (\neg\ xs \leq ys \wedge \neg\ ys \leq xs)$

lemma *parallelI* [*intro*]: $\neg\ xs \leq ys$ ==> $\neg\ ys \leq xs$ ==> $xs \parallel ys$
$\langle proof \rangle$

lemma *parallelE* [*elim*]:
assumes $xs \parallel ys$
obtains $\neg\ xs \leq ys \wedge \neg\ ys \leq xs$
$\langle proof \rangle$

theorem *prefix-cases*:
obtains $xs \leq ys \mid ys < xs \mid xs \parallel ys$
$\langle proof \rangle$

theorem *parallel-decomp*:
$xs \parallel ys$ ==> $\exists\, as\ b\ bs\ c\ cs.\ b \neq c \wedge xs = as\ @\ b\ \#\ bs \wedge ys = as\ @\ c\ \#\ cs$
$\langle proof \rangle$

lemma *parallel-append*:
$a \parallel b \Longrightarrow a\ @\ c \parallel b\ @\ d$
⟨*proof*⟩

lemma *parallel-appendI*:
$[\![\ xs \parallel ys;\ x = xs\ @\ xs'\ ;\ y = ys\ @\ ys'\ ]\!] \Longrightarrow x \parallel y$
⟨*proof*⟩

lemma *parallel-commute*: $(a \parallel b) = (b \parallel a)$
⟨*proof*⟩

39.4 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?*]: $xs = zs\ @\ ys ==> xs >>= ys$
⟨*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*: $xs >>= ys \longleftrightarrow rev\ ys \le rev\ xs$
⟨*proof*⟩

lemma *distinct-postfix*:
assumes *distinct xs*
and $xs >>= ys$
shows *distinct ys*
⟨*proof*⟩

lemma *postfix-map*:
assumes $xs >>= ys$
shows $map\ f\ xs >>= map\ f\ ys$
⟨*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*: $x \parallel y \Longrightarrow \neg\ x \le y$
⟨*proof*⟩

lemma *parallelD2*: $x \parallel y \Longrightarrow \neg\ y \le x$
⟨*proof*⟩

lemma *parallel-Nil1* [*simp*]: $\neg\ x \parallel []$
⟨*proof*⟩

lemma *parallel-Nil2* [*simp*]: $\neg\ [] \parallel x$
⟨*proof*⟩

lemma *Cons-parallelI1*:
$a \ne b \Longrightarrow a\ \#\ as \parallel b\ \#\ bs$ ⟨*proof*⟩

lemma *Cons-parallelI2*:
$[\![\ a = b;\ as \parallel bs\ ]\!] \Longrightarrow a\ \#\ as \parallel b\ \#\ bs$
⟨*proof*⟩

lemma *not-equal-is-parallel*:
assumes *neq*: $xs \ne ys$
and *len*: $length\ xs = length\ ys$
shows $xs \parallel ys$

⟨*proof*⟩

39.5 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

40 List-lexord: Lexicographic order on lists

theory *List-lexord*
imports *Main*
begin

instance *list* :: (*ord*) *ord*
list-le-def: $(xs::('a::ord)\ list) \leq ys \equiv (xs < ys \vee xs = ys)$
list-less-def: $(xs::('a::ord)\ list) < ys \equiv (xs,\ ys) \in lexord\ \{(u,v).\ u < v\}$ ⟨*proof*⟩

lemmas *list-ord-defs* [*code func del*] = *list-less-def list-le-def*

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

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

instance *list* :: (*linorder*) *distrib-lattice*
$inf \equiv min$
$sup \equiv max$
⟨*proof*⟩

lemmas [*code func del*] = *inf-list-def sup-list-def*

lemma *not-less-Nil* [*simp*]: $\neg\ (x < [])$
⟨*proof*⟩

lemma *Nil-less-Cons* [*simp*]: $[] < a \ \# \ x$
⟨*proof*⟩

lemma *Cons-less-Cons* [*simp*]: $a \ \# \ x < b \ \# \ y \longleftrightarrow a < b \lor a = b \land x < y$
⟨*proof*⟩

lemma *le-Nil* [*simp*]: $x \leq [] \longleftrightarrow x = []$
⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
⟨*proof*⟩

lemma *Cons-le-Cons* [*simp*]: $a \ \# \ x \leq b \ \# \ y \longleftrightarrow a < b \lor a = b \land x \leq y$
⟨*proof*⟩

lemma *less-code* [*code func*]:
$xs < ([]::'a::\{eq,\ order\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{eq,\ order\}) \ \# \ xs \longleftrightarrow True$
$(x::'a::\{eq,\ order\}) \ \# \ xs < y \ \# \ ys \longleftrightarrow x < y \lor x = y \land xs < ys$
⟨*proof*⟩

lemma *less-eq-code* [*code func*]:
$x \ \# \ xs \leq ([]::'a::\{eq,\ order\}\ list) \longleftrightarrow False$
$[] \leq (xs::'a::\{eq,\ order\}\ list) \longleftrightarrow True$
$(x::'a::\{eq,\ order\}) \ \# \ xs \leq y \ \# \ ys \longleftrightarrow x < y \lor x = y \land xs \leq ys$
⟨*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.