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*: $is\text{-}gcd\ m\ a\ b \Longrightarrow is\text{-}gcd\ n\ a\ b \Longrightarrow m = n$
by (*simp add*: *is-gcd-def*) (*blast intro*: *dvd-anti-sym*)

Connection to divides relation

lemma *is-gcd-dvd*: $is\text{-}gcd\ m\ a\ b \Longrightarrow k\ dvd\ a \Longrightarrow k\ dvd\ b \Longrightarrow k\ dvd\ m$
by (*auto simp add*: *is-gcd-def*)

Commutativity

lemma *is-gcd-commute*: $is\text{-}gcd\ k\ m\ n = is\text{-}gcd\ k\ n\ m$
by (*auto simp add*: *is-gcd-def*)

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 $P\ m\ n$
apply (*rule gcd.induct* [*of split P* (*m*, *n*), *unfolded Product-Type.split*])
apply (*case-tac* $n = 0$)
apply *simp-all*
using *assms* **apply** *simp-all*
done

lemma *gcd-0* [*simp*]: $gcd\ (m,\ 0) = m$
by *simp*

lemma *gcd-0-left* [*simp*]: $gcd\ (0,\ m) = m$

by *simp*

lemma *gcd-non-0*: $n > 0 \Longrightarrow gcd\ (m,\ n) = gcd\ (n,\ m\ mod\ n)$
by *simp*

lemma *gcd-1* [*simp*]: $gcd\ (m,\ Suc\ 0) = 1$
by *simp*

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*]: $gcd\ (m,\ n)\ dvd\ n$
apply (*induct m n rule*: *gcd-induct*)
apply (*simp-all add*: *gcd-non-0*)
apply (*blast dest*: *dvd-mod-imp-dvd*)
done

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

lemma *gcd-greatest*: $k\ dvd\ m \Longrightarrow k\ dvd\ n \Longrightarrow k\ dvd\ gcd\ (m,\ n)$
by (*induct m n rule*: *gcd-induct*) (*simp-all add*: *gcd-non-0 dvd-mod*)

Function gcd yields the Greatest Common Divisor.

lemma *is-gcd*: *is-gcd* $(gcd\ (m,\ n))\ m\ n$
by (*simp add*: *is-gcd-def gcd-greatest*)

1.3 Derived laws for GCD

lemma *gcd-greatest-iff* [*iff*]: $k\ dvd\ gcd\ (m,\ n) \longleftrightarrow k\ dvd\ m \wedge k\ dvd\ n$
by (*blast intro*!: *gcd-greatest intro*: *dvd-trans*)

lemma *gcd-zero*: $gcd\ (m,\ n) = 0 \longleftrightarrow m = 0 \wedge n = 0$
by (*simp only*: *dvd-0-left-iff* [*symmetric*] *gcd-greatest-iff*)

lemma *gcd-commute*: $gcd\ (m,\ n) = gcd\ (n,\ m)$
apply (*rule is-gcd-unique*)
apply (*rule is-gcd*)
apply (*subst is-gcd-commute*)
apply (*simp add*: *is-gcd*)
done

lemma *gcd-assoc*: $gcd\ (gcd\ (k,\ m),\ n) = gcd\ (k,\ gcd\ (m,\ n))$
apply (*rule is-gcd-unique*)
apply (*rule is-gcd*)
apply (*simp add*: *is-gcd-def*)
apply (*blast intro*: *dvd-trans*)

```
  done

lemma gcd-1-left [simp]: gcd (Suc 0, m) = 1
  by (simp add: gcd-commute)
```

Multiplication laws

```
lemma gcd-mult-distrib2: k * gcd (m, n) = gcd (k * m, k * n)
    — [3, page 27]
  apply (induct m n rule: gcd-induct)
   apply simp
  apply (case-tac k = 0)
   apply (simp-all add: mod-geq gcd-non-0 mod-mult-distrib2)
  done

lemma gcd-mult [simp]: gcd (k, k * n) = k
  apply (rule gcd-mult-distrib2 [of k 1 n, simplified, symmetric])
  done

lemma gcd-self [simp]: gcd (k, k) = k
  apply (rule gcd-mult [of k 1, simplified])
  done

lemma relprime-dvd-mult: gcd (k, n) = 1 ==> k dvd m * n ==> k dvd m
  apply (insert gcd-mult-distrib2 [of m k n])
  apply simp
  apply (erule-tac t = m in ssubst)
  apply simp
  done

lemma relprime-dvd-mult-iff: gcd (k, n) = 1 ==> (k dvd m * n) = (k dvd m)
  apply (blast intro: relprime-dvd-mult dvd-trans)
  done

lemma gcd-mult-cancel: gcd (k, n) = 1 ==> gcd (k * m, n) = gcd (m, n)
  apply (rule dvd-anti-sym)
   apply (rule gcd-greatest)
    apply (rule-tac n = k in relprime-dvd-mult)
     apply (simp add: gcd-assoc)
     apply (simp add: gcd-commute)
    apply (simp-all add: mult-commute)
  apply (blast intro: dvd-trans)
  done
```

Addition laws

```
lemma gcd-add1 [simp]: gcd (m + n, n) = gcd (m, n)
  apply (case-tac n = 0)
   apply (simp-all add: gcd-non-0)
  done
```

```
lemma gcd-add2 [simp]: gcd (m, m + n) = gcd (m, n)
proof -
  have gcd (m, m + n) = gcd (m + n, m) by (rule gcd-commute)
  also have ... = gcd (n + m, m) by (simp add: add-commute)
  also have ... = gcd (n, m) by simp
  also have  ... = gcd (m, n) by (rule gcd-commute)
  finally show ?thesis .
qed

lemma gcd-add2' [simp]: gcd (m, n + m) = gcd (m, n)
  apply (subst add-commute)
  apply (rule gcd-add2)
  done

lemma gcd-add-mult: gcd (m, k * m + n) = gcd (m, n)
  by (induct k) (simp-all add: add-assoc)

lemma gcd-dvd-prod: gcd (m, n) dvd m * n
  using mult-dvd-mono [of 1] by auto
```

Division by gcd yields rrelatively primes.

```
lemma div-gcd-relprime:
  assumes nz: a ≠ 0 ∨ b ≠ 0
  shows gcd (a div gcd(a,b), b div gcd(a,b)) = 1
proof -
  let ?g = gcd (a, b)
  let ?a' = a div ?g
  let ?b' = b div ?g
  let ?g' = gcd (?a', ?b')
  have dvdg: ?g dvd a ?g dvd b by simp-all
  have dvdg': ?g' dvd ?a' ?g' dvd ?b' by simp-all
  from dvdg dvdg' obtain ka kb ka' kb' where
      kab: a = ?g * ka b = ?g * kb ?a' = ?g' * ka' ?b' = ?g' * kb'
    unfolding dvd-def by blast
  then have ?g * ?a' = (?g * ?g') * ka' ?g * ?b' = (?g * ?g') * kb' by simp-all
  then have dvdgg':?g * ?g' dvd a ?g* ?g' dvd b
    by (auto simp add: dvd-mult-div-cancel [OF dvdg(1)]
      dvd-mult-div-cancel [OF dvdg(2)] dvd-def)
  have ?g ≠ 0 using nz by (simp add: gcd-zero)
  then have gp: ?g > 0 by simp
  from gcd-greatest [OF dvdgg'] have ?g * ?g' dvd ?g .
  with dvd-mult-cancel1 [OF gp] show ?g' = 1 by simp
qed
```

1.4 LCM defined by GCD

```
definition
  lcm :: nat × nat ⇒ nat
```

```
where
  lcm = (λ(m, n). m * n div gcd (m, n))

lemma lcm-def:
  lcm (m, n) = m * n div gcd (m, n)
  unfolding lcm-def by simp

lemma prod-gcd-lcm:
  m * n = gcd (m, n) * lcm (m, n)
  unfolding lcm-def by (simp add: dvd-mult-div-cancel [OF gcd-dvd-prod])

lemma lcm-0 [simp]: lcm (m, 0) = 0
  unfolding lcm-def by simp

lemma lcm-1 [simp]: lcm (m, 1) = m
  unfolding lcm-def by simp

lemma lcm-0-left [simp]: lcm (0, n) = 0
  unfolding lcm-def by simp

lemma lcm-1-left [simp]: lcm (1, m) = m
  unfolding lcm-def by simp

lemma dvd-pos:
  fixes n m :: nat
  assumes n > 0 and m dvd n
  shows m > 0
using assms by (cases m) auto

lemma lcm-least:
  assumes m dvd k and n dvd k
  shows lcm (m, n) dvd k
proof (cases k)
  case 0 then show ?thesis by auto
next
  case (Suc -) then have pos-k: k > 0 by auto
  from assms dvd-pos [OF this] have pos-mn: m > 0 n > 0 by auto
  with gcd-zero [of m n] have pos-gcd: gcd (m, n) > 0 by simp
  from assms obtain p where k-m: k = m * p using dvd-def by blast
  from assms obtain q where k-n: k = n * q using dvd-def by blast
  from pos-k k-m have pos-p: p > 0 by auto
  from pos-k k-n have pos-q: q > 0 by auto
  have k * k * gcd (q, p) = k * gcd (k * q, k * p)
    by (simp add: mult-ac gcd-mult-distrib2)
  also have ... = k * gcd (m * p * q, n * q * p)
    by (simp add: k-m [symmetric] k-n [symmetric])
  also have ... = k * p * q * gcd (m, n)
    by (simp add: mult-ac gcd-mult-distrib2)
  finally have (m * p) * (n * q) * gcd (q, p) = k * p * q * gcd (m, n)
```

```
    by (simp only: k-m [symmetric] k-n [symmetric])
  then have p * q * m * n * gcd (q, p) = p * q * k * gcd (m, n)
    by (simp add: mult-ac)
  with pos-p pos-q have m * n * gcd (q, p) = k * gcd (m, n)
    by simp
  with prod-gcd-lcm [of m n]
  have lcm (m, n) * gcd (q, p) * gcd (m, n) = k * gcd (m, n)
    by (simp add: mult-ac)
  with pos-gcd have lcm (m, n) * gcd (q, p) = k by simp
  then show ?thesis using dvd-def by auto
qed

lemma lcm-dvd1 [iff]:
  m dvd lcm (m, n)
proof (cases m)
  case 0 then show ?thesis by simp
next
  case (Suc -)
  then have mpos: m > 0 by simp
  show ?thesis
  proof (cases n)
    case 0 then show ?thesis by simp
  next
    case (Suc -)
    then have npos: n > 0 by simp
    have gcd (m, n) dvd n by simp
    then obtain k where n = gcd (m, n) * k using dvd-def by auto
    then have m * n div gcd (m, n) = m * (gcd (m, n) * k) div gcd (m, n) by
(simp add: mult-ac)
    also have ... = m * k using mpos npos gcd-zero by simp
    finally show ?thesis by (simp add: lcm-def)
  qed
qed

lemma lcm-dvd2 [iff]:
  n dvd lcm (m, n)
proof (cases n)
  case 0 then show ?thesis by simp
next
  case (Suc -)
  then have npos: n > 0 by simp
  show ?thesis
  proof (cases m)
    case 0 then show ?thesis by simp
  next
    case (Suc -)
    then have mpos: m > 0 by simp
    have gcd (m, n) dvd m by simp
    then obtain k where m = gcd (m, n) * k using dvd-def by auto
```

then have $m * n$ *div gcd* $(m, n) = (gcd\ (m, n) * k) * n$ *div gcd* (m, n) **by** (*simp add*: *mult-ac*)
also have ... $= n * k$ **using** *mpos npos gcd-zero* **by** *simp*
finally show *?thesis* **by** (*simp add*: *lcm-def*)
qed
qed

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*
by (*simp add*: *igcd-def int-dvd-iff*)

lemma *igcd-dvd2* [*simp*]: *igcd i j dvd j*
by (*simp add*: *igcd-def int-dvd-iff*)

lemma *igcd-pos*: *igcd i j* ≥ *0*
by (*simp add*: *igcd-def*)

lemma *igcd0* [*simp*]: (*igcd i j* = *0*) = (*i* = *0* ∧ *j* = *0*)
by (*simp add*: *igcd-def gcd-zero*) *arith*

lemma *igcd-commute*: *igcd i j* = *igcd j i*
unfolding *igcd-def* **by** (*simp add*: *gcd-commute*)

lemma *igcd-neg1* [*simp*]: *igcd* (− *i*) *j* = *igcd i j*
unfolding *igcd-def* **by** *simp*

lemma *igcd-neg2* [*simp*]: *igcd i* (− *j*) = *igcd i j*
unfolding *igcd-def* **by** *simp*

lemma *zrelprime-dvd-mult*: *igcd i j* = *1* ⟹ *i dvd k* ∗ *j* ⟹ *i dvd k*
unfolding *igcd-def*
proof −
assume *int* (*gcd* (*nat* |*i*|, *nat* |*j*|)) = *1* *i dvd k* ∗ *j*
then have *g*: *gcd* (*nat* |*i*|, *nat* |*j*|) = *1* **by** *simp*
from ‹*i dvd k* ∗ *j*› **obtain** *h* **where** *h*: *k*∗*j* = *i*∗*h* **unfolding** *dvd-def* **by** *blast*
have *th*: *nat* |*i*| *dvd nat* |*k*| ∗ *nat* |*j*|
unfolding *dvd-def*
by (*rule-tac x*= *nat* |*h*| **in** *exI*, *simp add*: *h nat-abs-mult-distrib* [*symmetric*])
from *relprime-dvd-mult* [*OF g th*] **obtain** *h′* **where** *h′*: *nat* |*k*| = *nat* |*i*| ∗ *h′*
unfolding *dvd-def* **by** *blast*
from *h′* **have** *int* (*nat* |*k*|) = *int* (*nat* |*i*| ∗ *h′*) **by** *simp*
then have |*k*| = |*i*| ∗ *int h′* **by** (*simp add*: *int-mult*)
then show *?thesis*
apply (*subst zdvd-abs1* [*symmetric*])

```
    apply (subst zdvd-abs2 [symmetric])
    apply (unfold dvd-def)
    apply (rule-tac x = int h' in exI, simp)
    done
qed

lemma int-nat-abs: int (nat (abs x)) = abs x  by arith

lemma igcd-greatest:
  assumes k dvd m and k dvd n
  shows k dvd igcd m n
proof −
  let ?k' = nat |k|
  let ?m' = nat |m|
  let ?n' = nat |n|
  from ‹k dvd m› and ‹k dvd n› have dvd': ?k' dvd ?m' ?k' dvd ?n'
    unfolding zdvd-int by (simp-all only: int-nat-abs zdvd-abs1 zdvd-abs2)
  from gcd-greatest [OF dvd'] have int (nat |k|) dvd igcd m n
    unfolding igcd-def by (simp only: zdvd-int)
  then have |k| dvd igcd m n by (simp only: int-nat-abs)
  then show k dvd igcd m n by (simp add: zdvd-abs1)
qed

lemma div-igcd-relprime:
  assumes nz: a ≠ 0 ∨ b ≠ 0
  shows igcd (a div (igcd a b)) (b div (igcd a b)) = 1
proof −
  from nz have nz': nat |a| ≠ 0 ∨ nat |b| ≠ 0 by arith
  let ?g = igcd a b
  let ?a' = a div ?g
  let ?b' = b div ?g
  let ?g' = igcd ?a' ?b'
  have dvdg: ?g dvd a ?g dvd b by (simp-all add: igcd-dvd1 igcd-dvd2)
  have dvdg': ?g' dvd ?a' ?g' dvd ?b' by (simp-all add: igcd-dvd1 igcd-dvd2)
  from dvdg dvdg' obtain ka kb ka' kb' where
   kab: a = ?g*ka b = ?g*kb ?a' = ?g'*ka' ?b' = ?g' * kb'
    unfolding dvd-def by blast
  then have ?g* ?a' = (?g * ?g') * ka' ?g* ?b' = (?g * ?g') * kb' by simp-all
  then have dvdgg':?g * ?g' dvd a ?g* ?g' dvd b
    by (auto simp add: zdvd-mult-div-cancel [OF dvdg(1)]
      zdvd-mult-div-cancel [OF dvdg(2)] dvd-def)
  have ?g ≠ 0 using nz by simp
  then have gp: ?g ≠ 0 using igcd-pos[where i=a and j=b] by arith
  from igcd-greatest [OF dvdgg'] have ?g * ?g' dvd ?g .
  with zdvd-mult-cancel1 [OF gp] have |?g'| = 1 by simp
  with igcd-pos show ?g' = 1 by simp
qed

definition ilcm = (λi j. int (lcm(nat(abs i),nat(abs j))))
```

```
lemma dvd-ilcm-self1[simp]: i dvd ilcm i j
by(simp add:ilcm-def dvd-int-iff)

lemma dvd-ilcm-self2[simp]: j dvd ilcm i j
by(simp add:ilcm-def dvd-int-iff)


lemma dvd-imp-dvd-ilcm1:
  assumes k dvd i shows k dvd (ilcm i j)
proof -
  have nat(abs k) dvd nat(abs i) using ‹k dvd i›
    by(simp add:int-dvd-iff[symmetric] dvd-int-iff[symmetric] zdvd-abs1)
  thus ?thesis by(simp add:ilcm-def dvd-int-iff)(blast intro: dvd-trans)
qed

lemma dvd-imp-dvd-ilcm2:
  assumes k dvd j shows k dvd (ilcm i j)
proof -
  have nat(abs k) dvd nat(abs j) using ‹k dvd j›
    by(simp add:int-dvd-iff[symmetric] dvd-int-iff[symmetric] zdvd-abs1)
  thus ?thesis by(simp add:ilcm-def dvd-int-iff)(blast intro: dvd-trans)
qed


lemma zdvd-self-abs1: (d::int) dvd (abs d)
by (case-tac d <0, simp-all)

lemma zdvd-self-abs2: (abs (d::int)) dvd d
by (case-tac d<0, simp-all)



lemma lcm-pos:
  assumes mpos: m > 0
  and npos: n>0
  shows lcm (m,n) > 0
proof(rule ccontr, simp add: lcm-def gcd-zero)
assume h:m*n div gcd(m,n) = 0
from mpos npos have gcd (m,n) ≠ 0 using gcd-zero by simp
hence gcdp: gcd(m,n) > 0 by simp
with h
have m*n < gcd(m,n)
 by (cases m * n < gcd (m, n)) (auto simp add: div-if[OF gcdp, where m=m*n])
moreover
have gcd(m,n) dvd m by simp
 with mpos dvd-imp-le have t1:gcd(m,n) ≤ m by simp
 with npos have t1:gcd(m,n)*n ≤ m*n by simp
 have gcd(m,n) ≤ gcd(m,n)*n using npos by simp
```

```
 with t1 have gcd(m,n) ≤ m∗n by arith
ultimately show False by simp
qed

lemma ilcm-pos:
  assumes anz: a ≠ 0
  and bnz: b ≠ 0
  shows 0 < ilcm a b
proof−
  let ?na = nat (abs a)
  let ?nb = nat (abs b)
  have nap: ?na >0 using anz by simp
  have nbp: ?nb >0 using bnz by simp
  have 0 < lcm (?na,?nb) by (rule lcm-pos[OF nap nbp])
  thus ?thesis by (simp add: ilcm-def)
qed

end
```

2 Abstract-Rat: Abstract rational numbers

```
theory Abstract-Rat
imports GCD
begin

types Num = int × int

abbreviation
  Num0-syn :: Num (0_N)
where 0_N ≡ (0, 0)

abbreviation
  Numi-syn :: int ⇒ Num (-_N)
where i_N ≡ (i, 1)

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

definition
  normNum :: Num ⇒ Num
where
  normNum = (λ(a,b). (if a=0 ∨ 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 −
 have ∃ *a b*. *x* = (*a*,*b*) **by** *auto*
 then obtain *a b* **where** *x*[*simp*]: *x* = (*a*,*b*) **by** *blast*
 {**assume** *a*=*0* ∨ *b* = *0* **hence** *?thesis* **by** (*simp add*: *normNum-def isnormNum-def*)}

 moreover
 {**assume** *anz*: *a* ≠ *0* **and** *bnz*: *b* ≠ *0*
  let *?g* = *igcd a b*
  let *?a′* = *a div ?g*
  let *?b′* = *b div ?g*
  let *?g′* = *igcd ?a′ ?b′*
  from *anz bnz* **have** *?g* ≠ *0* **by** *simp* **with** *igcd-pos*[*of a b*]
  have *gpos*: *?g* > *0* **by** *arith*
  have *gdvd*: *?g dvd a ?g dvd b* **by** (*simp-all add*: *igcd-dvd1 igcd-dvd2*)
  from *zdvd-mult-div-cancel*[*OF gdvd*(*1*)] *zdvd-mult-div-cancel*[*OF gdvd*(*2*)]
  anz bnz
  have *nz′*:*?a′* ≠ *0 ?b′* ≠ *0*
   by − (*rule notI*,*simp add*:*igcd-def*)+
  from *anz bnz* **have** *stupid*: *a* ≠ *0* ∨ *b* ≠ *0* **by** *blast*
  from *div-igcd-relprime*[*OF stupid*] **have** *gp1*: *?g′* = *1* **.**
  from *bnz* **have** *b* < *0* ∨ *b* > *0* **by** *arith*
  moreover
  {**assume** *b*: *b* > *0*
   from *pos-imp-zdiv-nonneg-iff*[*OF gpos*] *b*
   have *?b′* ≥ *0* **by** *simp*
   with *nz′* **have** *b′*: *?b′* > *0* **by** *simp*
   from *b b′ anz bnz nz′ gp1* **have** *?thesis*
     by (*simp add*: *isnormNum-def normNum-def Let-def split-def fst-conv snd-conv*)}
  moreover {**assume** *b*: *b* < *0*
   {**assume** *b′*: *?b′* ≥ *0*
    from *gpos* **have** *th*: *?g* ≥ *0* **by** *arith*
    from *mult-nonneg-nonneg*[*OF th b′*] *zdvd-mult-div-cancel*[*OF gdvd*(*2*)]
    have *False* **using** *b* **by** *simp* }
   hence *b′*: *?b′* < *0* **by** (*presburger add*: *linorder-not-le*[*symmetric*])
   from *anz bnz nz′ b b′ gp1* **have** *?thesis*
     by (*simp add*: *isnormNum-def normNum-def Let-def split-def fst-conv snd-conv*)}
  ultimately have *?thesis* **by** *blast*
 }
 ultimately show *?thesis* **by** *blast*
qed

Arithmetic over Num

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

else normNum$(a{*}b' + b{*}a',\ b{*}b')$)

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

definition
Nneg :: *Num* ⇒ *Num* ($\sim_N$)
where
Nneg ≡ $(\lambda(a,b).\ (-a,b))$

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

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

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

lemma *Nneg-normN*[*simp*]: *isnormNum* x ⟹ *isnormNum* ($\sim_N x$)
by(*simp add*: *isnormNum-def Nneg-def split-def*)
lemma *Nadd-normN*[*simp*]: *isnormNum* ($x +_N y$)
by (*simp add*: *Nadd-def split-def*)
lemma *Nsub-normN*[*simp*]: ⟦ *isnormNum* y⟧ ⟹ *isnormNum* ($x -_N y$)
by (*simp add*: *Nsub-def split-def*)
lemma *Nmul-normN*[*simp*]: **assumes** *xn*:*isnormNum* x **and** *yn*: *isnormNum* y
shows *isnormNum* ($x *_N y$)
proof−
have $\exists\ a\ b.\ x = (a,b)$ **and** $\exists\ a'\ b'.\ y = (a',b')$ **by** *auto*
then obtain $a\ b\ a'\ b'$ **where** *ab*: $x = (a,b)$ **and** *ab'*: $y = (a',b')$ **by** *blast*
{**assume** $a = 0$
hence *?thesis* **using** *xn ab ab'*
by (*simp add*: *igcd-def isnormNum-def Let-def Nmul-def split-def*)}
moreover
{**assume** $a' = 0$
hence *?thesis* **using** *yn ab ab'*
by (*simp add*: *igcd-def isnormNum-def Let-def Nmul-def split-def*)}
moreover
{**assume** *a*: $a \neq 0$ **and** *a'*: $a' \neq 0$
hence *bp*: $b > 0\ b' > 0$ **using** *xn yn ab ab'* **by** (*simp-all add*: *isnormNum-def*)

from *mult-pos-pos*[*OF bp*] **have** $x *_N y = normNum\ (a{*}a',\ b{*}b')$
using *ab ab′ a a′ bp* **by** (*simp add*: *Nmul-def Let-def split-def normNum-def*)
hence *?thesis* **by** *simp*}
ultimately show *?thesis* **by** *blast*
qed

lemma *Ninv-normN*[*simp*]: $isnormNum\ x \Longrightarrow isnormNum\ (Ninv\ x)$
by (*simp add*: *Ninv-def isnormNum-def split-def*)
(*cases fst x = 0*, *auto simp add*: *igcd-commute*)

lemma *isnormNum-int*[*simp*]:
$isnormNum\ 0_N\ \ isnormNum\ (1{::}int)_N\ \ i \neq 0 \Longrightarrow isnormNum\ i_N$
by (*simp-all add*: *isnormNum-def igcd-def*)

Relations over Num

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

definition
$Nle0{::}\ Num \Rightarrow bool\ (0{\geq}_N)$
where
$Nle0 = (\lambda(a,b).\ a \leq 0)$

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

definition
$Nge0{::}\ Num \Rightarrow bool\ (0{\leq}_N)$
where
$Nge0 = (\lambda(a,b).\ a \geq 0)$

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

definition
$Nle :: Num \Rightarrow Num \Rightarrow bool$ (**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*]: $INum\ i_N = ((of\text{-}int\ i) :: 'a{::}field)\ \ INum\ 0_N = (0{::}'a{::}field)$

```
  by (simp-all add: INum-def)

lemma isnormNum-unique[simp]:
  assumes na: isnormNum x and nb: isnormNum y
  shows ((INum x ::'a::{ring-char-0,field, division-by-zero}) = INum y) = (x =
y) (is ?lhs = ?rhs)
proof
  have ∃ a b a' b'. x = (a,b) ∧ y = (a',b') by auto
  then obtain a b a' b' where xy[simp]: x = (a,b) y=(a',b') by blast
  assume H: ?lhs
  {assume a = 0 ∨ b = 0 ∨ a' = 0 ∨ b' = 0 hence ?rhs
    using na nb H
    apply (simp add: INum-def split-def isnormNum-def)
    apply (cases a = 0, simp-all)
    apply (cases b = 0, simp-all)
    apply (cases a' = 0, simp-all)
    apply (cases a' = 0, simp-all add: of-int-eq-0-iff)
    done}
  moreover
  { assume az: a ≠ 0 and bz: b ≠ 0 and a'z: a'≠0 and b'z: b'≠0
   from az bz a'z b'z na nb have pos: b > 0 b' > 0 by (simp-all add: isnormNum-def)
   from prems have eq:a * b' = a'*b
     by (simp add: INum-def  eq-divide-eq divide-eq-eq of-int-mult[symmetric] del:
of-int-mult)
   from prems have gcd1: igcd a b = 1 igcd b a = 1 igcd a' b' = 1 igcd b' a' =
1
     by (simp-all add: isnormNum-def add: igcd-commute)
   from eq have raw-dvd: a dvd a'*b b dvd b'*a a' dvd a*b' b' dvd b*a'
     apply(unfold dvd-def)
     apply (rule-tac x=b' in exI, simp add: mult-ac)
     apply (rule-tac x=a' in exI, simp add: mult-ac)
     apply (rule-tac x=b in exI, simp add: mult-ac)
     apply (rule-tac x=a in exI, simp add: mult-ac)
     done
   from zdvd-dvd-eq[OF bz zrelprime-dvd-mult[OF gcd1(2) raw-dvd(2)]
     zrelprime-dvd-mult[OF gcd1(4) raw-dvd(4)]]
     have eq1: b = b' using pos by simp-all
     with eq have a = a' using pos by simp
     with eq1 have ?rhs by simp}
  ultimately show ?rhs by blast
next
  assume ?rhs thus ?lhs by simp
qed


lemma isnormNum0[simp]: isnormNum x ⟹ (INum x = (0::'a::{ring-char-0,
field,division-by-zero})) = (x = 0_N)
  unfolding INum-int(2)[symmetric]
  by (rule isnormNum-unique, simp-all)
```

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 −
 assume *d* ~= *0*
 hence *dz*: *of-int d* ≠ (*0*::′*a*) **by** (*simp add*: *of-int-eq-0-iff*)
 let *?t* = *of-int* (*x div d*) ∗ ((*of-int d*)::′*a*) + *of-int*(*x mod d*)
 let *?f* = λ*x*. *x* / *of-int d*
 have *x* = (*x div d*) ∗ *d* + *x mod d*
  by *auto*
 then have *eq*: *of-int x* = *?t*
  by (*simp only*: *of-int-mult*[*symmetric*] *of-int-add* [*symmetric*])
 then have *of-int x* / *of-int d* = *?t* / *of-int d*
  using *cong*[*OF refl*[*of ?f*] *eq*] **by** *simp*
 then show *?thesis* **by** (*simp add*: *add-divide-distrib ring-simps prems*)
qed

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*
 apply (*frule of-int-div-aux* [*of d n*, **where** *?′a* = ′*a*])
 apply *simp*
 apply (*simp add*: *zdvd-iff-zmod-eq-0*)
done

lemma *normNum*[*simp*]: *INum* (*normNum x*) = (*INum x* :: ′*a*::{*ring-char-0*,*field*, *division-by-zero*})
proof−
 have ∃ *a b*. *x* = (*a*,*b*) **by** *auto*
 then obtain *a b* **where** *x*[*simp*]: *x* = (*a*,*b*) **by** *blast*
 {assume *a*=*0* ∨ *b* = *0* **hence** *?thesis*
   by (*simp add*: *INum-def normNum-def split-def Let-def*)**}**
 moreover
 {assume *a*: *a*≠*0* **and** *b*: *b*≠*0*
  let *?g* = *igcd a b*
  from *a b* **have** *g*: *?g* ≠ *0***by** *simp*
  from *of-int-div*[*OF g*, **where** *?′a* = ′*a*]
  have *?thesis* **by** (*auto simp add*: *INum-def normNum-def split-def Let-def*)**}**
 ultimately show *?thesis* **by** *blast*
qed

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 −
 have *normNum x* = *normNum y* ⟷ (*INum* (*normNum x*) :: ′*a*) = *INum* (*normNum y*)
  by (*simp del*: *normNum*)
 also have ... = *?lhs* **by** *simp*

  finally show *?thesis* **by** *simp*
qed

lemma *Nadd*[*simp*]: *INum* $(x +_N y)$ = *INum* x + (*INum* y :: $'a$ :: {*ring-char-0*,*division-by-zero*,*field*})
proof−
let $?z = 0$:: $'a$
  have $\exists\ a\ b.\ x = (a,b)\ \ \exists\ a'\ b'.\ y = (a',b')$ **by** *auto*
  then obtain $a\ b\ a'\ b'$ **where** x[*simp*]: $x = (a,b)$
    and y[*simp*]: $y = (a',b')$ **by** *blast*
  {**assume** $a=0 \vee a'= 0 \vee b =0 \vee b' = 0$ **hence** *?thesis*
      apply (*cases* $a=0$,*simp-all add*: *Nadd-def*)
      apply (*cases* $b= 0$,*simp-all add*: *INum-def*)
       apply (*cases* $a'= 0$,*simp-all*)
       apply (*cases* $b'= 0$,*simp-all*)
       done }
  moreover
  {**assume** aa':$a \neq 0\ a'\neq 0$ **and** bb': $b \neq 0\ b' \neq 0$
    {**assume** z: $a * b' + b * a' = 0$
      hence *of-int* $(a*b' + b*a')$ / (*of-int* $b*$ *of-int* b') = $?z$ **by** *simp*
     hence *of-int* b' $*$ *of-int* a / (*of-int* b $*$ *of-int* b') + *of-int* b $*$ *of-int* a' / (*of-int* b $*$ *of-int* b') = $?z$ **by** (*simp add*:*add-divide-distrib*)
      hence th: *of-int* a / *of-int* b + *of-int* a' / *of-int* b' = $?z$ **using** bb' aa' **by** *simp*
      from z aa' bb' **have** *?thesis*
        by (*simp add*: *th Nadd-def normNum-def INum-def split-def*)}
    moreover {**assume** z: $a * b' + b * a' \neq 0$
      let $?g$ = *igcd* $(a * b' + b * a')$ $(b*b')$
      have gz: $?g \neq 0$ **using** z **by** *simp*
      have *?thesis* **using** aa' bb' z gz
        of-int-div[**where** $?'a = 'a$,
        OF gz igcd-dvd1[**where** $i=a * b' + b * a'$ **and** $j=b*b'$]]
        of-int-div[**where** $?'a = 'a$,
        OF gz igcd-dvd2[**where** $i=a * b' + b * a'$ **and** $j=b*b'$]]
      by (*simp add*: *x y Nadd-def INum-def normNum-def Let-def add-divide-distrib*)}
    ultimately have *?thesis* **using** aa' bb'
      by (*simp add*: *Nadd-def INum-def normNum-def x y Let-def*) }
  ultimately show *?thesis* **by** *blast*
qed

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

proof−
  let $?z = 0$::$'a$
  have $\exists\ a\ b.\ x = (a,b)\ \ \exists\ a'\ b'.\ y = (a',b')$ **by** *auto*
  then obtain $a\ b\ a'\ b'$ **where** x: $x = (a,b)$ **and** y: $y = (a',b')$ **by** *blast*
  {**assume** $a=0 \vee a'= 0 \vee b = 0 \vee b' = 0$ **hence** *?thesis*
      apply (*cases* $a=0$,*simp-all add*: *x y Nmul-def INum-def Let-def*)
      apply (*cases* $b=0$,*simp-all*)
      apply (*cases* $a'=0$,*simp-all*)

done }
moreover
{**assume** *z*: $a \neq 0$ $a' \neq 0$ $b \neq 0$ $b' \neq 0$
let *?g=igcd* $(a*a')$ $(b*b')$
have *gz*: *?g* $\neq 0$ **using** *z* **by** *simp*
from *z of-int-div*[**where** *?'a* = *'a*, *OF gz igcd-dvd1*[**where** *i*=$a*a'$ **and** *j*=$b*b'$]]

of-int-div[**where** *?'a* = *'a* , *OF gz igcd-dvd2*[**where** *i*=$a*a'$ **and** *j*=$b*b'$]]
have *?thesis* **by** (*simp add*: *Nmul-def x y Let-def INum-def*)}
ultimately show *?thesis* **by** *blast*
qed

lemma *Nneg*[*simp*]: *INum* ($\sim_N$ *x*) = $-$ (*INum x* ::*'a*:: *field*)
by (*simp add*: *Nneg-def split-def INum-def*)

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

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

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

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$-$
have $\exists$ *a b*. *x* = (*a*,*b*) **by** *simp*
then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{**assume** *a* = *0* **hence** *?thesis* **by** (*simp add*: *Nlt0-def INum-def*) }
moreover
{**assume** *a*: *a*$\neq$*0* **hence** *b*: (*of-int b*::*'a*) $>$ *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-divide-less-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::*'a*]
have *?thesis* **by** (*simp add*: *Nlt0-def INum-def*)}
ultimately show *?thesis* **by** *blast*
qed

lemma *Nle0-iff*[*simp*]:**assumes** *nx*: *isnormNum x*
shows ((*INum x* :: *'a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) $\leq$ *0*) = $0 \geq_N x$
proof$-$
have $\exists$ *a b*. *x* = (*a*,*b*) **by** *simp*
then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{**assume** *a* = *0* **hence** *?thesis* **by** (*simp add*: *Nle0-def INum-def*) }
moreover
{**assume** *a*: *a*$\neq$*0* **hence** *b*: (*of-int b* :: *'a*) $>$ *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-divide-le-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::*'a*]

have *?thesis* **by** (*simp add*: *Nle0-def INum-def*)}
ultimately show *?thesis* **by** *blast*
qed

lemma *Ngt0-iff* [*simp*]:**assumes** *nx*: *isnormNum x* **shows** ((*INum x* :: *'a* :: {*ring-char-0*,*division-by-zero*,*ordere*
0) = *0*$<_N$ *x*
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *simp*
then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{**assume** *a* = *0* **hence** *?thesis* **by** (*simp add*: *Ngt0-def INum-def*) }
moreover
{**assume** *a*: *a*≠*0* **hence** *b*: (*of-int b*::*'a*) > *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-less-divide-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::*'a*]
have *?thesis* **by** (*simp add*: *Ngt0-def INum-def*)}
ultimately show *?thesis* **by** *blast*
qed
lemma *Nge0-iff* [*simp*]:**assumes** *nx*: *isnormNum x*
shows ((*INum x* :: *'a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) ≥ *0*) = *0*$\leq_N$
x
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *simp*
then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{**assume** *a* = *0* **hence** *?thesis* **by** (*simp add*: *Nge0-def INum-def*) }
moreover
{**assume** *a*: *a*≠*0* **hence** *b*: (*of-int b*::*'a*) > *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-le-divide-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::*'a*]
have *?thesis* **by** (*simp add*: *Nge0-def INum-def*)}
ultimately show *?thesis* **by** *blast*
qed

lemma *Nlt-iff* [*simp*]: **assumes** *nx*: *isnormNum x* **and** *ny*: *isnormNum y*
shows ((*INum x* :: *'a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) < *INum y*)
= (*x* $<_N$ *y*)
proof−
let *?z* = *0*::*'a*
have ((*INum x* ::*'a*) < *INum y*) = (*INum* (*x* $-_N$ *y*) < *?z*) **using** *nx ny* **by** *simp*
also have ... = (*0*$>_N$ (*x* $-_N$ *y*)) **using** *Nlt0-iff* [*OF Nsub-normN*[*OF ny*]] **by**
simp
finally show *?thesis* **by** (*simp add*: *Nlt-def*)
qed

lemma *Nle-iff* [*simp*]: **assumes** *nx*: *isnormNum x* **and** *ny*: *isnormNum y*
shows ((*INum x* :: *'a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*})≤ *INum y*)
= (*x* $\leq_N$ *y*)
proof−
have ((*INum x* ::*'a*) ≤ *INum y*) = (*INum* (*x* $-_N$ *y*) ≤ (*0*::*'a*)) **using** *nx ny* **by**
simp
also have ... = (*0*$\geq_N$ (*x* $-_N$ *y*)) **using** *Nle0-iff* [*OF Nsub-normN*[*OF ny*]] **by**
simp

finally show *?thesis* **by** (*simp add*: *Nle-def*)
qed

lemma *Nadd-commute*: $x +_N y = y +_N x$
proof −
have *n*: *isnormNum* $(x +_N y)$ *isnormNum* $(y +_N x)$ **by** *simp-all*
have (*INum* $(x +_N y)$::′*a* :: {*ring-char-0*,*division-by-zero*,*field*}) = *INum* $(y +_N x)$ **by** *simp*
with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

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*
apply (*simp add*: *Nadd-def split-def*, *simp add*: *Nadd-def split-def*)
apply (*subst Nadd-commute*,*simp add*: *Nadd-def split-def*)
apply (*subst Nadd-commute*,*simp add*: *Nadd-def split-def*)
done

lemma *normNum-nilpotent-aux*[*simp*]: **assumes** *nx*: *isnormNum x*
shows *normNum x* = *x*
proof −
let *?a* = *normNum x*
have *n*: *isnormNum ?a* **by** *simp*
have *th*:*INum ?a* = (*INum x* ::′*a* :: {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
with *isnormNum-unique*[*OF n nx*]
show *?thesis* **by** *simp*
qed

lemma *normNum-nilpotent*[*simp*]: *normNum* (*normNum x*) = *normNum x*
by *simp*
lemma *normNum0*[*simp*]: *normNum* (*0*,*b*) = 0_N *normNum* (*a*,*0*) = 0_N
by (*simp-all add*: *normNum-def*)
lemma *normNum-Nadd*: *normNum* $(x +_N y) = x +_N y$ **by** *simp*
lemma *Nadd-normNum1*[*simp*]: *normNum* $x +_N y = x +_N y$
proof −
have *n*: *isnormNum* (*normNum* $x +_N y$) *isnormNum* $(x +_N y)$ **by** *simp-all*
have *INum* (*normNum* $x +_N y$) = *INum x* + (*INum y* :: ′*a* :: {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
also have ... = *INum* $(x +_N y)$ **by** *simp*
finally show *?thesis* **using** *isnormNum-unique*[*OF n*] **by** *simp*
qed
lemma *Nadd-normNum2*[*simp*]: $x +_N$ *normNum* $y = x +_N y$
proof −
have *n*: *isnormNum* $(x +_N$ *normNum y*$)$ *isnormNum* $(x +_N y)$ **by** *simp-all*
have *INum* $(x +_N$ *normNum y*$)$ = *INum x* + (*INum y* :: ′*a* :: {*ring-char-0*, *division-by-zero*,*field*}) **by** *simp*
also have ... = *INum* $(x +_N y)$ **by** *simp*
finally show *?thesis* **using** *isnormNum-unique*[*OF n*] **by** *simp*
qed

lemma *Nadd-assoc*: $x +_N y +_N z = x +_N (y +_N z)$
proof−
have *n*: *isnormNum* $(x +_N y +_N z)$ *isnormNum* $(x +_N (y +_N z))$ **by** *simp-all*
have *INum* $(x +_N y +_N z) = (INum\ (x +_N (y +_N z)) :: 'a :: \{ring\text{-}char\text{-}0,$ $division\text{-}by\text{-}zero,field\})$ **by** *simp*
with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

lemma *Nmul-commute*: *isnormNum* $x \Longrightarrow$ *isnormNum* $y \Longrightarrow x *_N y = y *_N x$
by (*simp add*: *Nmul-def split-def Let-def igcd-commute mult-commute*)

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−
from *nx ny nz* **have** *n*: *isnormNum* $(x *_N y *_N z)$ *isnormNum* $(x *_N (y *_N z))$

by *simp-all*
have *INum* $(x +_N y +_N z) = (INum\ (x +_N (y +_N z)) :: 'a :: \{ring\text{-}char\text{-}0,$ $division\text{-}by\text{-}zero,field\})$ **by** *simp*
with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

lemma *Nsub0*: **assumes** *x*: *isnormNum* x **and** *y*:*isnormNum* y **shows** $(x -_N y = 0_N) = (x = y)$
proof−
{**fix** $h :: 'a :: \{ring\text{-}char\text{-}0,division\text{-}by\text{-}zero,ordered\text{-}field\}$
from *isnormNum-unique*[**where** $?'a = 'a$, *OF Nsub-normN*[*OF y*], **where** $y=0_N$]
have $(x -_N y = 0_N) = (INum\ (x -_N y) = (INum\ 0_N :: 'a))$ **by** *simp*
also have $\ldots = (INum\ x = (INum\ y :: 'a))$ **by** *simp*
also have $\ldots = (x = y)$ **using** *x y* **by** *simp*
finally show *?thesis* **.**}
qed

lemma *Nmul0*[*simp*]: $c *_N 0_N = 0_N$ $0_N *_N c = 0_N$
by (*simp-all add*: *Nmul-def Let-def split-def*)

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−
{**fix** $h :: 'a :: \{ring\text{-}char\text{-}0,division\text{-}by\text{-}zero,ordered\text{-}field\}$
have $\exists\ a\ b\ a'\ b'.\ x = (a,b) \land y= (a',b')$ **by** *auto*
then obtain $a\ b\ a'\ b'$ **where** *xy*[*simp*]: $x = (a,b)$ $y = (a',b')$ **by** *blast*
have *n0*: *isnormNum* 0_N **by** *simp*
show *?thesis* **using** *nx ny*
apply (*simp only*: *isnormNum-unique*[**where** $?'a = 'a$, *OF Nmul-normN*[*OF nx ny*] *n0*, *symmetric*] *Nmul*[**where** $?'a = 'a$])

apply (*simp add*: *INum-def split-def isnormNum-def fst-conv snd-conv*)
apply (*cases* $a{=}0$,*simp-all*)
apply (*cases* $a'{=}0$,*simp-all*)
done }
qed
lemma *Nneg-Nneg*[*simp*]: $\sim_N (\sim_N c) = c$
by (*simp add*: *Nneg-def split-def*)

lemma *Nmul1*[*simp*]:
isnormNum $c \Longrightarrow 1_N *_N c = c$
isnormNum $c \Longrightarrow c *_N 1_N = c$
apply (*simp-all add*: *Nmul-def Let-def split-def isnormNum-def*)
by (*cases fst* $c = 0$, *simp-all*,*cases* c, *simp-all*)+

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

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

function
updates :: $'key$ *list* $\Rightarrow 'val$ *list* $\Rightarrow ('key \times 'val)$ *list* $\Rightarrow ('key \times 'val)$ *list*
where
updates [] *vs ps* = *ps*
| *updates* (*k*#*ks*) *vs ps* = (*case vs*
of [] ⇒ *ps*
| (*v*#*vs'*) ⇒ *updates ks vs'* (*update k v ps*))
by *pat-completeness auto*
termination by *lexicographic-order*

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

lemma *length-delete-le*: *length* (*delete k al*) ≤ *length al*
proof (*induct al*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons a al*)
  note *length-filter-le* [*of* λ*p*. *fst p* ≠ *fst a al*]
  also have ⋀*n*. *n* ≤ *Suc n*
    by *simp*
  finally have *length* [*p*←*al* . *fst p* ≠ *fst a*] ≤ *Suc* (*length al*) **.**
  with *Cons* **show** *?case*
    by *auto*
qed

lemma *compose-hint* [*simp*]:
  length (*delete k al*) < *Suc* (*length al*)
proof −
  note *length-delete-le*
  also have ⋀*n*. *n* < *Suc n*
    by *simp*
  finally show *?thesis* **.**
qed

function
  compose :: ('*key* × '*a*) *list* ⇒ ('*a* × '*b*) *list* ⇒ ('*key* × '*b*) *list*
where
    compose [] *ys* = []
  | *compose* (*x*#*xs*) *ys* = (*case map-of ys* (*snd x*)
      of None ⇒ *compose* (*delete* (*fst x*) *xs*) *ys*
       | *Some v* ⇒ (*fst x*, *v*) # *compose xs ys*)
by *pat-completeness auto*
termination by *lexicographic-order*

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

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

fun
clearjunk :: $('key \times 'val)$ *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-of [] *k* = *None*
map-of (*p#ps*) *k* = (*if fst p* = *k then Some* (*snd p*) *else map-of ps k*)
by *simp-all*

3.2 *delete*

lemma *delete-def*:
delete k xs = *filter* (λp. *fst p* $\neq$ *k*) *xs*
by (*induct xs*) *auto*

lemma *delete-id* [*simp*]: *k* $\notin$ *fst* ‘ *set al* $\Longrightarrow$ *delete k al* = *al*
by (*induct al*) *auto*

lemma *delete-conv*: *map-of* (*delete k al*) *k'* = ((*map-of al*)(*k* := *None*)) *k'*
by (*induct al*) *auto*

lemma *delete-conv'*: *map-of* (*delete k al*) = ((*map-of al*)(*k* := *None*))
by (*rule ext*) (*rule delete-conv*)

lemma *delete-idem*: *delete k* (*delete k al*) = *delete k al*
by (*induct al*) *auto*

lemma *map-of-delete* [*simp*]:
k' $\neq$ *k* $\Longrightarrow$ *map-of* (*delete k al*) *k'* = *map-of al k'*
by (*induct al*) *auto*

lemma *delete-notin-dom*: *k* $\notin$ *fst* ‘ *set* (*delete k al*)
by (*induct al*) *auto*

lemma *dom-delete-subset*: *fst* ‘ *set* (*delete k al*) $\subseteq$ *fst* ‘ *set al*
by (*induct al*) *auto*

lemma *distinct-delete*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*delete k al*))
using *assms*
proof (*induct al*)
case *Nil* **thus** *?case* **by** *simp*

```
next
  case (Cons a al)
  from Cons.prems obtain
    a-notin-al: fst a ∉ fst ‘ set al and
    dist-al: distinct (map fst al)
    by auto
  show ?case
  proof (cases fst a = k)
    case True
    with Cons dist-al show ?thesis by simp
  next
    case False
    from dist-al
    have distinct (map fst (delete k al))
      by (rule Cons.hyps)
    moreover from a-notin-al dom-delete-subset [of k al]
    have fst a ∉ fst ‘ set (delete k al)
      by blast
    ultimately show ?thesis using False by simp
  qed
qed

lemma delete-twist: delete x (delete y al) = delete y (delete x al)
  by (induct al) auto

lemma clearjunk-delete: clearjunk (delete x al) = delete x (clearjunk al)
  by (induct al rule: clearjunk.induct) (auto simp add: delete-idem delete-twist)
```

3.3 *clearjunk*

```
lemma insert-fst-filter:
  insert a(fst ‘ {x ∈ set ps. fst x ≠ a}) = insert a (fst ‘ set ps)
  by (induct ps) auto

lemma dom-clearjunk: fst ‘ set (clearjunk al) = fst ‘ set al
  by (induct al rule: clearjunk.induct) (simp-all add: insert-fst-filter delete-def)

lemma notin-filter-fst: a ∉ fst ‘ {x ∈ set ps. fst x ≠ a}
  by (induct ps) auto

lemma distinct-clearjunk [simp]: distinct (map fst (clearjunk al))
  by (induct al rule: clearjunk.induct)
    (simp-all add: dom-clearjunk notin-filter-fst delete-def)

lemma map-of-filter: k ≠ a ⟹ map-of [q←ps . fst q ≠ a] k = map-of ps k
  by (induct ps) auto

lemma map-of-clearjunk: map-of (clearjunk al) = map-of al
  apply (rule ext)
```

apply (*induct al rule*: *clearjunk.induct*)
apply *simp*
apply (*simp add*: *map-of-filter*)
done

lemma *length-clearjunk*: *length* (*clearjunk al*) $\leq$ *length al*
proof (*induct al rule*: *clearjunk.induct* [*case-names Nil Cons*])
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons p ps*)
from *Cons* **have** *length* (*clearjunk* [*q*$\leftarrow$*ps* . *fst q* $\neq$ *fst p*]) $\leq$ *length* [*q*$\leftarrow$*ps* . *fst q* $\neq$ *fst p*]
by (*simp add*: *delete-def*)
also have ... $\leq$ *length ps*
by *simp*
finally show *?case*
by (*simp add*: *delete-def*)
qed

lemma *notin-fst-filter*: *a* $\notin$ *fst* ' *set ps* $\Longrightarrow$ [*q*$\leftarrow$*ps* . *fst q* $\neq$ *a*] = *ps*
by (*induct ps*) *auto*

lemma *distinct-clearjunk-id* [*simp*]: *distinct* (*map fst al*) $\Longrightarrow$ *clearjunk al* = *al*
by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *notin-fst-filter*)

lemma *clearjunk-idem*: *clearjunk* (*clearjunk al*) = *clearjunk al*
by *simp*

3.4 *dom* and *ran*

lemma *dom-map-of'*: *fst* ' *set al* = *dom* (*map-of al*)
by (*induct al*) *auto*

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

lemma *ran-clearjunk*: *ran* (*map-of* (*clearjunk al*)) = *ran* (*map-of al*)
by (*simp add*: *map-of-clearjunk*)

lemma *ran-distinct*:
assumes *dist*: *distinct* (*map fst al*)
shows *ran* (*map-of al*) = *snd* ' *set al*
using *dist*
proof (*induct al*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons a al*)
hence *hyp*: *snd* ' *set al* = *ran* (*map-of al*)
by *simp*

```
  have ran (map-of (a # al)) = {snd a} ∪ ran (map-of al)
  proof
    show ran (map-of (a # al)) ⊆ {snd a} ∪ ran (map-of al)
    proof
      fix v
      assume v ∈ ran (map-of (a#al))
      then obtain x where map-of (a#al) x = Some v
        by (auto simp add: ran-def)
      then show v ∈ {snd a} ∪ ran (map-of al)
        by (auto split: split-if-asm simp add: ran-def)
    qed
  next
    show {snd a} ∪ ran (map-of al) ⊆ ran (map-of (a # al))
    proof
      fix v
      assume v-in: v ∈ {snd a} ∪ ran (map-of al)
      show v ∈ ran (map-of (a#al))
      proof (cases v=snd a)
        case True
        with v-in show ?thesis
          by (auto simp add: ran-def)
      next
        case False
        with v-in have v ∈ ran (map-of al) by auto
        then obtain x where al-x: map-of al x = Some v
          by (auto simp add: ran-def)
        from map-of-SomeD [OF this]
        have x ∈ fst ` set al
          by (force simp add: image-def)
        with Cons.prems have x≠fst a
          by − (rule ccontr,simp)
        with al-x
        show ?thesis
          by (auto simp add: ran-def)
      qed
    qed
  qed
  with hyp show ?case
    by (simp only:) auto
qed

lemma ran-map-of: ran (map-of al) = snd ` set (clearjunk al)
proof −
  have ran (map-of al) = ran (map-of (clearjunk al))
    by (simp add: ran-clearjunk)
  also have . . . = snd ` set (clearjunk al)
    by (simp add: ran-distinct)
  finally show ?thesis .
```

qed

3.5 *update*

lemma *update-conv*: *map-of* (*update k v al*) k' = ((*map-of al*)($k \mapsto v$)) k'
  by (*induct al*) *auto*

lemma *update-conv'*: *map-of* (*update k v al*) = ((*map-of al*)($k \mapsto v$))
  by (*rule ext*) (*rule update-conv*)

lemma *dom-update*: *fst* ' *set* (*update k v al*) = $\{k\} \cup$ *fst* ' *set al*
  by (*induct al*) *auto*

lemma *distinct-update*:
  assumes *distinct* (*map fst al*)
  shows *distinct* (*map fst* (*update k v al*))
using *assms*
proof (*induct al*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons a al*)
  from *Cons.prems* **obtain**
    a-notin-al: *fst a* $\notin$ *fst* ' *set al* **and**
    dist-al: *distinct* (*map fst al*)
    by *auto*
  show *?case*
  proof (*cases fst a* = *k*)
    case *True*
    from *True dist-al a-notin-al* **show** *?thesis* **by** *simp*
  next
    case *False*
    from *dist-al*
    have *distinct* (*map fst* (*update k v al*))
      by (*rule Cons.hyps*)
    with *False a-notin-al* **show** *?thesis* **by** (*simp add*: *dom-update*)
  qed
qed

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*]
  by (*induct ps*) *auto*

lemma *clearjunk-update*: *clearjunk* (*update k v al*) = *update k v* (*clearjunk al*)
  by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *update-filter delete-def*)

lemma *update-triv*: *map-of al k* = *Some v* $\Longrightarrow$ *update k v al* = *al*
  by (*induct al*) *auto*

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

by (*induct al*) *auto*

lemma *update-eqD*: *update k v al* = *update k v′ al′* $\Longrightarrow$ *v*=*v′*
proof (*induct al arbitrary*: *al′*)
  case *Nil* **thus** *?case*
    by (*cases al′*) (*auto split*: *split-if-asm*)
next
  case *Cons* **thus** *?case*
    by (*cases al′*) (*auto split*: *split-if-asm*)
qed

lemma *update-last* [*simp*]: *update k v* (*update k v′ al*) = *update k v al*
  by (*induct al*) *auto*

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-of* (*update k v* (*update k′ v′ al*)) = *map-of* (*update k′ v′* (*update k v al*))
  by (*auto simp add*: *update-conv′ intro*: *ext*)

lemma *update-Some-unfold*:
  (*map-of* (*update k v al*) *x* = *Some y*) =
    (*x* = *k* $\wedge$ *v* = *y* $\vee$ *x* $\neq$ *k* $\wedge$ *map-of al x* = *Some y*)
  by (*simp add*: *update-conv′ map-upd-Some-unfold*)

lemma *image-update*[*simp*]: *x* $\notin$ *A* $\Longrightarrow$ *map-of* (*update x y al*) ‘ *A* = *map-of al* ‘ *A*
  by (*simp add*: *update-conv′ image-map-upd*)

3.6 *updates*

lemma *updates-conv*: *map-of* (*updates ks vs al*) *k* = ((*map-of al*)(*ks*[$\mapsto$]*vs*)) *k*
proof (*induct ks arbitrary*: *vs al*)
  case *Nil*
  thus *?case* **by** *simp*
next
  case (*Cons k ks*)
  show *?case*
  proof (*cases vs*)
    case *Nil*
    with *Cons* **show** *?thesis* **by** *simp*
  next
    case (*Cons k ks′*)
    with *Cons.hyps* **show** *?thesis*
      by (*simp add*: *update-conv fun-upd-def*)
  qed
qed

```
lemma updates-conv': map-of (updates ks vs al) = ((map-of al)(ks[↦]vs))
  by (rule ext) (rule updates-conv)

lemma distinct-updates:
  assumes distinct (map fst al)
  shows distinct (map fst (updates ks vs al))
  using assms
  by (induct ks arbitrary: vs al)
   (auto simp add: distinct-update split: list.splits)

lemma clearjunk-updates:
 clearjunk (updates ks vs al) = updates ks vs (clearjunk al)
  by (induct ks arbitrary: vs al) (auto simp add: clearjunk-update split: list.splits)

lemma updates-empty[simp]: updates vs [] al = al
  by (induct vs) auto

lemma updates-Cons: updates (k#ks) (v#vs) al = updates ks vs (update k v al)
  by simp

lemma updates-append1[simp]: size ks < size vs ⟹
  updates (ks@[k]) vs al = update k (vs!size ks) (updates ks vs al)
  by (induct ks arbitrary: vs al) (auto split: list.splits)

lemma updates-list-update-drop[simp]:
 ⟦size ks ≤ i; i < size vs⟧
   ⟹ updates ks (vs[i:=v]) al = updates ks vs al
  by (induct ks arbitrary: al vs i) (auto split:list.splits nat.splits)

lemma update-updates-conv-if:
 map-of (updates xs ys (update x y al)) =
 map-of (if x ∈  set(take (length ys) xs) then updates xs ys al
                                  else (update x y (updates xs ys al)))
  by (simp add: updates-conv' update-conv' map-upd-upds-conv-if)

lemma updates-twist [simp]:
 k ∉ set ks ⟹
  map-of (updates ks vs (update k v al)) = map-of (update k v (updates ks vs al))
  by (simp add: updates-conv' update-conv' map-upds-twist)

lemma updates-apply-notin[simp]:
 k ∉ set ks ==> map-of (updates ks vs al) k = map-of al k
  by (simp add: updates-conv)

lemma updates-append-drop[simp]:
  size xs = size ys ⟹ updates (xs@zs) ys al = updates xs ys al
  by (induct xs arbitrary: ys al) (auto split: list.splits)

lemma updates-append2-drop[simp]:
```

size xs = *size ys* $\Longrightarrow$ *updates xs* (*ys@zs*) *al* = *updates xs ys al*
by (*induct xs arbitrary*: *ys al*) (*auto split*: *list.splits*)

3.7 *map-ran*

lemma *map-ran-conv*: *map-of* (*map-ran f al*) *k* = *option-map* (*f k*) (*map-of al k*)
by (*induct al*) *auto*

lemma *dom-map-ran*: *fst* ‘ *set* (*map-ran f al*) = *fst* ‘ *set al*
by (*induct al*) *auto*

lemma *distinct-map-ran*: *distinct* (*map fst al*) $\Longrightarrow$ *distinct* (*map fst* (*map-ran f al*))
by (*induct al*) (*auto simp add*: *dom-map-ran*)

lemma *map-ran-filter*: *map-ran f* [*p*←*ps*. *fst p* ≠ *a*] = [*p*←*map-ran f ps*. *fst p* ≠ *a*]
by (*induct ps*) *auto*

lemma *clearjunk-map-ran*: *clearjunk* (*map-ran f al*) = *map-ran f* (*clearjunk al*)
by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *delete-def map-ran-filter*)

3.8 *merge*

lemma *dom-merge*: *fst* ‘ *set* (*merge xs ys*) = *fst* ‘ *set xs* ∪ *fst* ‘ *set ys*
by (*induct ys arbitrary*: *xs*) (*auto simp add*: *dom-update*)

lemma *distinct-merge*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*merge xs ys*))
using *assms*
by (*induct ys arbitrary*: *xs*) (*auto simp add*: *dom-merge distinct-update*)

lemma *clearjunk-merge*:
clearjunk (*merge xs ys*) = *merge* (*clearjunk xs*) *ys*
by (*induct ys*) (*auto simp add*: *clearjunk-update*)

lemma *merge-conv*: *map-of* (*merge xs ys*) *k* = (*map-of xs* ++ *map-of ys*) *k*
proof (*induct ys*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons y ys*)
show *?case*
proof (*cases k* = *fst y*)
case *True*
from *True* **show** *?thesis*
by (*simp add*: *update-conv*)
next
case *False*
from *False* **show** *?thesis*

```
    by (auto simp add: update-conv Cons.hyps map-add-def)
  qed
qed

lemma merge-conv': map-of (merge xs ys) = (map-of xs ++ map-of ys)
  by (rule ext) (rule merge-conv)

lemma merge-emty: map-of (merge [] ys) = map-of ys
  by (simp add: merge-conv')

lemma merge-assoc[simp]: map-of (merge m1 (merge m2 m3)) =
                          map-of (merge (merge m1 m2) m3)
  by (simp add: merge-conv')

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)
  by (simp add: merge-conv' map-add-Some-iff)

lemmas merge-SomeD = merge-Some-iff [THEN iffD1, standard]
declare merge-SomeD [dest!]

lemma merge-find-right[simp]: map-of n k = Some v ⟹ map-of (merge m n) k
= Some v
  by (simp add: merge-conv')

lemma merge-None [iff]:
  (map-of (merge m n) k = None) = (map-of n k = None ∧ map-of m k = None)
  by (simp add: merge-conv')

lemma merge-upd[simp]:
  map-of (merge m (update k v n)) = map-of (update k v (merge m n))
  by (simp add: update-conv' merge-conv')

lemma merge-updatess[simp]:
  map-of (merge m (updates xs ys n)) = map-of (updates xs ys (merge m n))
  by (simp add: updates-conv' merge-conv')

lemma merge-append: map-of (xs@ys) = map-of (merge ys xs)
  by (simp add: merge-conv')
```

3.9 *compose*

```
lemma compose-first-None [simp]:
  assumes map-of xs k = None
  shows map-of (compose xs ys) k = None
using assms by (induct xs ys rule: compose.induct)
  (auto split: option.splits split-if-asm)
```

```
lemma compose-conv:
  shows map-of (compose xs ys) k = (map-of ys ∘m map-of xs) k
proof (induct xs ys rule: compose.induct)
  case 1 then show ?case by simp
next
  case (2 x xs ys) show ?case
  proof (cases map-of ys (snd x))
    case None with 2
    have hyp: map-of (compose (delete (fst x) xs) ys) k =
              (map-of ys ∘m map-of (delete (fst x) xs)) k
      by simp
    show ?thesis
    proof (cases fst x = k)
      case True
      from True delete-notin-dom [of k xs]
      have map-of (delete (fst x) xs) k = None
        by (simp add: map-of-eq-None-iff)
      with hyp show ?thesis
        using True None
        by simp
    next
      case False
      from False have map-of (delete (fst x) xs) k = map-of xs k
        by simp
      with hyp show ?thesis
        using False None
        by (simp add: map-comp-def)
    qed
  next
    case (Some v)
    with 2
    have map-of (compose xs ys) k = (map-of ys ∘m map-of xs) k
      by simp
    with Some show ?thesis
      by (auto simp add: map-comp-def)
  qed
qed

lemma compose-conv':
  shows map-of (compose xs ys) = (map-of ys ∘m map-of xs)
  by (rule ext) (rule compose-conv)

lemma compose-first-Some [simp]:
  assumes map-of xs k = Some v
  shows map-of (compose xs ys) k = map-of ys v
using assms by (simp add: compose-conv)

lemma dom-compose: fst ` set (compose xs ys) ⊆ fst ` set xs
proof (induct xs ys rule: compose.induct)
```

```
  case 1 thus ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2.hyps
    have fst ‘ set (compose (delete (fst x) xs) ys) ⊆ fst ‘ set (delete (fst x) xs)
      by simp
    also
    have ... ⊆ fst ‘ set xs
      by (rule dom-delete-subset)
    finally show ?thesis
      using None
      by auto
  next
    case (Some v)
    with 2.hyps
    have fst ‘ set (compose xs ys) ⊆ fst ‘ set xs
      by simp
    with Some show ?thesis
      by auto
  qed
qed

lemma distinct-compose:
 assumes distinct (map fst xs)
 shows distinct (map fst (compose xs ys))
using assms
proof (induct xs ys rule: compose.induct)
  case 1 thus ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2 show ?thesis by simp
  next
    case (Some v)
    with 2 dom-compose [of xs ys] show ?thesis
      by (auto)
  qed
qed

lemma compose-delete-twist: (compose (delete k xs) ys) = delete k (compose xs
ys)
proof (induct xs ys rule: compose.induct)
  case 1 thus ?case by simp
next
```

```
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2 have
      hyp: compose (delete k (delete (fst x) xs)) ys =
            delete k (compose (delete (fst x) xs) ys)
      by simp
    show ?thesis
    proof (cases fst x = k)
      case True
      with None hyp
      show ?thesis
        by (simp add: delete-idem)
    next
      case False
      from None False hyp
      show ?thesis
        by (simp add: delete-twist)
    qed
  next
    case (Some v)
    with 2 have hyp: compose (delete k xs) ys = delete k (compose xs ys) by simp
    with Some show ?thesis
      by simp
  qed
qed

lemma compose-clearjunk: compose xs (clearjunk ys) = compose xs ys
  by (induct xs ys rule: compose.induct)
     (auto simp add: map-of-clearjunk split: option.splits)

lemma clearjunk-compose: clearjunk (compose xs ys) = compose (clearjunk xs) ys
  by (induct xs rule: clearjunk.induct)
     (auto split: option.splits simp add: clearjunk-delete delete-idem
          compose-delete-twist)

lemma compose-empty [simp]:
 compose xs [] = []
  by (induct xs) (auto simp add: compose-delete-twist)

lemma compose-Some-iff:
  (map-of (compose xs ys) k = Some v) =
     (∃ k'. map-of xs k = Some k' ∧ map-of ys k' = Some v)
  by (simp add: compose-conv map-comp-Some-iff)

lemma map-comp-None-iff:
  (map-of (compose xs ys) k = None) =
    (map-of xs k = None ∨ (∃ k'. map-of xs k = Some k' ∧ map-of ys k' = None))
```

by (*simp add*: *compose-conv map-comp-None-iff*)

3.10 *restrict*

lemma *restrict-def*:
restrict A = *filter* (λp. *fst p* $\in$ *A*)
proof
fix *xs*
show *restrict A xs* = *filter* (λp. *fst p* $\in$ *A*) *xs*
by (*induct xs*) *auto*
qed

lemma *distinct-restr*: *distinct* (*map fst al*) $\Longrightarrow$ *distinct* (*map fst* (*restrict A al*))
by (*induct al*) (*auto simp add*: *restrict-def*)

lemma *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|` *A*) *k*
apply (*induct al*)
apply (*simp add*: *restrict-def*)
apply (*cases k*$\in$*A*)
apply (*auto simp add*: *restrict-def*)
done

lemma *restr-conv'*: *map-of* (*restrict A al*) = ((*map-of al*)|` *A*)
by (*rule ext*) (*rule restr-conv*)

lemma *restr-empty* [*simp*]:
restrict {} *al* = []
restrict A [] = []
by (*induct al*) (*auto simp add*: *restrict-def*)

lemma *restr-in* [*simp*]: $x \in A \Longrightarrow$ *map-of* (*restrict A al*) *x* = *map-of al x*
by (*simp add*: *restr-conv'*)

lemma *restr-out* [*simp*]: $x \notin A \Longrightarrow$ *map-of* (*restrict A al*) *x* = *None*
by (*simp add*: *restr-conv'*)

lemma *dom-restr* [*simp*]: *fst* ` *set* (*restrict A al*) = *fst* ` *set al* $\cap$ *A*
by (*induct al*) (*auto simp add*: *restrict-def*)

lemma *restr-upd-same* [*simp*]: *restrict* ($-\{x\}$) (*update x y al*) = *restrict* ($-\{x\}$) *al*
by (*induct al*) (*auto simp add*: *restrict-def*)

lemma *restr-restr* [*simp*]: *restrict A* (*restrict B al*) = *restrict* ($A \cap B$) *al*
by (*induct al*) (*auto simp add*: *restrict-def*)

lemma *restr-update*[*simp*]:
map-of (*restrict D* (*update x y al*)) =

```
  map-of ((if x ∈ D then (update x y (restrict (D−{x}) al)) else restrict D al))
  by (simp add: restr-conv′ update-conv′)

lemma restr-delete [simp]:
  (delete x (restrict D al)) =
    (if x∈ D then restrict (D − {x}) al else restrict D al)
proof (induct al)
  case Nil thus ?case by simp
next
  case (Cons a al)
  show ?case
  proof (cases x ∈ D)
    case True
    note x-D = this
    with Cons have hyp: delete x (restrict D al) = restrict (D − {x}) al
      by simp
    show ?thesis
    proof (cases fst a = x)
      case True
      from Cons.hyps
      show ?thesis
        using x-D True
        by simp
    next
      case False
      note not-fst-a-x = this
      show ?thesis
      proof (cases fst a ∈ D)
        case True
        with not-fst-a-x
        have delete x (restrict D (a#al)) = a#(delete x (restrict D al))
          by (cases a) (simp add: restrict-def)
        also from not-fst-a-x True hyp have ... = restrict (D − {x}) (a # al)
          by (cases a) (simp add: restrict-def)
        finally show ?thesis
          using x-D by simp
      next
        case False
        hence delete x (restrict D (a#al)) = delete x (restrict D al)
          by (cases a) (simp add: restrict-def)
        moreover from False not-fst-a-x
        have restrict (D − {x}) (a # al) = restrict (D − {x}) al
          by (cases a) (simp add: restrict-def)
        ultimately
        show ?thesis using x-D hyp by simp
      qed
    qed
  next
    case False
```

from *False Cons* **show** *?thesis*
by *simp*
qed
qed

lemma *update-restr*:
$map\text{-}of\ (update\ x\ y\ (restrict\ D\ al)) = map\text{-}of\ (update\ x\ y\ (restrict\ (D-\{x\})\ al))$
by (*simp add*: *update-conv′ restr-conv′*) (*rule fun-upd-restrict*)

lemma *upate-restr-conv* [*simp*]:
$x \in D \Longrightarrow$
$map\text{-}of\ (update\ x\ y\ (restrict\ D\ al)) = map\text{-}of\ (update\ x\ y\ (restrict\ (D-\{x\})\ al))$
by (*simp add*: *update-conv′ restr-conv′*)

lemma *restr-updates* [*simp*]:
$[\![\ length\ xs = length\ ys;\ set\ xs \subseteq D\ ]\!]$
$\Longrightarrow map\text{-}of\ (restrict\ D\ (updates\ xs\ ys\ al)) =$
$map\text{-}of\ (updates\ xs\ ys\ (restrict\ (D - set\ xs)\ al))$
by (*simp add*: *updates-conv′ restr-conv′*)

lemma *restr-delete-twist*: $(restrict\ A\ (delete\ a\ ps)) = delete\ a\ (restrict\ A\ ps)$
by (*induct ps*) *auto*

lemma *clearjunk-restrict*:
$clearjunk\ (restrict\ A\ al) = restrict\ A\ (clearjunk\ al)$
by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *restr-delete-twist*)

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* **..**
instance *fun* :: (*type*, *plus*) *plus* **..**

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 **..**
instance fun :: (type, times) times **..**

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 **..**

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

instance fun :: (type, zero) zero **..**
instance set :: (zero) zero **..**

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

instance fun :: (type, one) one **..**
instance set :: (one) one **..**

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
by default (auto simp add: func-plus add-assoc)

instance fun :: (type,comm-monoid-add)comm-monoid-add
by default (auto simp add: func-zero func-plus add-ac)

instance fun :: (type,ab-group-add)ab-group-add
apply default

apply (*simp add*: *func-minus func-plus func-zero*)
apply (*simp add*: *func-minus func-plus func-diff diff-minus*)
done

instance *fun* :: (*type*,*semigroup-mult*)*semigroup-mult*
apply *default*
apply (*auto simp add*: *func-times mult-assoc*)
done

instance *fun* :: (*type*,*comm-monoid-mult*)*comm-monoid-mult*
apply *default*
apply (*auto simp add*: *func-one func-times mult-ac*)
done

instance *fun* :: (*type*,*comm-ring-1*)*comm-ring-1*
apply *default*
apply (*auto simp add*: *func-plus func-times func-minus func-diff ext func-one func-zero ring-simps*)
apply (*drule fun-cong*)
apply *simp*
done

instance *set* :: (*semigroup-add*)*semigroup-add*
apply *default*
apply (*unfold set-plus*)
apply (*force simp add*: *add-assoc*)
done

instance *set* :: (*semigroup-mult*)*semigroup-mult*
apply *default*
apply (*unfold set-times*)
apply (*force simp add*: *mult-assoc*)
done

instance *set* :: (*comm-monoid-add*)*comm-monoid-add*
apply *default*
apply (*unfold set-plus*)
apply (*force simp add*: *add-ac*)
apply (*unfold set-zero*)
apply *force*
done

instance *set* :: (*comm-monoid-mult*)*comm-monoid-mult*
apply *default*
apply (*unfold set-times*)
apply (*force simp add*: *mult-ac*)
apply (*unfold set-one*)
apply *force*
done

4.2 Basic properties

lemma *set-plus-intro* [*intro*]: *a* : *C* ==> *b* : *D* ==> *a* + *b* : *C* + *D*
by (*auto simp add*: *set-plus*)

lemma *set-plus-intro2* [*intro*]: *b* : *C* ==> *a* + *b* : *a* +*o* *C*
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-rearrange*: ((*a*::'*a*::*comm-monoid-add*) +*o* *C*) +
(*b* +*o* *D*) = (*a* + *b*) +*o* (*C* + *D*)
apply (*auto simp add*: *elt-set-plus-def set-plus add-ac*)
apply (*rule-tac x* = *ba* + *bb* **in** *exI*)
apply (*auto simp add*: *add-ac*)
apply (*rule-tac x* = *aa* + *a* **in** *exI*)
apply (*auto simp add*: *add-ac*)
done

lemma *set-plus-rearrange2*: (*a*::'*a*::*semigroup-add*) +*o* (*b* +*o* *C*) =
(*a* + *b*) +*o* *C*
by (*auto simp add*: *elt-set-plus-def add-assoc*)

lemma *set-plus-rearrange3*: ((*a*::'*a*::*semigroup-add*) +*o* *B*) + *C* =
a +*o* (*B* + *C*)
apply (*auto simp add*: *elt-set-plus-def set-plus*)
apply (*blast intro*: *add-ac*)
apply (*rule-tac x* = *a* + *aa* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x* = *aa* **in** *bexI*)
apply *auto*
apply (*rule-tac x* = *ba* **in** *bexI*)
apply (*auto simp add*: *add-ac*)
done

theorem *set-plus-rearrange4*: *C* + ((*a*::'*a*::*comm-monoid-add*) +*o* *D*) =
a +*o* (*C* + *D*)
apply (*auto intro*!: *subsetI simp add*: *elt-set-plus-def set-plus add-ac*)
apply (*rule-tac x* = *aa* + *ba* **in** *exI*)
apply (*auto simp add*: *add-ac*)
done

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*
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-mono2* [*intro*]: (*C*::('*a*::*plus*) *set*) <= *D* ==> *E* <= *F* ==>
C + *E* <= *D* + *F*
by (*auto simp add*: *set-plus*)

```
lemma set-plus-mono3 [intro]: a : C ==> a +o D <= C + D
  by (auto simp add: elt-set-plus-def set-plus)

lemma set-plus-mono4 [intro]: (a::'a::comm-monoid-add) : C ==>
    a +o D <= D + C
  by (auto simp add: elt-set-plus-def set-plus add-ac)

lemma set-plus-mono5: a:C ==> B <= D ==> a +o B <= C + D
  apply (subgoal-tac a +o B <= a +o D)
   apply (erule order-trans)
   apply (erule set-plus-mono3)
  apply (erule set-plus-mono)
  done

lemma set-plus-mono-b: C <= D ==> x : a +o C
    ==> x : a +o D
  apply (frule set-plus-mono)
  apply auto
  done

lemma set-plus-mono2-b: C <= D ==> E <= F ==> x : C + E ==>
    x : D + F
  apply (frule set-plus-mono2)
   prefer 2
   apply force
  apply assumption
  done

lemma set-plus-mono3-b: a : C ==> x : a +o D ==> x : C + D
  apply (frule set-plus-mono3)
  apply auto
  done

lemma set-plus-mono4-b: (a::'a::comm-monoid-add) : C ==>
    x : a +o D ==> x : D + C
  apply (frule set-plus-mono4)
  apply auto
  done

lemma set-zero-plus [simp]: (0::'a::comm-monoid-add) +o C = C
  by (auto simp add: elt-set-plus-def)

lemma set-zero-plus2: (0::'a::comm-monoid-add) : A ==> B <= A + B
  apply (auto intro!: subsetI simp add: set-plus)
  apply (rule-tac x = 0 in bexI)
   apply (rule-tac x = x in bexI)
    apply (auto simp add: add-ac)
  done
```

lemma *set-plus-imp-minus*: $(a::'a::ab\text{-}group\text{-}add) : b +o\ C ==> (a - b) : C$
  by (*auto simp add*: *elt-set-plus-def add-ac diff-minus*)

lemma *set-minus-imp-plus*: $(a::'a::ab\text{-}group\text{-}add) - b : C ==> a : b +o\ C$
  apply (*auto simp add*: *elt-set-plus-def add-ac diff-minus*)
  apply (*subgoal-tac* $a = (a + - b) + b$)
   apply (*rule bexI*, *assumption*, *assumption*)
  apply (*auto simp add*: *add-ac*)
  done

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

lemma *set-times-intro* [*intro*]: $a : C ==> b : D ==> a * b : C * D$
  by (*auto simp add*: *set-times*)

lemma *set-times-intro2* [*intro*!]: $b : C ==> a * b : a *o\ C$
  by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-rearrange*: $((a::'a::comm\text{-}monoid\text{-}mult) *o\ C) * (b *o\ D) = (a * b) *o\ (C * D)$
  apply (*auto simp add*: *elt-set-times-def set-times*)
   apply (*rule-tac* $x = ba * bb$ **in** *exI*)
   apply (*auto simp add*: *mult-ac*)
  apply (*rule-tac* $x = aa * a$ **in** *exI*)
  apply (*auto simp add*: *mult-ac*)
  done

lemma *set-times-rearrange2*: $(a::'a::semigroup\text{-}mult) *o\ (b *o\ C) = (a * b) *o\ C$
  by (*auto simp add*: *elt-set-times-def mult-assoc*)

lemma *set-times-rearrange3*: $((a::'a::semigroup\text{-}mult) *o\ B) * C = a *o\ (B * C)$
  apply (*auto simp add*: *elt-set-times-def set-times*)
   apply (*blast intro*: *mult-ac*)
  apply (*rule-tac* $x = a * aa$ **in** *exI*)
  apply (*rule conjI*)
   apply (*rule-tac* $x = aa$ **in** *bexI*)
    apply *auto*
  apply (*rule-tac* $x = ba$ **in** *bexI*)
   apply (*auto simp add*: *mult-ac*)
  done

theorem *set-times-rearrange4*: $C * ((a::'a::comm\text{-}monoid\text{-}mult) *o\ D) = a *o\ (C * D)$
  apply (*auto intro*!: *subsetI simp add*: *elt-set-times-def set-times mult-ac*)

```
   apply (rule-tac x = aa * ba in exI)
   apply (auto simp add: mult-ac)
  done

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
  by (auto simp add: elt-set-times-def)

lemma set-times-mono2 [intro]: (C::('a::times) set) <= D ==> E <= F ==>
    C * E <= D * F
  by (auto simp add: set-times)

lemma set-times-mono3 [intro]: a : C ==> a *o D <= C * D
  by (auto simp add: elt-set-times-def set-times)

lemma set-times-mono4 [intro]: (a::'a::comm-monoid-mult) : C ==>
    a *o D <= D * C
  by (auto simp add: elt-set-times-def set-times mult-ac)

lemma set-times-mono5: a:C ==> B <= D ==> a *o B <= C * D
  apply (subgoal-tac a *o B <= a *o D)
   apply (erule order-trans)
   apply (erule set-times-mono3)
  apply (erule set-times-mono)
  done

lemma set-times-mono-b: C <= D ==> x : a *o C
    ==> x : a *o D
  apply (frule set-times-mono)
  apply auto
  done

lemma set-times-mono2-b: C <= D ==> E <= F ==> x : C * E ==>
    x : D * F
  apply (frule set-times-mono2)
   prefer 2
   apply force
  apply assumption
  done

lemma set-times-mono3-b: a : C ==> x : a *o D ==> x : C * D
  apply (frule set-times-mono3)
  apply auto
  done

lemma set-times-mono4-b: (a::'a::comm-monoid-mult) : C ==>
    x : a *o D ==> x : D * C
```

```
  apply (frule set-times-mono4)
  apply auto
  done

lemma set-one-times [simp]: (1::'a::comm-monoid-mult) *o C = C
  by (auto simp add: elt-set-times-def)

lemma set-times-plus-distrib: (a::'a::semiring) *o (b +o C)=
    (a * b) +o (a *o C)
  by (auto simp add: elt-set-plus-def elt-set-times-def ring-distribs)

lemma set-times-plus-distrib2: (a::'a::semiring) *o (B + C) =
    (a *o B) + (a *o C)
  apply (auto simp add: set-plus elt-set-times-def ring-distribs)
   apply blast
  apply (rule-tac x = b + bb in exI)
  apply (auto simp add: ring-distribs)
  done

lemma set-times-plus-distrib3: ((a::'a::semiring) +o C) * D <=
    a *o D + C * D
  apply (auto intro!: subsetI simp add:
    elt-set-plus-def elt-set-times-def set-times
    set-plus ring-distribs)
  apply auto
  done

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

lemma set-neg-intro: (a::'a::ring-1) : (- 1) *o C ==>
    - a : C
  by (auto simp add: elt-set-times-def)

lemma set-neg-intro2: (a::'a::ring-1) : C ==>
    - a : (- 1) *o C
  by (auto simp add: elt-set-times-def)

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*)))))
  apply *auto*
  apply (*case-tac c* = *0*)
  apply *simp*
  apply (*rule-tac x* = *1* **in** *exI*)
  apply *simp*
  apply (*rule-tac x* = *abs c* **in** *exI*)
  apply *auto*
  apply (*subgoal-tac c* ∗ *abs*(*f x*) <= *abs c* ∗ *abs* (*f x*))
  apply (*erule-tac x* = *x* **in** *allE*)
  apply *force*
  apply (*rule mult-right-mono*)
  apply (*rule abs-ge-self*)
  apply (*rule abs-ge-zero*)
  done

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

by (*auto simp add*: *bigo-def bigo-pos-const*)

lemma *bigo-elt-subset* [*intro*]: $f : O(g) ==> O(f) <= O(g)$
apply (*auto simp add*: *bigo-alt-def*)
apply (*rule-tac* $x = ca * c$ **in** *exI*)
apply (*rule conjI*)
apply (*rule mult-pos-pos*)
apply (*assumption*)+
apply (*rule allI*)
apply (*drule-tac* $x = xa$ **in** *spec*)+
apply (*subgoal-tac* $ca * abs(f\ xa) <= ca * (c * abs(g\ xa))$)
apply (*erule order-trans*)
apply (*simp add*: *mult-ac*)
apply (*rule mult-left-mono*, *assumption*)
apply (*rule order-less-imp-le*, *assumption*)
done

lemma *bigo-refl* [*intro*]: $f : O(f)$
apply(*auto simp add*: *bigo-def*)
apply(*rule-tac* $x = 1$ **in** *exI*)
apply *simp*
done

lemma *bigo-zero*: $0 : O(g)$
apply (*auto simp add*: *bigo-def func-zero*)
apply (*rule-tac* $x = 0$ **in** *exI*)
apply *auto*
done

lemma *bigo-zero2*: $O(\%x.0) = \{\%x.0\}$
apply (*auto simp add*: *bigo-def*)
apply (*rule ext*)
apply *auto*
done

lemma *bigo-plus-self-subset* [*intro*]:
$O(f) + O(f) <= O(f)$
apply (*auto simp add*: *bigo-alt-def set-plus*)
apply (*rule-tac* $x = c + ca$ **in** *exI*)
apply *auto*
apply (*simp add*: *ring-distribs func-plus*)
apply (*rule order-trans*)
apply (*rule abs-triangle-ineq*)
apply (*rule add-mono*)
apply *force*
apply *force*
done

lemma *bigo-plus-idemp* [*simp*]: $O(f) + O(f) = O(f)$

```
  apply (rule equalityI)
  apply (rule bigo-plus-self-subset)
  apply (rule set-zero-plus2)
  apply (rule bigo-zero)
  done

lemma bigo-plus-subset [intro]: O(f + g) <= O(f) + O(g)
  apply (rule subsetI)
  apply (auto simp add: bigo-def bigo-pos-const func-plus set-plus)
  apply (subst bigo-pos-const [symmetric])+
  apply (rule-tac x =
    %n. if abs (g n) <= (abs (f n)) then x n else 0 in exI)
  apply (rule conjI)
  apply (rule-tac x = c + c in exI)
  apply (clarsimp)
  apply (auto)
  apply (subgoal-tac c * abs (f xa + g xa) <= (c + c) * abs (f xa))
  apply (erule-tac x = xa in allE)
  apply (erule order-trans)
  apply (simp)
  apply (subgoal-tac c * abs (f xa + g xa) <= c * (abs (f xa) + abs (g xa)))
  apply (erule order-trans)
  apply (simp add: ring-distribs)
  apply (rule mult-left-mono)
  apply assumption
  apply (simp add: order-less-le)
  apply (rule mult-left-mono)
  apply (simp add: abs-triangle-ineq)
  apply (simp add: order-less-le)
  apply (rule mult-nonneg-nonneg)
  apply (rule add-nonneg-nonneg)
  apply auto
  apply (rule-tac x = %n. if (abs (f n)) <  abs (g n) then x n else 0
    in exI)
  apply (rule conjI)
  apply (rule-tac x = c + c in exI)
  apply auto
  apply (subgoal-tac c * abs (f xa + g xa) <= (c + c) * abs (g xa))
  apply (erule-tac x = xa in allE)
  apply (erule order-trans)
  apply (simp)
  apply (subgoal-tac c * abs (f xa + g xa) <= c * (abs (f xa) + abs (g xa)))
  apply (erule order-trans)
  apply (simp add: ring-distribs)
  apply (rule mult-left-mono)
  apply (simp add: order-less-le)
  apply (simp add: order-less-le)
  apply (rule mult-left-mono)
  apply (rule abs-triangle-ineq)
```

apply (*simp add*: *order-less-le*)
apply (*rule mult-nonneg-nonneg*)
apply (*rule add-nonneg-nonneg*)
apply (*erule order-less-imp-le*)+
apply *simp*
apply (*rule ext*)
apply (*auto simp add*: *if-splits linorder-not-le*)
done

lemma *bigo-plus-subset2* [*intro*]: $A <= O(f) ==> B <= O(f) ==> A + B <= O(f)$
apply (*subgoal-tac* $A + B <= O(f) + O(f)$)
apply (*erule order-trans*)
apply *simp*
apply (*auto del*: *subsetI simp del*: *bigo-plus-idemp*)
done

lemma *bigo-plus-eq*: $ALL\ x.\ 0 <= f\ x ==> ALL\ x.\ 0 <= g\ x ==>$
$O(f + g) = O(f) + O(g)$
apply (*rule equalityI*)
apply (*rule bigo-plus-subset*)
apply (*simp add*: *bigo-alt-def set-plus func-plus*)
apply *clarify*
apply (*rule-tac* $x = max\ c\ ca$ **in** *exI*)
apply (*rule conjI*)
apply (*subgoal-tac* $c <= max\ c\ ca$)
apply (*erule order-less-le-trans*)
apply *assumption*
apply (*rule le-maxI1*)
apply *clarify*
apply (*drule-tac* $x = xa$ **in** *spec*)+
apply (*subgoal-tac* $0 <= f\ xa + g\ xa$)
apply (*simp add*: *ring-distribs*)
apply (*subgoal-tac* $abs(a\ xa + b\ xa) <= abs(a\ xa) + abs(b\ xa)$)
apply (*subgoal-tac* $abs(a\ xa) + abs(b\ xa) <=$
$max\ c\ ca * f\ xa + max\ c\ ca * g\ xa$)
apply (*force*)
apply (*rule add-mono*)
apply (*subgoal-tac* $c * f\ xa <= max\ c\ ca * f\ xa$)
apply (*force*)
apply (*rule mult-right-mono*)
apply (*rule le-maxI1*)
apply *assumption*
apply (*subgoal-tac* $ca * g\ xa <= max\ c\ ca * g\ xa$)
apply (*force*)
apply (*rule mult-right-mono*)
apply (*rule le-maxI2*)
apply *assumption*
apply (*rule abs-triangle-ineq*)

apply (*rule add-nonneg-nonneg*)
apply *assumption+*
done

lemma *bigo-bounded-alt*: *ALL x. 0 <= f x ==> ALL x. f x <= c * g x ==>*
f : O(g)
apply (*auto simp add*: *bigo-def*)
apply (*rule-tac x = abs c* **in** *exI*)
apply *auto*
apply (*drule-tac x = x* **in** *spec*)+
apply (*simp add*: *abs-mult* [*symmetric*])
done

lemma *bigo-bounded*: *ALL x. 0 <= f x ==> ALL x. f x <= g x ==>*
f : O(g)
apply (*erule bigo-bounded-alt* [*of f 1 g*])
apply *simp*
done

lemma *bigo-bounded2*: *ALL x. lb x <= f x ==> ALL x. f x <= lb x + g x ==>*
f : lb +o O(g)
apply (*rule set-minus-imp-plus*)
apply (*rule bigo-bounded*)
apply (*auto simp add*: *diff-minus func-minus func-plus*)
apply (*drule-tac x = x* **in** *spec*)+
apply *force*
apply (*drule-tac x = x* **in** *spec*)+
apply *force*
done

lemma *bigo-abs*: (*%x. abs*(*f x*)) *=o O*(*f*)
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac x = 1* **in** *exI*)
apply *auto*
done

lemma *bigo-abs2*: *f =o O*(*%x. abs*(*f x*))
apply (*unfold bigo-def*)
apply *auto*
apply (*rule-tac x = 1* **in** *exI*)
apply *auto*
done

lemma *bigo-abs3*: *O*(*f*) = *O*(*%x. abs*(*f x*))
apply (*rule equalityI*)
apply (*rule bigo-elt-subset*)
apply (*rule bigo-abs2*)
apply (*rule bigo-elt-subset*)

```
  apply (rule bigo-abs)
  done

lemma bigo-abs4: f =o g +o O(h) ==>
    (%x. abs (f x)) =o (%x. abs (g x)) +o O(h)
  apply (drule set-plus-imp-minus)
  apply (rule set-minus-imp-plus)
  apply (subst func-diff)
proof -
  assume a: f - g : O(h)
  have (%x. abs (f x) - abs (g x)) =o O(%x. abs(abs (f x) - abs (g x)))
    by (rule bigo-abs2)
  also have ... <= O(%x. abs (f x - g x))
    apply (rule bigo-elt-subset)
    apply (rule bigo-bounded)
    apply force
    apply (rule allI)
    apply (rule abs-triangle-ineq3)
    done
  also have ... <= O(f - g)
    apply (rule bigo-elt-subset)
    apply (subst func-diff)
    apply (rule bigo-abs)
    done
  also from a have ... <= O(h)
    by (rule bigo-elt-subset)
  finally show (%x. abs (f x) - abs (g x)) : O(h).
qed

lemma bigo-abs5: f =o O(g) ==> (%x. abs(f x)) =o O(g)
  by (unfold bigo-def, auto)

lemma bigo-elt-subset2 [intro]: f : g +o O(h) ==> O(f) <= O(g) + O(h)
proof -
  assume f : g +o O(h)
  also have ... <= O(g) + O(h)
    by (auto del: subsetI)
  also have ... = O(%x. abs(g x)) + O(%x. abs(h x))
    apply (subst bigo-abs3 [symmetric])+
    apply (rule refl)
    done
  also have ... = O((%x. abs(g x)) + (%x. abs(h x)))
    by (rule bigo-plus-eq [symmetric], auto)
  finally have f : ....
  then have O(f) <= ...
    by (elim bigo-elt-subset)
  also have ... = O(%x. abs(g x)) + O(%x. abs(h x))
    by (rule bigo-plus-eq, auto)
  finally show ?thesis
```

by (*simp add*: *bigo-abs3* [*symmetric*])
qed

lemma *bigo-mult* [*intro*]: $O(f)*O(g) <= O(f * g)$
apply (*rule subsetI*)
apply (*subst bigo-def*)
apply (*auto simp add*: *bigo-alt-def set-times func-times*)
apply (*rule-tac* $x = c * ca$ **in** *exI*)
apply(*rule allI*)
apply(*erule-tac* $x = x$ **in** *allE*)+
apply(*subgoal-tac* $c * ca * abs(f\,x * g\,x) =$
$(c * abs(f\,x)) * (ca * abs(g\,x)))$
apply(*erule ssubst*)
apply (*subst abs-mult*)
apply (*rule mult-mono*)
apply *assumption*+
apply (*rule mult-nonneg-nonneg*)
apply *auto*
apply (*simp add*: *mult-ac abs-mult*)
done

lemma *bigo-mult2* [*intro*]: $f *o\ O(g) <= O(f * g)$
apply (*auto simp add*: *bigo-def elt-set-times-def func-times abs-mult*)
apply (*rule-tac* $x = c$ **in** *exI*)
apply *auto*
apply (*drule-tac* $x = x$ **in** *spec*)
apply (*subgoal-tac* $abs(f\,x) * abs(b\,x) <= abs(f\,x) * (c * abs(g\,x)))$
apply (*force simp add*: *mult-ac*)
apply (*rule mult-left-mono*, *assumption*)
apply (*rule abs-ge-zero*)
done

lemma *bigo-mult3*: $f : O(h) ==> g : O(j) ==> f * g : O(h * j)$
apply (*rule subsetD*)
apply (*rule bigo-mult*)
apply (*erule set-times-intro*, *assumption*)
done

lemma *bigo-mult4* [*intro*]:$f : k +o\ O(h) ==> g * f : (g * k) +o\ O(g * h)$
apply (*drule set-plus-imp-minus*)
apply (*rule set-minus-imp-plus*)
apply (*drule bigo-mult3* [**where** $g = g$ **and** $j = g$])
apply (*auto simp add*: *ring-simps*)
done

lemma *bigo-mult5*: *ALL* $x.\ f\,x$ ~= *0* ==>
$O(f * g) <= (f::'a => ('b::ordered\text{-}field)) *o\ O(g)$
proof –
assume *ALL* $x.\ f\,x$ ~= *0*

```
  show O(f * g) <= f *o O(g)
  proof
    fix h
    assume h : O(f * g)
    then have (%x. 1 / (f x)) * h : (%x. 1 / f x) *o O(f * g)
      by auto
    also have ... <= O((%x. 1 / f x) * (f * g))
      by (rule bigo-mult2)
    also have (%x. 1 / f x) * (f * g) = g
      apply (simp add: func-times)
      apply (rule ext)
      apply (simp add: prems nonzero-divide-eq-eq mult-ac)
      done
    finally have (%x. (1::'b) / f x) * h : O(g).
    then have f * ((%x. (1::'b) / f x) * h) : f *o O(g)
      by auto
    also have f * ((%x. (1::'b) / f x) * h) = h
      apply (simp add: func-times)
      apply (rule ext)
      apply (simp add: prems nonzero-divide-eq-eq mult-ac)
      done
    finally show h : f *o O(g).
  qed
qed

lemma bigo-mult6: ALL x. f x ~= 0 ==>
    O(f * g) = (f::'a => ('b::ordered-field)) *o O(g)
  apply (rule equalityI)
  apply (erule bigo-mult5)
  apply (rule bigo-mult2)
  done

lemma bigo-mult7: ALL x. f x ~= 0 ==>
    O(f * g) <= O(f::'a => ('b::ordered-field)) * O(g)
  apply (subst bigo-mult6)
  apply assumption
  apply (rule set-times-mono3)
  apply (rule bigo-refl)
  done

lemma bigo-mult8: ALL x. f x ~= 0 ==>
    O(f * g) = O(f::'a => ('b::ordered-field)) * O(g)
  apply (rule equalityI)
  apply (erule bigo-mult7)
  apply (rule bigo-mult)
  done

lemma bigo-minus [intro]: f : O(g) ==> - f : O(g)
  by (auto simp add: bigo-def func-minus)
```

```
lemma bigo-minus2: f : g +o O(h) ==> −f : −g +o O(h)
  apply (rule set-minus-imp-plus)
  apply (drule set-plus-imp-minus)
  apply (drule bigo-minus)
  apply (simp add: diff-minus)
  done

lemma bigo-minus3: O(−f) = O(f)
  by (auto simp add: bigo-def func-minus abs-minus-cancel)

lemma bigo-plus-absorb-lemma1: f : O(g) ==> f +o O(g) <= O(g)
proof −
  assume a: f : O(g)
  show f +o O(g) <= O(g)
  proof −
    have f : O(f) by auto
    then have f +o O(g) <= O(f) + O(g)
      by (auto del: subsetI)
    also have ... <= O(g) + O(g)
    proof −
      from a have O(f) <= O(g) by (auto del: subsetI)
      thus ?thesis by (auto del: subsetI)
    qed
    also have ... <= O(g) by (simp add: bigo-plus-idemp)
    finally show ?thesis .
  qed
qed

lemma bigo-plus-absorb-lemma2: f : O(g) ==> O(g) <= f +o O(g)
proof −
  assume a: f : O(g)
  show O(g) <= f +o O(g)
  proof −
    from a have −f : O(g) by auto
    then have −f +o O(g) <= O(g) by (elim bigo-plus-absorb-lemma1)
    then have f +o (−f +o O(g)) <= f +o O(g) by auto
    also have f +o (−f +o O(g)) = O(g)
      by (simp add: set-plus-rearranges)
    finally show ?thesis .
  qed
qed

lemma bigo-plus-absorb [simp]: f : O(g) ==> f +o O(g) = O(g)
  apply (rule equalityI)
  apply (erule bigo-plus-absorb-lemma1)
  apply (erule bigo-plus-absorb-lemma2)
  done
```

lemma *bigo-plus-absorb2* [*intro*]: f : $O(g)$ ==> A <= $O(g)$ ==> f +o A <= $O(g)$
  apply (*subgoal-tac* f +o A <= f +o $O(g)$)
  apply *force*+
  done

lemma *bigo-add-commute-imp*: f : g +o $O(h)$ ==> g : f +o $O(h)$
  apply (*subst set-minus-plus* [*symmetric*])
  apply (*subgoal-tac* $g - f = - (f - g)$)
  apply (*erule ssubst*)
  apply (*rule bigo-minus*)
  apply (*subst set-minus-plus*)
  apply *assumption*
  apply (*simp add*: *diff-minus add-ac*)
  done

lemma *bigo-add-commute*: (f : g +o $O(h)$) = (g : f +o $O(h)$)
  apply (*rule iffI*)
  apply (*erule bigo-add-commute-imp*)+
  done

lemma *bigo-const1*: (%x. c) : O(%x. *1*)
  by (*auto simp add*: *bigo-def mult-ac*)

lemma *bigo-const2* [*intro*]: O(%x. c) <= O(%x. *1*)
  apply (*rule bigo-elt-subset*)
  apply (*rule bigo-const1*)
  done

lemma *bigo-const3*: (c::$'a$::*ordered-field*) ~= *0* ==> (%x. *1*) : O(%x. c)
  apply (*simp add*: *bigo-def*)
  apply (*rule-tac* x = *abs*(*inverse* c) **in** *exI*)
  apply (*simp add*: *abs-mult* [*symmetric*])
  done

lemma *bigo-const4*: (c::$'a$::*ordered-field*) ~= *0* ==> O(%x. *1*) <= O(%x. c)
  by (*rule bigo-elt-subset*, *rule bigo-const3*, *assumption*)

lemma *bigo-const* [*simp*]: (c::$'a$::*ordered-field*) ~= *0* ==>
    O(%x. c) = O(%x. *1*)
  by (*rule equalityI*, *rule bigo-const2*, *rule bigo-const4*, *assumption*)

lemma *bigo-const-mult1*: (%x. $c * f\ x$) : $O(f)$
  apply (*simp add*: *bigo-def*)
  apply (*rule-tac* x = *abs*(c) **in** *exI*)
  apply (*auto simp add*: *abs-mult* [*symmetric*])
  done

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

```
  by (rule bigo-elt-subset, rule bigo-const-mult1)

lemma bigo-const-mult3: (c::'a::ordered-field) ~= 0 ==> f : O(%x. c * f x)
  apply (simp add: bigo-def)
  apply (rule-tac x = abs(inverse c) in exI)
  apply (simp add: abs-mult [symmetric] mult-assoc [symmetric])
  done

lemma bigo-const-mult4: (c::'a::ordered-field) ~= 0 ==>
    O(f) <= O(%x. c * f x)
  by (rule bigo-elt-subset, rule bigo-const-mult3, assumption)

lemma bigo-const-mult [simp]: (c::'a::ordered-field) ~= 0 ==>
    O(%x. c * f x) = O(f)
  by (rule equalityI, rule bigo-const-mult2, erule bigo-const-mult4)

lemma bigo-const-mult5 [simp]: (c::'a::ordered-field) ~= 0 ==>
    (%x. c) *o O(f) = O(f)
  apply (auto del: subsetI)
  apply (rule order-trans)
  apply (rule bigo-mult2)
  apply (simp add: func-times)
  apply (auto intro!: subsetI simp add: bigo-def elt-set-times-def func-times)
  apply (rule-tac x = %y. inverse c * x y in exI)
  apply (simp add: mult-assoc [symmetric] abs-mult)
  apply (rule-tac x = abs (inverse c) * ca in exI)
  apply (rule allI)
  apply (subst mult-assoc)
  apply (rule mult-left-mono)
  apply (erule spec)
  apply force
  done

lemma bigo-const-mult6 [intro]: (%x. c) *o O(f) <= O(f)
  apply (auto intro!: subsetI
    simp add: bigo-def elt-set-times-def func-times)
  apply (rule-tac x = ca * (abs c) in exI)
  apply (rule allI)
  apply (subgoal-tac ca * abs(c) * abs(f x) = abs(c) * (ca * abs(f x)))
  apply (erule ssubst)
  apply (subst abs-mult)
  apply (rule mult-left-mono)
  apply (erule spec)
  apply simp
  apply(simp add: mult-ac)
  done

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

assume f =o O(g)
then have (%x. c) * f =o (%x. c) *o O(g)
by *auto*
also have (%x. c) * f = (%x. c * f x)
by (*simp add*: *func-times*)
also have (%x. c) *o O(g) <= O(g)
by (*auto del*: *subsetI*)
finally show *?thesis* **.**
qed

lemma *bigo-compose1*: f =o O(g) ==> (%x. f(k x)) =o O(%x. g(k x))
by (*unfold bigo-def*, *auto*)

lemma *bigo-compose2*: f =o g +o O(h) ==> (%x. f(k x)) =o (%x. g(k x)) +o O(%x. h(k x))
apply (*simp only*: *set-minus-plus* [*symmetric*] *diff-minus func-minus func-plus*)
apply (*erule bigo-compose1*)
done

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)
apply (*auto simp add*: *bigo-def*)
apply (*rule-tac x = abs c* **in** *exI*)
apply (*subst abs-of-nonneg*) **back back**
apply (*rule setsum-nonneg*)
apply *force*
apply (*subst setsum-right-distrib*)
apply (*rule allI*)
apply (*rule order-trans*)
apply (*rule setsum-abs*)
apply (*rule setsum-mono*)
apply (*rule order-trans*)
apply (*drule spec*)+
apply (*drule bspec*)+
apply *assumption*+
apply (*drule bspec*)
apply *assumption*+
apply (*rule mult-right-mono*)
apply (*rule abs-ge-self*)
apply *force*
done

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)

```
  apply (rule bigo-setsum-main)
  apply force
  apply clarsimp
  apply (rule-tac x = c in exI)
  apply force
  done

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)
  by (rule bigo-setsum1, auto)

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)))
  apply (rule bigo-setsum1)
  apply (rule allI)+
  apply (rule abs-ge-zero)
  apply (unfold bigo-def)
  apply auto
  apply (rule-tac x = c in exI)
  apply (rule allI)+
  apply (subst abs-mult)+
  apply (subst mult-left-commute)
  apply (rule mult-left-mono)
  apply (erule spec)
  apply (rule abs-ge-zero)
  done

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)))
  apply (rule set-minus-imp-plus)
  apply (subst func-diff)
  apply (subst setsum-subtractf [symmetric])
  apply (subst right-diff-distrib [symmetric])
  apply (rule bigo-setsum3)
  apply (subst func-diff [symmetric])
  apply (erule set-plus-imp-minus)
  done

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))
  apply (subgoal-tac (%x. SUM y : A x. (l x y) * h(k x y)) =
      (%x. SUM y : A x. abs((l x y) * h(k x y))))
  apply (erule ssubst)
```

apply (erule bigo-setsum3)
apply (rule ext)
apply (rule setsum-cong2)
apply (subst abs-of-nonneg)
apply (rule mult-nonneg-nonneg)
apply auto
done

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))
apply (rule set-minus-imp-plus)
apply (subst func-diff)
apply (subst setsum-subtractf [symmetric])
apply (subst right-diff-distrib [symmetric])
apply (rule bigo-setsum5)
apply (subst func-diff [symmetric])
apply (drule set-plus-imp-minus)
apply auto
done

5.3 Misc useful stuff

lemma bigo-useful-intro: A <= O(f) ==> B <= O(f) ==>
A + B <= O(f)
apply (subst bigo-plus-idemp [symmetric])
apply (rule set-plus-mono2)
apply assumption+
done

lemma bigo-useful-add: f =o O(h) ==> g =o O(h) ==> f + g =o O(h)
apply (subst bigo-plus-idemp [symmetric])
apply (rule set-plus-intro)
apply assumption+
done

lemma bigo-useful-const-mult: (c::'a::ordered-field) ~= 0 ==>
(%x. c) * f =o O(h) ==> f =o O(h)
apply (rule subsetD)
apply (subgoal-tac (%x. 1 / c) *o O(h) <= O(h))
apply assumption
apply (rule bigo-const-mult6)
apply (subgoal-tac f = (%x. 1 / c) * ((%x. c) * f))
apply (erule ssubst)
apply (erule set-times-intro2)
apply (simp add: func-times)
done

```
lemma bigo-fix: (%x. f ((x::nat) + 1)) =o O(%x. h(x + 1)) ==> f 0 = 0 ==>
   f =o O(h)
  apply (simp add: bigo-alt-def)
  apply auto
  apply (rule-tac x = c in exI)
  apply auto
  apply (case-tac x = 0)
  apply simp
  apply (rule mult-nonneg-nonneg)
  apply force
  apply force
  apply (subgoal-tac x = Suc (x - 1))
  apply (erule ssubst) back
  apply (erule spec)
  apply simp
  done

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)
  apply (rule set-minus-imp-plus)
  apply (rule bigo-fix)
  apply (subst func-diff)
  apply (subst func-diff [symmetric])
  apply (rule set-plus-imp-minus)
  apply simp
  apply (simp add: func-diff)
  done
```

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)
  apply (unfold bigo-def)
  apply clarsimp
  apply (rule-tac x = c in exI)
  apply (rule allI)
  apply (rule order-trans)
  apply (erule spec)+
  done

lemma bigo-lesseq2: f =o O(h) ==> ALL x. abs (g x) <= f x ==>
     g =o O(h)
```

apply (*erule bigo-lesseq1*)
apply (*rule allI*)
apply (*drule-tac x = x* **in** *spec*)
apply (*rule order-trans*)
apply *assumption*
apply (*rule abs-ge-self*)
done

lemma *bigo-lesseq3*: *f =o O(h) ==> ALL x. 0 <= g x ==> ALL x. g x <= f x ==>*
g =o O(h)
apply (*erule bigo-lesseq2*)
apply (*rule allI*)
apply (*subst abs-of-nonneg*)
apply (*erule spec*)+
done

lemma *bigo-lesseq4*: *f =o O(h) ==>*
ALL x. 0 <= g x ==> ALL x. g x <= abs (f x) ==>
g =o O(h)
apply (*erule bigo-lesseq1*)
apply (*rule allI*)
apply (*subst abs-of-nonneg*)
apply (*erule spec*)+
done

lemma *bigo-lesso1*: *ALL x. f x <= g x ==> f <o g =o O(h)*
apply (*unfold lesso-def*)
apply (*subgoal-tac* (*%x. max* (*f x − g x*) *0*) = *0*)
apply (*erule ssubst*)
apply (*rule bigo-zero*)
apply (*unfold func-zero*)
apply (*rule ext*)
apply (*simp split*: *split-max*)
done

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)
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq4*)
apply (*erule set-plus-imp-minus*)
apply (*rule allI*)
apply (*rule le-maxI2*)
apply (*rule allI*)
apply (*subst func-diff*)
apply (*case-tac 0 <= k x − g x*)
apply *simp*
apply (*subst abs-of-nonneg*)

```
  apply (drule-tac x = x in spec) back
  apply (simp add: compare-rls)
  apply (subst diff-minus)+
  apply (rule add-right-mono)
  apply (erule spec)
  apply (rule order-trans)
  prefer 2
  apply (rule abs-ge-zero)
  apply (simp add: compare-rls)
  done

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)
  apply (unfold lesso-def)
  apply (rule bigo-lesseq4)
  apply (erule set-plus-imp-minus)
  apply (rule allI)
  apply (rule le-maxI2)
  apply (rule allI)
  apply (subst func-diff)
  apply (case-tac 0 <= f x - k x)
  apply simp
  apply (subst abs-of-nonneg)
  apply (drule-tac x = x in spec) back
  apply (simp add: compare-rls)
  apply (subst diff-minus)+
  apply (rule add-left-mono)
  apply (rule le-imp-neg-le)
  apply (erule spec)
  apply (rule order-trans)
  prefer 2
  apply (rule abs-ge-zero)
  apply (simp add: compare-rls)
  done

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)
  apply (unfold lesso-def)
  apply (drule set-plus-imp-minus)
  apply (drule bigo-abs5) back
  apply (simp add: func-diff)
  apply (drule bigo-useful-add)
  apply assumption
  apply (erule bigo-lesseq2) back
  apply (rule allI)
  apply (auto simp add: func-plus func-diff compare-rls
    split: split-max abs-split)
  done
```

```
lemma bigo-lesso5: f <o g =o O(h) ==>
   EX C. ALL x. f x <= g x + C * abs(h x)
  apply (simp only: lesso-def bigo-alt-def)
  apply clarsimp
  apply (rule-tac x = c in exI)
  apply (rule allI)
  apply (drule-tac x = x in spec)
  apply (subgoal-tac abs(max (f x - g x) 0) = max (f x - g x) 0)
  apply (clarsimp simp add: compare-rls add-ac)
  apply (rule abs-of-nonneg)
  apply (rule le-maxI2)
  done

lemma lesso-add: f <o g =o O(h) ==>
     k <o l =o O(h) ==> (f + k) <o (g + l) =o O(h)
  apply (unfold lesso-def)
  apply (rule bigo-lesseq3)
  apply (erule bigo-useful-add)
  apply assumption
  apply (force split: split-max)
  apply (auto split: split-max simp add: func-plus)
  done

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
by (cases n) simp-all

lemma binomial-0-Suc [simp]: (0 choose Suc k) = 0
by simp

lemma binomial-Suc-Suc [simp]:
```

```
  (Suc n choose Suc k) = (n choose k) + (n choose Suc k)
by simp

lemma binomial-eq-0: !!k. n < k ==> (n choose k) = 0
by (induct n) auto

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

lemma binomial-n-n [simp]: (n choose n) = 1
by (induct n) (simp-all add: binomial-eq-0)

lemma binomial-Suc-n [simp]: (Suc n choose n) = Suc n
by (induct n) simp-all

lemma binomial-1 [simp]: (n choose Suc 0) = n
by (induct n) simp-all

lemma zero-less-binomial: k ≤ n ==> (n choose k) > 0
by (induct n k rule: diff-induct) simp-all

lemma binomial-eq-0-iff: (n choose k = 0) = (n<k)
apply (safe intro!: binomial-eq-0)
apply (erule contrapos-pp)
apply (simp add: zero-less-binomial)
done

lemma zero-less-binomial-iff: (n choose k > 0) = (k≤n)
by(simp add: linorder-not-less binomial-eq-0-iff neq0-conv[symmetric]
      del:neq0-conv)


lemma Suc-times-binomial-eq:
  !!k. k ≤ n ==> Suc n * (n choose k) = (Suc n choose Suc k) * Suc k
apply (induct n)
apply (simp add: binomial-0)
apply (case-tac k)
apply (auto simp add: add-mult-distrib add-mult-distrib2 le-Suc-eq
    binomial-eq-0)
done
```

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

```
lemma binomial-Suc-Suc-eq-times:
    k ≤ n ==> (Suc n choose Suc k) = (Suc n * (n choose k)) div Suc k
  by (simp add: Suc-times-binomial-eq div-mult-self-is-m zero-less-Suc
    del: mult-Suc mult-Suc-right)
```

Another version, with -1 instead of Suc.

```
lemma times-binomial-minus1-eq:
```

```
  [|k ≤ n;  0<k|] ==> (n choose k) * k = n * ((n − 1) choose (k − 1))
 apply (cut-tac n = n − 1 and k = k − 1 in Suc-times-binomial-eq)
 apply (simp split add: nat-diff-split, auto)
 done
```

6.1 Theorems about *choose*

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

```
lemma card-s-0-eq-empty:
  finite A ==> card {B. B ⊆ A & card B = 0} = 1
 apply (simp cong add: conj-cong add: finite-subset [THEN card-0-eq])
 apply (simp cong add: rev-conj-cong)
 done

lemma choose-deconstruct: finite M ==> x ∉ M
 ==> {s. s <= insert x M & card(s) = Suc k}
     = {s. s <= M & card(s) = Suc k} Un
       {s. EX t. t <= M & card(t) = k & s = insert x t}
 apply safe
  apply (auto intro: finite-subset [THEN card-insert-disjoint])
 apply (drule-tac x = xa − {x} in spec)
 apply (subgoal-tac x ∉ xa, auto)
 apply (erule rev-mp, subst card-Diff-singleton)
 apply (auto intro: finite-subset)
 done
```

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

```
lemma constr-bij:
  [|finite A; x ∉ A|] ==>
   card {B. EX C. C <= A & card(C) = k & B = insert x C} =
   card {B. B <= A & card(B) = k}
 apply (rule-tac f = %s. s − {x} and g = insert x in card-bij-eq)
      apply (auto elim!: equalityE simp add: inj-on-def)
   apply (subst Diff-insert0, auto)
```

finiteness of the two sets

```
  apply (rule-tac [2] B = Pow (A) in finite-subset)
  apply (rule-tac B = Pow (insert x A) in finite-subset)
  apply fast+
 done
```

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)
 apply (induct k)
  apply (simp add: card-s-0-eq-empty, atomize)
```

```
  apply (rotate-tac -1, erule finite-induct)
   apply (simp-all (no-asm-simp) cong add: conj-cong
     add: card-s-0-eq-empty choose-deconstruct)
  apply (subst card-Un-disjoint)
     prefer 4 apply (force simp add: constr-bij)
    prefer 3 apply force
   prefer 2 apply (blast intro: finite-Pow-iff [THEN iffD2]
     finite-subset [of - Pow (insert x F), standard])
  apply (blast intro: finite-Pow-iff [THEN iffD2, THEN [2] finite-subset])
  done

theorem n-subsets:
    finite A ==> card {B. B <= A & card B = k} = (card A choose k)
  by (simp add: n-sub-lemma)
```

The binomial theorem (courtesy of Tobias Nipkow):

```
theorem binomial: (a+b::nat)^n = (∑ k=0..n. (n choose k) * a^k * b^(n-k))
proof (induct n)
  case 0 thus ?case by simp
next
  case (Suc n)
  have decomp: {0..n+1} = {0} ∪ {n+1} ∪ {1..n}
    by (auto simp add:atLeastAtMost-def atLeast-def atMost-def)
  have decomp2: {0..n} = {0} ∪ {1..n}
    by (auto simp add:atLeastAtMost-def atLeast-def atMost-def)
  have (a+b::nat)^(n+1) = (a+b) * (∑ k=0..n. (n choose k) * a^k * b^(n-k))
    using Suc by simp
  also have ... =  a*(∑ k=0..n. (n choose k) * a^k * b^(n-k)) +
                 b*(∑ k=0..n. (n choose k) * a^k * b^(n-k))
    by (rule nat-distrib)
  also have ... = (∑ k=0..n. (n choose k) * a^(k+1) * b^(n-k)) +
                 (∑ k=0..n. (n choose k) * a^k * b^(n-k+1))
    by (simp add: setsum-right-distrib mult-ac)
  also have ... = (∑ k=0..n. (n choose k) * a^k * b^(n+1-k)) +
                 (∑ k=1..n+1. (n choose (k - 1)) * a^k * b^(n+1-k))
    by (simp add:setsum-shift-bounds-cl-Suc-ivl Suc-diff-le
             del:setsum-cl-ivl-Suc)
  also have ... = a^(n+1) + b^(n+1) +
                 (∑ k=1..n. (n choose (k - 1)) * a^k * b^(n+1-k)) +
                 (∑ k=1..n. (n choose k) * a^k * b^(n+1-k))
    by (simp add: decomp2)
  also have
     ... = a^(n+1) + b^(n+1) + (∑ k=1..n. (n+1 choose k) * a^k * b^(n+1-k))
    by (simp add: nat-distrib setsum-addf binomial.simps)
  also have ... = (∑ k=0..n+1. (n+1 choose k) * a^k * b^(n+1-k))
    using decomp by simp
  finally show ?case by simp
qed
```

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*
apply (*rule boolean.intro*)
apply (*rule disj-assoc*)
apply (*rule conj-assoc*)
apply (*rule disj-commute*)
apply (*rule conj-commute*)
apply (*rule disj-conj-distrib*)
apply (*rule conj-disj-distrib*)
apply (*rule disj-zero-right*)
apply (*rule conj-one-right*)
apply (*rule disj-cancel-right*)
apply (*rule conj-cancel-right*)
done

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 $-$
have $(a \sqcap x) \sqcup (x \sqcap y) = (a \sqcap y) \sqcup (x \sqcap y)$ **using** *1 3* **by** *simp*
hence $(x \sqcap a) \sqcup (x \sqcap y) = (y \sqcap a) \sqcup (y \sqcap x)$ **using** *conj-commute* **by** *simp*
hence $x \sqcap (a \sqcup y) = y \sqcap (a \sqcup x)$ **using** *conj-disj-distrib* **by** *simp*
hence $x \sqcap \mathbf{1} = y \sqcap \mathbf{1}$ **using** *2 4* **by** *simp*
thus $x = y$ **using** *conj-one-right* **by** *simp*
qed

lemma *compl-unique*: $[\![x \sqcap y = \mathbf{0};\ x \sqcup y = \mathbf{1}]\!] \Longrightarrow \sim x = y$
by (*rule complement-unique* [*OF conj-cancel-right disj-cancel-right*])

lemma *double-compl* [*simp*]: $\sim (\sim x) = x$
proof (*rule compl-unique*)
from *conj-cancel-right* **show** $\sim x \sqcap x = \mathbf{0}$ **by** (*simp only*: *conj-commute*)
from *disj-cancel-right* **show** $\sim x \sqcup x = \mathbf{1}$ **by** (*simp only*: *disj-commute*)
qed

lemma *compl-eq-compl-iff* [*simp*]: $(\sim x = \sim y) = (x = y)$
by (*rule inj-eq* [*OF inj-on-inverseI*], *rule double-compl*)

7.2 Conjunction

lemma *conj-absorb* [*simp*]: $x \sqcap x = x$
proof $-$
have $x \sqcap x = (x \sqcap x) \sqcup \mathbf{0}$ **using** *disj-zero-right* **by** *simp*
also have ... $= (x \sqcap x) \sqcup (x \sqcap \sim x)$ **using** *conj-cancel-right* **by** *simp*
also have ... $= x \sqcap (x \sqcup \sim x)$ **using** *conj-disj-distrib* **by** (*simp only*:)
also have ... $= x \sqcap \mathbf{1}$ **using** *disj-cancel-right* **by** *simp*
also have ... $= x$ **using** *conj-one-right* **by** *simp*
finally show *?thesis* **.**
qed

lemma *conj-zero-right* [*simp*]: $x \sqcap \mathbf{0} = \mathbf{0}$
proof $-$
have $x \sqcap \mathbf{0} = x \sqcap (x \sqcap \sim x)$ **using** *conj-cancel-right* **by** *simp*
also have ... $= (x \sqcap x) \sqcap \sim x$ **using** *conj-assoc* **by** (*simp only*:)
also have ... $= x \sqcap \sim x$ **using** *conj-absorb* **by** *simp*
also have ... $= \mathbf{0}$ **using** *conj-cancel-right* **by** *simp*
finally show *?thesis* **.**
qed

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

by (*rule compl-unique* [*OF conj-zero-right disj-zero-right*])

lemma *conj-zero-left* [*simp*]: $\mathbf{0} \sqcap x = \mathbf{0}$
by (*subst conj-commute*) (*rule conj-zero-right*)

lemma *conj-one-left* [*simp*]: $\mathbf{1} \sqcap x = x$
by (*subst conj-commute*) (*rule conj-one-right*)

lemma *conj-cancel-left* [*simp*]: $\sim x \sqcap x = \mathbf{0}$
by (*subst conj-commute*) (*rule conj-cancel-right*)

lemma *conj-left-absorb* [*simp*]: $x \sqcap (x \sqcap y) = x \sqcap y$
by (*simp only*: *conj-assoc* [*symmetric*] *conj-absorb*)

lemma *conj-disj-distrib2*:
$(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
by (*simp only*: *conj-commute conj-disj-distrib*)

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

7.3 Disjunction

lemma *disj-absorb* [*simp*]: $x \sqcup x = x$
by (*rule boolean.conj-absorb* [*OF dual*])

lemma *disj-one-right* [*simp*]: $x \sqcup \mathbf{1} = \mathbf{1}$
by (*rule boolean.conj-zero-right* [*OF dual*])

lemma *compl-zero* [*simp*]: $\sim \mathbf{0} = \mathbf{1}$
by (*rule boolean.compl-one* [*OF dual*])

lemma *disj-zero-left* [*simp*]: $\mathbf{0} \sqcup x = x$
by (*rule boolean.conj-one-left* [*OF dual*])

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$
by (*rule boolean.conj-zero-left* [*OF dual*])

lemma *disj-cancel-left* [*simp*]: $\sim x \sqcup x = \mathbf{1}$
by (*rule boolean.conj-cancel-left* [*OF dual*])

lemma *disj-left-absorb* [*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
by (*rule boolean.conj-left-absorb* [*OF dual*])

lemma *disj-conj-distrib2*:
$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
by (*rule boolean.conj-disj-distrib2* [*OF dual*])

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 (*rule compl-unique*)
  have $(x \sqcap y) \sqcap (\sim x \sqcup \sim y) = ((x \sqcap y) \sqcap \sim x) \sqcup ((x \sqcap y) \sqcap \sim y)$
    by (*rule conj-disj-distrib*)
  also have ... $= (y \sqcap (x \sqcap \sim x)) \sqcup (x \sqcap (y \sqcap \sim y))$
    by (*simp only*: *conj-ac*)
  finally show $(x \sqcap y) \sqcap (\sim x \sqcup \sim y) = \mathbf{0}$
    by (*simp only*: *conj-cancel-right conj-zero-right disj-zero-right*)
next
  have $(x \sqcap y) \sqcup (\sim x \sqcup \sim y) = (x \sqcup (\sim x \sqcup \sim y)) \sqcap (y \sqcup (\sim x \sqcup \sim y))$
    by (*rule disj-conj-distrib2*)
  also have ... $= (\sim y \sqcup (x \sqcup \sim x)) \sqcap (\sim x \sqcup (y \sqcup \sim y))$
    by (*simp only*: *disj-ac*)
  finally show $(x \sqcap y) \sqcup (\sim x \sqcup \sim y) = \mathbf{1}$
    by (*simp only*: *disj-cancel-right disj-one-right conj-one-right*)
qed

lemma *de-Morgan-disj* [*simp*]: $\sim (x \sqcup y) = \sim x \sqcap \sim y$
by (*rule boolean.de-Morgan-conj* [*OF dual*])

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*:
  $x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$
by (*simp only*: *xor-def conj-disj-distribs*
        disj-ac conj-ac conj-cancel-right disj-zero-left)

lemma *xor-commute*: $x \oplus y = y \oplus x$
by (*simp only*: *xor-def conj-commute disj-commute*)

lemma *xor-assoc*: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
proof −
  let *?t* $= (x \sqcap y \sqcap z) \sqcup (x \sqcap \sim y \sqcap \sim z) \sqcup$
        $(\sim x \sqcap y \sqcap \sim z) \sqcup (\sim x \sqcap \sim y \sqcap z)$
  have *?t* $\sqcup (z \sqcap x \sqcap \sim x) \sqcup (z \sqcap y \sqcap \sim y) =$
      ?t $\sqcup (x \sqcap y \sqcap \sim y) \sqcup (x \sqcap z \sqcap \sim z)$
    by (*simp only*: *conj-cancel-right conj-zero-right*)
  thus $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
    apply (*simp only*: *xor-def de-Morgan-disj de-Morgan-conj double-compl*)

apply (*simp only*: *conj-disj-distribs conj-ac disj-ac*)
done
qed

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*]: $x \oplus \mathbf{0} = x$
by (*simp only*: *xor-def compl-zero conj-one-right conj-zero-right disj-zero-right*)

lemma *xor-zero-left* [*simp*]: $\mathbf{0} \oplus x = x$
by (*subst xor-commute*) (*rule xor-zero-right*)

lemma *xor-one-right* [*simp*]: $x \oplus \mathbf{1} = \sim x$
by (*simp only*: *xor-def compl-one conj-zero-right conj-one-right disj-zero-left*)

lemma *xor-one-left* [*simp*]: $\mathbf{1} \oplus x = \sim x$
by (*subst xor-commute*) (*rule xor-one-right*)

lemma *xor-self* [*simp*]: $x \oplus x = \mathbf{0}$
by (*simp only*: *xor-def conj-cancel-right conj-cancel-left disj-zero-right*)

lemma *xor-left-self* [*simp*]: $x \oplus (x \oplus y) = y$
by (*simp only*: *xor-assoc* [*symmetric*] *xor-self xor-zero-left*)

lemma *xor-compl-left*: $\sim x \oplus y = \sim (x \oplus y)$
apply (*simp only*: *xor-def de-Morgan-disj de-Morgan-conj double-compl*)
apply (*simp only*: *conj-disj-distribs*)
apply (*simp only*: *conj-cancel-right conj-cancel-left*)
apply (*simp only*: *disj-zero-left disj-zero-right*)
apply (*simp only*: *disj-ac conj-ac*)
done

lemma *xor-compl-right*: $x \oplus \sim y = \sim (x \oplus y)$
apply (*simp only*: *xor-def de-Morgan-disj de-Morgan-conj double-compl*)
apply (*simp only*: *conj-disj-distribs*)
apply (*simp only*: *conj-cancel-right conj-cancel-left*)
apply (*simp only*: *disj-zero-left disj-zero-right*)
apply (*simp only*: *disj-ac conj-ac*)
done

lemma *xor-cancel-right* [*simp*]: $x \oplus \sim x = \mathbf{1}$
by (*simp only*: *xor-compl-right xor-self compl-zero*)

lemma *xor-cancel-left* [*simp*]: $\sim x \oplus x = \mathbf{1}$
by (*subst xor-commute*) (*rule xor-cancel-right*)

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

proof −
  have $(x \sqcap y \sqcap \sim z) \sqcup (x \sqcap \sim y \sqcap z) =$
      $(y \sqcap x \sqcap \sim x) \sqcup (z \sqcap x \sqcap \sim x) \sqcup (x \sqcap y \sqcap \sim z) \sqcup (x \sqcap \sim y \sqcap z)$
    by (*simp only*: *conj-cancel-right conj-zero-right disj-zero-left*)
  thus $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
    by (*simp* (*no-asm-use*) *only*:
      xor-def de-Morgan-disj de-Morgan-conj double-compl
      conj-disj-distribs conj-ac disj-ac)
qed

lemma *conj-xor-distrib2*:
  $(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$
proof −
  have $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
    by (*rule conj-xor-distrib*)
  thus $(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$
    by (*simp only*: *conj-commute*)
qed

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: $(x \leq y) \equiv (fst\ x < fst\ y) \vee (fst\ x = fst\ y \wedge snd\ x \leq snd\ y)$
  prod-less-def: $(x < y) \equiv (fst\ x < fst\ y) \vee (fst\ x = fst\ y \wedge snd\ x < snd\ y)$ ..

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

lemma [*code func*]:
  $(x1::'a::\{ord,\ eq\},\ y1) \leq (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 = x2 \wedge y1 \leq y2$
  $(x1::'a::\{ord,\ eq\},\ y1) < (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 = x2 \wedge y1 < y2$
  unfolding *prod-ord-defs* **by** *simp-all*

lemma [*code*]:
  $(x1,\ y1) \leq (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 = x2 \wedge y1 \leq y2$
  $(x1,\ y1) < (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 = x2 \wedge y1 < y2$
  unfolding *prod-ord-defs* **by** *simp-all*

instance $*$ :: (*order*, *order*) *order*
  by *default* (*auto simp*: *prod-ord-defs intro*: *order-less-trans*)

instance $*$ :: (*linorder*, *linorder*) *linorder*
  by *default* (*auto simp*: *prod-le-def*)

instance $*$ :: (*linorder*, *linorder*) *distrib-lattice*
  inf-prod-def: *inf* $\equiv$ *min*
  sup-prod-def: *sup* $\equiv$ *max*
  by *intro-classes*
    (*auto simp add*: *inf-prod-def sup-prod-def min-max.sup-inf-distrib1*)

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* $\Rightarrow$ *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* $\Rightarrow$ *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*
unfolding *nibble-of-nat-def Let-def* **by** *auto*

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
unfolding *nibble-of-nat-def Let-def* **by** *auto*

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*
by (*cases n*) (*simp-all only*: *nat-of-nibble.simps nibble-of-nat-simps*)

lemma *nat-of-nibble-of-nat*: *nat-of-nibble* (*nibble-of-nat n*) = *n mod 16*
proof −
have *nibble-nat-enum*:

n mod 16 ∈ {*0*, *1*, *2*, *3*, *4*, *5*, *6*, *7*, *8*, *9*, *10*, *11*, *12*, *13*, *14*, *15*}
proof –
have *set-unfold*: ⋀*n*. {*0..Suc n*} = *insert* (*Suc n*) {*0..n*} **by** *auto*
have (*n::nat*) *mod 16* ∈ {*0..Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))))))))))))} **by** *simp*
from *this* [*simplified set-unfold atLeastAtMost-singleton*]
show *?thesis* **by** *auto*
qed
then show *?thesis* **unfolding** *nibble-of-nat-def Let-def*
by *auto*
qed

lemma *inj-nat-of-nibble*: *inj nat-of-nibble*
by (*rule inj-on-inverseI*) (*rule nibble-of-nat-of-nibble*)

lemma *nat-of-nibble-eq*: *nat-of-nibble n* = *nat-of-nibble m* ⟷ *n* = *m*
by (*rule inj-eq*) (*rule inj-nat-of-nibble*)

lemma *nat-of-nibble-less-16*: *nat-of-nibble n* < *16*
by (*cases n*) *auto*

lemma *nat-of-nibble-div-16*: *nat-of-nibble n div 16* = *0*
by (*cases n*) *auto*

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*)
unfolding *nibble-of-nat-norm* [*of n*, *symmetric*] *nibble-pair-of-nat-def* **..**

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*)
unfolding *char-of-nat-def Let-def nibble-pair-of-nat-def*
by (*auto simp add*: *div-add1-eq mod-add1-eq nat-of-nibble-div-16 nibble-of-nat-norm nibble-of-nat-of-nibble*)

```
lemma char-of-nat-of-char:
  char-of-nat (nat-of-char c) = c
  by (cases c) (simp add: nat-of-char.simps, simp add: Char-char-of-nat)

lemma nat-of-char-of-nat:
  nat-of-char (char-of-nat n) = n mod 256
proof −
  from mod-div-equality [of n, symmetric, of 16]
  have mod-mult-self3: ⋀m k n :: nat. (k ∗ n + m) mod n = m mod n
  proof −
    fix m k n :: nat
    show (k ∗ n + m) mod n = m mod n
      by (simp only: mod-mult-self1 [symmetric, of m n k] add-commute)
  qed
  from mod-div-decomp [of n 256] obtain k l where n: n = k ∗ 256 + l
    and k: k = n div 256 and l: l = n mod 256 by blast
  have 16: (0::nat) < 16 by auto
  have 256: (256 :: nat) = 16 ∗ 16 by auto
  have l-256: l mod 256 = l using l by auto
  have l-div-256: l div 16 ∗ 16 mod 256 = l div 16 ∗ 16
    using l by auto
  have aux2: (k ∗ 256 mod 16 + l mod 16) div 16 = 0
    unfolding 256 mult-assoc [symmetric] mod-mult-self-is-0 by simp
  have aux3: (k ∗ 256 + l) div 16 = k ∗ 16 + l div 16
    unfolding div-add1-eq [of k ∗ 256 l 16] aux2 256
      mult-assoc [symmetric] div-mult-self-is-m [OF 16] by simp
  have aux4: (k ∗ 256 + l) mod 16 = l mod 16
    unfolding 256 mult-assoc [symmetric] mod-mult-self3 ..
  show ?thesis
    by (simp add: nat-of-char.simps char-of-nat-def nibble-of-pair
      nat-of-nibble-of-nat mod-mult-distrib
      n aux3 mod-mult-self3 l-256 aux4 mod-add1-eq [of 256 ∗ k] l-div-256)
qed

lemma nibble-pair-of-nat-char:
  nibble-pair-of-nat (nat-of-char (Char n m)) = (n, m)
proof −
  have nat-of-nibble-256:
    ⋀n m. (nat-of-nibble n ∗ 16 + nat-of-nibble m) mod 256 =
      nat-of-nibble n ∗ 16 + nat-of-nibble m
  proof −
    fix n m
    have nat-of-nibble-less-eq-15: ⋀n. nat-of-nibble n ≤ 15
      using Suc-leI [OF nat-of-nibble-less-16] by (auto simp add: nat-number)
    have less-eq-240: nat-of-nibble n ∗ 16 ≤ 240
      using nat-of-nibble-less-eq-15 by auto
    have nat-of-nibble n ∗ 16 + nat-of-nibble m ≤ 240 + 15
       by (rule add-le-mono [of - 240 - 15]) (auto intro: nat-of-nibble-less-eq-15
less-eq-240)
```

then have *nat-of-nibble* $n * 16 +$ *nat-of-nibble* $m < 256$ (**is** *?rhs* < -) **by** *auto*
then show *?rhs mod 256* = *?rhs* **by** *auto*
qed
show *?thesis*
unfolding *nibble-pair-of-nat-def Char-char-of-nat nat-of-char-of-nat nat-of-nibble-256*
by (*simp add*: *add-commute nat-of-nibble-div-16 nibble-of-nat-norm nibble-of-nat-of-nibble*)
qed

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
fix n :: *nibble*
show $n \leq n$ **unfolding** *nibble-less-eq-def nibble-less-def* **by** *auto*
next
fix $n\ m\ q$ :: *nibble*
assume $n \leq m$
and $m \leq q$
then show $n \leq q$ **unfolding** *nibble-less-eq-def nibble-less-def* **by** *auto*
next
fix $n\ m$ :: *nibble*
assume $n \leq m$
and $m \leq n$
then show $n = m$
unfolding *nibble-less-eq-def nibble-less-def*
by (*auto simp add*: *nat-of-nibble-eq*)
next
fix $n\ m$ :: *nibble*
show $n < m \longleftrightarrow n \leq m \wedge n \neq m$
unfolding *nibble-less-eq-def nibble-less-def less-le*
by (*auto simp add*: *nat-of-nibble-eq*)

next
  fix *n m* :: *nibble*
  show $n \leq m \vee m \leq n$
    unfolding *nibble-less-eq-def* **by** *auto*
qed

instance *nibble* :: *distrib-lattice*
    inf ≡ *min*
    sup ≡ *max*
  by *default* (*auto simp add*:
    inf-nibble-def sup-nibble-def min-max.sup-inf-distrib1)

instance *char* :: *linorder*
  char-less-eq-def: $c1 \leq c2 \equiv$ *case c1 of Char n1 m1* ⇒ *case c2 of Char n2 m2* ⇒
    $n1 < n2 \vee n1 = n2 \wedge m1 \leq m2$
  char-less-def: $c1 < c2 \equiv$ *case c1 of Char n1 m1* ⇒ *case c2 of Char n2 m2* ⇒
    $n1 < n2 \vee n1 = n2 \wedge m1 < m2$
  by *default* (*auto simp*: *char-less-eq-def char-less-def split*: *char.splits*)

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

instance *char* :: *distrib-lattice*
    inf ≡ *min*
    sup ≡ *max*
  by *default* (*auto simp add*:
    inf-char-def sup-char-def min-max.sup-inf-distrib1)

lemma [*simp*, *code func*]:
  shows *char-less-eq-simp*: *Char n1 m1* ≤ *Char n2 m2* ⟷ $n1 < n2 \vee n1 = n2$
$\wedge\ m1 \leq m2$
  and *char-less-simp*:   *Char n1 m1* < *Char n2 m2* ⟷ $n1 < n2 \vee n1 = n2$
$\wedge\ m1 < m2$
  unfolding *char-less-eq-def char-less-def* **by** *simp-all*

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*
  by (*cases k*) *simp-all*

lemma *index*:
  $(\bigwedge k{::}index.\ PROP\ P\ k) \equiv (\bigwedge k{::}int.\ PROP\ P\ (index\text{-}of\text{-}int\ k))$
proof
  fix *k* :: *int*
  assume $\bigwedge k{::}index.\ PROP\ P\ k$
  then show *PROP P* (*index-of-int k*) **.**
next
  fix *k* :: *index*
  assume $\bigwedge k{::}int.\ PROP\ P\ (index\text{-}of\text{-}int\ k)$
  then have *PROP P* (*index-of-int* (*int-of-index k*)) **.**
  then show *PROP P k* **by** *simp*
qed

lemma [*code func*]: *size* (*k*::*index*) = *0*
  by (*cases k*) *simp-all*

11.2 Built-in integers as datatype on numerals

instance *index* :: *number*
  number-of ≡ *index-of-int* **..**

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

lemma *number-of-index-id* [*simp*]:
  number-of (*int-of-index k*) = *k*
  unfolding *number-of-index-def* **by** *simp*

lemma *number-of-index-shift*:
  number-of k = *index-of-int* (*number-of k*)
  by (*simp add*: *number-of-is-id number-of-index-def*)

lemma *int-of-index-number-of* [*simp*]:
  int-of-index (*number-of k*) = *number-of k*
  unfolding *number-of-index-def number-of-is-id* **by** *simp*

11.3 Basic arithmetic

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

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

instance *index* :: *plus*
  [*simp*]: $k + l \equiv$ *index-of-int* (*int-of-index k* + *int-of-index l*) **..**
lemmas [*code func del*] = *plus-index-def*
lemma *plus-index-code* [*code func*]:
  index-of-int k + *index-of-int l* = *index-of-int* $(k + l)$
  unfolding *plus-index-def* **by** *simp*

instance *index* :: *minus*
  [*simp*]: $- k \equiv$ *index-of-int* (− *int-of-index k*)
  [*simp*]: $k - l \equiv$ *index-of-int* (*int-of-index k* − *int-of-index l*) **..**
lemmas [*code func del*] = *uminus-index-def minus-index-def*
lemma *uminus-index-code* [*code func*]:
  − *index-of-int k* ≡ *index-of-int* $(- k)$
  unfolding *uminus-index-def* **by** *simp*
lemma *minus-index-code* [*code func*]:
  index-of-int k − *index-of-int l* = *index-of-int* $(k - l)$
  unfolding *minus-index-def* **by** *simp*

instance *index* :: *times*
  [*simp*]: $k * l \equiv$ *index-of-int* (*int-of-index k* * *int-of-index l*) **..**
lemmas [*code func del*] = *times-index-def*
lemma *times-index-code* [*code func*]:
  index-of-int k * *index-of-int l* = *index-of-int* $(k * l)$
  unfolding *times-index-def* **by** *simp*

instance *index* :: *ord*
  [*simp*]: $k \leq l \equiv$ *int-of-index k* ≤ *int-of-index l*
  [*simp*]: $k < l \equiv$ *int-of-index k* < *int-of-index l* **..**
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* $\longleftrightarrow k \leq l$
  unfolding *less-eq-index-def* **by** *simp*
lemma *less-index-code* [*code func*]:
  index-of-int k < *index-of-int l* $\longleftrightarrow k < l$
  unfolding *less-index-def* **by** *simp*

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*) **..**

instance *index* :: *ring-1*
  by *default* (*auto simp add*: *left-distrib right-distrib*)

lemma *of-nat-index*: *of-nat n* = *index-of-int* (*of-nat n*)
proof (*induct n*)
  case *0* **show** *?case* **by** *simp*
next
  case (*Suc n*)
  then have *int-of-index* (*index-of-int* (*int n*))
    = *int-of-index* (*of-nat n*) **by** *simp*
  then have *int n* = *int-of-index* (*of-nat n*) **by** *simp*
  then show *?case* **by** *simp*
qed

instance *index* :: *number-ring*
  by *default*
    (*simp-all add*: *left-distrib number-of-index-def of-int-of-nat of-nat-index*)

lemma *zero-index-code* [*code inline*, *code func*]:
  (*0*::*index*) = *Numeral0*
  by *simp*

lemma *one-index-code* [*code inline*, *code func*]:
  (*1*::*index*) = *Numeral1*
  by *simp*

instance *index* :: *abs*
  $|k| \equiv$ *if* $k < 0$ *then* $-k$ *else* k **..**

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*))
  by (*simp add*: *number-of-index-shift Let-def split-def divAlg-mod-div*) *arith*

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*))
  by (*auto simp add*: *number-of-index-shift Let-def split-def*) *arith*

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*))
by (*auto simp add*: *nat-of-index-aux-def nat-of-index-def*)

lemma *nat-of-index-code* [*code*]:
nat-of-index i = *nat-of-index-aux i 0*
by (*simp add*: *nat-of-index-aux-def*)

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*
unfolding *index-of-nat-def* **by** *simp-all*

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*)
unfolding *index-of-nat-def nat-of-index-def* **by** *simp-all*

11.5 ML interface

ML ⟪
structure Index =
struct

fun mk k = @{*term index-of-int*} $ *HOLogic.mk-number* @{*typ index*} *k*;

end;
⟫

11.6 Code serialization

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

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

setup ⟪

fold (*fn target* => *CodeTarget.add-pretty-numeral target true*
  @{*const-name number-index-inst.number-of-index*}
  @{*const-name Numeral.B0*} @{*const-name Numeral.B1*}
  @{*const-name Numeral.Pls*} @{*const-name Numeral.Min*}
  @{*const-name Numeral.Bit*}
) [*SML*, *OCaml*, *Haskell*]
⟫

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*
  by (*cases s*) *simp-all*

12.2 ML interface

```
ML ⟪
structure Message-String =
struct

fun mk s = @{term STR} $ HOLogic.mk-string s;

end;
⟫
```

12.3 Code serialization

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

setup ⟪
let
  val charr = @{const-name Char}
  val nibbles = [@{const-name Nibble0}, @{const-name Nibble1},
    @{const-name Nibble2}, @{const-name Nibble3},
    @{const-name Nibble4}, @{const-name Nibble5},
    @{const-name Nibble6}, @{const-name Nibble7},
    @{const-name Nibble8}, @{const-name Nibble9},
    @{const-name NibbleA}, @{const-name NibbleB},
    @{const-name NibbleC}, @{const-name NibbleD},
    @{const-name NibbleE}, @{const-name NibbleF}];
in
```

fold (*fn target* => *CodeTarget.add-pretty-message target*
charr nibbles @{*const-name Nil*} @{*const-name Cons*} @{*const-name STR*})
[*SML*, *OCaml*, *Haskell*]
end
⟫

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*]: $CONS\ M\ N \neq NIL$
and *NIL-not-CONS* [*iff*]: $NIL \neq CONS\ M\ N$
and *CONS-inject* [*iff*]: $(CONS\ K\ M) = (CONS\ L\ N) = (K = L \wedge M = N)$
by (*simp-all add*: *NIL-def CONS-def*)

lemma *CONS-mono*: $M \subseteq M' \Longrightarrow N \subseteq N' \Longrightarrow CONS\ M\ N \subseteq CONS\ M'\ N'$
by (*simp add*: *CONS-def In1-mono Scons-mono*)

lemma *CONS-UN1*: $CONS\ M\ (\bigcup x.\ f\ x) = (\bigcup x.\ CONS\ M\ (f\ x))$
— A continuity result?
by (*simp add*: *CONS-def In1-UN1 Scons-UN1-y*)

definition $List\text{-}case\ c\ h = Datatype.Case\ (\lambda\text{-}.\ c)\ (Datatype.Split\ h)$

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

by (*simp-all add*: *List-case-def NIL-def CONS-def*)

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
fix x
assume $x \in LList\ A$
then show $x \in LList\ B$
proof *coinduct*
case *LList*
then show *?case* **using** *subset*
by *cases blast+*
qed
qed

consts
LList-corec-aux :: $nat \Rightarrow ('a \Rightarrow ('b\ Datatype.item \times 'a)\ option) \Rightarrow$
$'a \Rightarrow 'b\ Datatype.item$
primrec
LList-corec-aux $0\ f\ x = \{\}$
LList-corec-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-corec $a\ f =$
(*case* $f\ a$ *of* $None \Rightarrow NIL \mid Some\ (z, w) \Rightarrow CONS\ z\ (LList\text{-}corec\ w\ f)$)
(**is** *?lhs* = *?rhs*)
proof
show *?lhs* $\subseteq$ *?rhs*
apply (*unfold LList-corec-def*)
apply (*rule UN-least*)
apply (*case-tac k*)
apply (*simp-all* (*no-asm-simp*) *split*: *option.splits*)
apply (*rule allI impI subset-refl* [*THEN CONS-mono*] *UNIV-I* [*THEN UN-upper*])+

```
    done
  show ?rhs ⊆ ?lhs
    apply (simp add: LList-corec-def split: option.splits)
    apply (simp add: CONS-UN1)
    apply safe
     apply (rule-tac a = Suc ?k in UN-I, simp, simp)+
    done
qed

lemma LList-corec-type: LList-corec a f ∈ LList UNIV
proof −
  have ∃ x. LList-corec a f = LList-corec x f by blast
  then show ?thesis
  proof coinduct
    case (LList L)
    then obtain x where L: L = LList-corec x f by blast
    show ?case
    proof (cases f x)
      case None
      then have LList-corec x f = NIL
        by (simp add: LList-corec)
      with L have ?NIL by simp
      then show ?thesis ..
    next
      case (Some p)
      then have LList-corec x f = CONS (fst p) (LList-corec (snd p) f)
        by (simp add: LList-corec split: prod.split)
      with L have ?CONS by auto
      then show ?thesis ..
    qed
  qed
qed
```

13.3 Abstract type definition

```
typedef 'a llist = LList (range Datatype.Leaf) :: 'a Datatype.item set
proof
  show NIL ∈ ?llist ..
qed

lemma NIL-type: NIL ∈ llist
  unfolding llist-def by (rule LList.NIL)

lemma CONS-type: a ∈ range Datatype.Leaf =⇒
    M ∈ llist =⇒ CONS a M ∈ llist
  unfolding llist-def by (rule LList.CONS)

lemma llistI: x ∈ LList (range Datatype.Leaf) =⇒ x ∈ llist
  by (simp add: llist-def)
```

lemma *llistD*: $x \in llist \Longrightarrow x \in LList\ (range\ Datatype.Leaf)$
  by $(simp\ add{:}\ llist\text{-}def)$

lemma *Rep-llist-UNIV*: $Rep\text{-}llist\ x \in LList\ UNIV$
proof $-$
  have $Rep\text{-}llist\ x \in llist$ **by** $(rule\ Rep\text{-}llist)$
  then have $Rep\text{-}llist\ x \in LList\ (range\ Datatype.Leaf)$
    by $(simp\ add{:}\ llist\text{-}def)$
  also have $\ldots \subseteq LList\ UNIV$ **by** $(rule\ LList\text{-}mono)\ simp$
  finally show *?thesis* **.**
qed

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*]: $LCons\ x\ xs \neq LNil$
  apply $(simp\ add{:}\ LNil\text{-}def\ LCons\text{-}def)$
  apply $(subst\ Abs\text{-}llist\text{-}inject)$
    apply $(auto\ intro{:}\ NIL\text{-}type\ CONS\text{-}type\ Rep\text{-}llist)$
  done

lemma *LNil-not-LCons* [*iff*]: $LNil \neq LCons\ x\ xs$
  by $(rule\ LCons\text{-}not\text{-}LNil\ [symmetric])$

lemma *LCons-inject* [*iff*]: $(LCons\ x\ xs = LCons\ y\ ys) = (x = y \wedge xs = ys)$
  apply $(simp\ add{:}\ LCons\text{-}def)$
  apply $(subst\ Abs\text{-}llist\text{-}inject)$
    apply $(auto\ simp\ add{:}\ Rep\text{-}llist\text{-}inject\ intro{:}\ CONS\text{-}type\ Rep\text{-}llist)$
  done

lemma *Rep-llist-LNil*: $Rep\text{-}llist\ LNil = NIL$
  by $(simp\ add{:}\ LNil\text{-}def\ add{:}\ Abs\text{-}llist\text{-}inverse\ NIL\text{-}type)$

lemma *Rep-llist-LCons*: $Rep\text{-}llist\ (LCons\ x\ l) =$
    $CONS\ (Datatype.Leaf\ x)\ (Rep\text{-}llist\ l)$
  by $(simp\ add{:}\ LCons\text{-}def\ Abs\text{-}llist\text{-}inverse\ CONS\text{-}type\ Rep\text{-}llist)$

lemma *llist-cases* [*cases type*: *llist*]:
  obtains
    $(LNil)\ l = LNil$
  $|\ (LCons)\ x\ l'$ **where** $l = LCons\ x\ l'$
proof $(cases\ l)$
  case $(Abs\text{-}llist\ L)$
  from $\langle L \in llist\rangle$ **have** $L \in LList\ (range\ Datatype.Leaf)$ **by** $(rule\ llistD)$
  then show *?thesis*
  proof *cases*
    case *NIL*
    with *Abs-llist* **have** $l = LNil$ **by** $(simp\ add{:}\ LNil\text{-}def)$

```
    with LNil show ?thesis .
  next
    case (CONS a K)
    then have K ∈ llist by (blast intro: llistI)
    then obtain l' where K = Rep-llist l' by cases
    with CONS and Abs-llist obtain x where l = LCons x l'
      by (auto simp add: LCons-def Abs-llist-inject)
    with LCons show ?thesis .
  qed
qed


definition
  llist-case c d l =
    List-case c (λx y. d (inv Datatype.Leaf x) (Abs-llist y)) (Rep-llist l)

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

lemma llist-case-LNil [simp]: llist-case c d LNil = c
  by (simp add: llist-case-def LNil-def
    NIL-type Abs-llist-inverse)

lemma llist-case-LCons [simp]: llist-case c d (LCons M N) = d M N
  by (simp add: llist-case-def LCons-def
    CONS-type Abs-llist-inverse Rep-llist Rep-llist-inverse inj-Leaf)


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

lemma LList-corec-type2:
  LList-corec a
    (λz. case f z of None ⇒ None
      | Some (v, w) ⇒ Some (Datatype.Leaf v, w)) ∈ llist
  (is ?corec a ∈ -)
proof (unfold llist-def)
  let LList-corec a ?g = ?corec a
  have ∃ x. ?corec a = ?corec x by blast
  then show ?corec a ∈ LList (range Datatype.Leaf)
  proof coinduct
    case (LList L)
```

```
    then obtain x where L: L = ?corec x by blast
    show ?case
    proof (cases f x)
      case None
      then have ?corec x = NIL
        by (simp add: LList-corec)
      with L have ?NIL by simp
      then show ?thesis ..
    next
      case (Some p)
      then have ?corec x =
          CONS (Datatype.Leaf (fst p)) (?corec (snd p))
        by (simp add: LList-corec split: prod.split)
      with L have ?CONS by auto
      then show ?thesis ..
    qed
  qed
qed

lemma llist-corec:
  llist-corec a f =
    (case f a of None ⇒ LNil | Some (z, w) ⇒ LCons z (llist-corec w f))
proof (cases f a)
  case None
  then show ?thesis
    by (simp add: llist-corec-def LList-corec LNil-def)
next
  case (Some p)

  let ?corec a = llist-corec a f
  let ?rep-corec a =
    LList-corec a
      (λz. case f z of None ⇒ None
        | Some (v, w) ⇒ Some (Datatype.Leaf v, w))

  have ?corec a = Abs-llist (?rep-corec a)
    by (simp only: llist-corec-def)
  also from Some have ?rep-corec a =
      CONS (Datatype.Leaf (fst p)) (?rep-corec (snd p))
    by (simp add: LList-corec split: prod.split)
  also have ?rep-corec (snd p) = Rep-llist (?corec (snd p))
    by (simp only: llist-corec-def Abs-llist-inverse LList-corec-type2)
  finally have ?corec a = LCons (fst p) (?corec (snd p))
    by (simp only: LCons-def)
  with Some show ?thesis by (simp split: prod.split)
qed
```

13.4 Equality as greatest fixed-point – the bisimulation principle

coinductive-set *EqLList* **for** r
where *EqNIL*: (*NIL*, *NIL*) $\in$ *EqLList* r
  | *EqCONS*: $(a,\ b) \in r \Longrightarrow (M,\ N) \in$ *EqLList* $r \Longrightarrow$
    $(CONS\ a\ M,\ CONS\ b\ N) \in$ *EqLList* r

lemma *EqLList-unfold*:
  EqLList r = *dsum* (*diag* {*Datatype.Numb 0*}) (*dprod* r (*EqLList* r))
 by (*fast intro*!: *EqLList.intros* [*unfolded NIL-def CONS-def*]
    elim: *EqLList.cases* [*unfolded NIL-def CONS-def*])

lemma *EqLList-implies-ntrunc-equality*:
  $(M,\ N) \in$ *EqLList* (*diag* A) $\Longrightarrow$ *ntrunc* k M = *ntrunc* k N
 apply (*induct k arbitrary*: *M N rule*: *nat-less-induct*)
 apply (*erule EqLList.cases*)
  apply (*safe del*: *equalityI*)
 apply (*case-tac n*)
  apply *simp*
 apply (*rename-tac* n')
 apply (*case-tac* n')
  apply (*simp-all add*: *CONS-def less-Suc-eq*)
 done

lemma *Domain-EqLList*: *Domain* (*EqLList* (*diag* A)) $\subseteq$ *LList* A
 apply (*rule subsetI*)
 apply (*erule LList.coinduct*)
 apply (*subst* (*asm*) *EqLList-unfold*)
 apply (*auto simp add*: *NIL-def CONS-def*)
 done

lemma *EqLList-diag*: *EqLList* (*diag* A) = *diag* (*LList* A)
 (**is** *?lhs* = *?rhs*)
proof
 show *?lhs* $\subseteq$ *?rhs*
  apply (*rule subsetI*)
  apply (*rule-tac* $p = x$ **in** *PairE*)
  apply *clarify*
  apply (*rule diag-eqI*)
   apply (*rule EqLList-implies-ntrunc-equality* [*THEN ntrunc-equality*],
    assumption)
  apply (*erule DomainI* [*THEN Domain-EqLList* [*THEN subsetD*]])
  done
 {
  fix *M N* **assume** $(M,\ N) \in$ *diag* (*LList* A)
  then have $(M,\ N) \in$ *EqLList* (*diag* A)
  proof *coinduct*
   case (*EqLList M N*)
   then obtain L **where** L: $L \in$ *LList* A **and** *MN*: $M = L$ $N = L$ **by** *blast*

from L **show** *?case*
proof *cases*
case *NIL* **with** *MN* **have** *?EqNIL* **by** *simp*
then show *?thesis* **..**
next
case *CONS* **with** *MN* **have** *?EqCONS* **by** (*simp add*: *diagI*)
then show *?thesis* **..**
qed
qed
}
then show $?rhs \subseteq ?lhs$ **by** *auto*
qed

lemma *EqLList-diag-iff* [*iff*]: $(p \in EqLList\ (diag\ A)) = (p \in diag\ (LList\ A))$
by (*simp only*: *EqLList-diag*)

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*: $\bigwedge M\ N.\ (M, N) \in r \Longrightarrow$
$M = NIL \land N = NIL \lor$
$(\exists\, a\ b\ M'\ N'.$
$M = CONS\ a\ M' \land N = CONS\ b\ N' \land (a, b) \in diag\ A \land$
$((M', N') \in r \lor (M', N') \in EqLList\ (diag\ A)))$
shows $M = N$
proof $-$
from r **have** $(M, N) \in EqLList\ (diag\ A)$
proof *coinduct*
case *EqLList*
then show *?case* **by** (*rule step*)
qed
then show *?thesis* **by** *auto*
qed

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

```
proof -
  let ?bisim = {(f L, g L) | L. L ∈ LList A}
  have (f M, g M) ∈ ?bisim using M by blast
  then show ?thesis
  proof (coinduct taking: A rule: LList-equalityI)
    case (EqLList M N)
    then obtain L where MN: M = f L N = g L and L: L ∈ LList A by blast
    from L show ?case
    proof (cases L)
      case NIL
      with fun-NIL and MN have (M, N) ∈ diag (LList A) by auto
      then have (M, N) ∈ EqLList (diag A) ..
      then show ?thesis by cases simp-all
    next
      case (CONS a K)
      from fun-CONS and ‹a ∈ A› ‹K ∈ LList A›
      have ?fun-CONS a K (is ?NIL ∨ ?CONS) .
      then show ?thesis
      proof
        assume ?NIL
        with MN CONS have (M, N) ∈ diag (LList A) by auto
        then have (M, N) ∈ EqLList (diag A) ..
        then show ?thesis by cases simp-all
      next
        assume ?CONS
        with CONS obtain a b M' N' where
            fg: (f L, g L) = (CONS a M', CONS b N')
          and ab: (a, b) ∈ diag A
          and M'N': (M', N') ∈ ?bisim ∪ diag (LList A)
          by blast
        from M'N' show ?thesis
        proof
          assume (M', N') ∈ ?bisim
          with MN fg ab show ?thesis by simp
        next
          assume (M', N') ∈ diag (LList A)
          then have (M', N') ∈ EqLList (diag A) ..
          with MN fg ab show ?thesis by simp
        qed
      qed
    qed
  qed
qed
```

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

```
lemma equals-LList-corec:
  assumes h: ⋀x. h x =
    (case f x of None ⇒ NIL | Some (z, w) ⇒ CONS z (h w))
  shows h x = (λx. LList-corec x f) x
```

proof −
  def $h' \equiv \lambda x.\ \mathit{LList\text{-}corec}\ x\ f$
  then have h': $\bigwedge x.\ h'\ x =$
      $(\mathit{case}\ f\ x\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{NIL} \mid \mathit{Some}\ (z,\ w) \Rightarrow \mathit{CONS}\ z\ (h'\ w))$
    unfolding h'-*def* **by** $(\mathit{simp\ add}\colon \mathit{LList\text{-}corec})$
  have $(h\ x,\ h'\ x) \in \{(h\ u,\ h'\ u) \mid u.\ \mathit{True}\}$ **by** *blast*
  then show $h\ x = h'\ x$
  proof $(\mathit{coinduct\ taking}\colon \mathit{UNIV\ rule}\colon \mathit{LList\text{-}equalityI})$
    case $(\mathit{EqLList}\ M\ N)$
    then obtain x **where** $\mathit{MN}\colon M = h\ x\ N = h'\ x$ **by** *blast*
    show *?case*
    proof $(\mathit{cases}\ f\ x)$
      case *None*
      with $h\ h'\ \mathit{MN}$ **have** *?EqNIL* **by** *simp*
      then show *?thesis* **..**
    next
      case $(\mathit{Some}\ p)$
      with $h\ h'\ \mathit{MN}$ **have** $M = \mathit{CONS}\ (\mathit{fst}\ p)\ (h\ (\mathit{snd}\ p))$
        and $N = \mathit{CONS}\ (\mathit{fst}\ p)\ (h'\ (\mathit{snd}\ p))$
        by $(\mathit{simp\text{-}all\ split}\colon \mathit{prod.split})$
      then have *?EqCONS* **by** $(\mathit{auto\ iff}\colon \mathit{diag\text{-}iff})$
      then show *?thesis* **..**
    qed
  qed
qed

lemma *llist-equalityI*
  $[\mathit{consumes}\ 1,\ \mathit{case\text{-}names}\ \mathit{Eqllist},\ \mathit{case\text{-}conclusion}\ \mathit{Eqllist}\ \mathit{EqLNil}\ \mathit{EqLCons}]$:
  assumes $r\colon (l1,\ l2) \in r$
    and $\mathit{step}\colon \bigwedge q.\ q \in r \Longrightarrow$
      $q = (\mathit{LNil},\ \mathit{LNil}) \vee$
        $(\exists\, l1\ l2\ a\ b.$
          $q = (\mathit{LCons}\ a\ l1,\ \mathit{LCons}\ b\ l2) \wedge a = b \wedge$
            $((l1,\ l2) \in r \vee l1 = l2))$
      (**is** $\bigwedge q.\ \text{-} \Longrightarrow\ ?\mathit{EqLNil}\ q \vee\ ?\mathit{EqLCons}\ q$)
  shows $l1 = l2$
proof −
  def $M \equiv \mathit{Rep\text{-}llist}\ l1$ **and** $N \equiv \mathit{Rep\text{-}llist}\ l2$
  with r **have** $(M,\ N) \in \{(\mathit{Rep\text{-}llist}\ l1,\ \mathit{Rep\text{-}llist}\ l2) \mid l1\ l2.\ (l1,\ l2) \in r\}$
    by *blast*
  then have $M = N$
  proof $(\mathit{coinduct\ taking}\colon \mathit{UNIV\ rule}\colon \mathit{LList\text{-}equalityI})$
    case $(\mathit{EqLList}\ M\ N)$
    then obtain $l1\ l2$ **where**
        $\mathit{MN}\colon M = \mathit{Rep\text{-}llist}\ l1\ N = \mathit{Rep\text{-}llist}\ l2$ **and** $r\colon (l1,\ l2) \in r$
      by *auto*
    from $\mathit{step}\ [\mathit{OF}\ r]$ **show** *?case*
    proof

```
    assume ?EqLNil (l1, l2)
    with MN have ?EqNIL by (simp add: Rep-llist-LNil)
    then show ?thesis ..
  next
    assume ?EqLCons (l1, l2)
    with MN have ?EqCONS
      by (force simp add: Rep-llist-LCons EqLList-diag intro: Rep-llist-UNIV)
    then show ?thesis ..
  qed
 qed
 then show ?thesis by (simp add: M-def N-def Rep-llist-inject)
qed

lemma llist-fun-equalityI
  [case-names LNil LCons, case-conclusion LCons EqLNil EqLCons]:
  assumes fun-LNil: f LNil = g LNil
    and fun-LCons: ⋀x l.
      (f (LCons x l), g (LCons x l)) = (LNil, LNil) ∨
        (∃ l1 l2 a b.
          (f (LCons x l), g (LCons x l)) = (LCons a l1, LCons b l2) ∧
            a = b ∧ ((l1, l2) ∈ {(f u, g u) | u. True} ∨ l1 = l2))
      (is ⋀x l. ?fun-LCons x l)
  shows f l = g l
proof −
  have (f l, g l) ∈ {(f l, g l) | l. True} by blast
  then show ?thesis
  proof (coinduct rule: llist-equalityI)
    case (Eqllist q)
    then obtain l where q: q = (f l, g l) by blast
    show ?case
    proof (cases l)
      case LNil
      with fun-LNil and q have q = (g LNil, g LNil) by simp
      then show ?thesis by (cases g LNil) simp-all
    next
      case (LCons x l')
      with ‹?fun-LCons x l'› q LCons show ?thesis by blast
    qed
  qed
qed
```

13.5 Derived operations – both on the set and abstract type

13.5.1 *Lconst*

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

lemma *Lconst-fun-mono*: *mono* (*CONS M*)
  by (*simp add*: *monoI CONS-mono*)

```
lemma Lconst: Lconst M = CONS M (Lconst M)
  by (rule Lconst-def [THEN def-lfp-unfold]) (rule Lconst-fun-mono)

lemma Lconst-type:
  assumes M ∈ A
  shows Lconst M ∈ LList A
proof −
  have Lconst M ∈ {Lconst (id M)} by simp
  then show ?thesis
  proof coinduct
    case (LList N)
    then have N = Lconst M by simp
    also have ... = CONS M (Lconst M) by (rule Lconst)
    finally have ?CONS using ‹M ∈ A› by simp
    then show ?case ..
  qed
qed

lemma Lconst-eq-LList-corec: Lconst M = LList-corec M (λx. Some (x, x))
  apply (rule equals-LList-corec)
  apply simp
  apply (rule Lconst)
  done

lemma gfp-Lconst-eq-LList-corec:
    gfp (λN. CONS M N) = LList-corec M (λx. Some(x, x))
  apply (rule equals-LList-corec)
  apply simp
  apply (rule Lconst-fun-mono [THEN gfp-unfold])
  done
```

13.5.2 *Lmap* and *lmap*

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

lemma Lmap-NIL [simp]: Lmap f NIL = NIL
  and Lmap-CONS [simp]: Lmap f (CONS M N) = CONS (f M) (Lmap f N)
  by (simp-all add: Lmap-def LList-corec)

lemma Lmap-type:
  assumes M: M ∈ LList A
    and f: ⋀x. x ∈ A ⟹ f x ∈ B
  shows Lmap f M ∈ LList B
```

proof –
  from *M* **have** $Lmap\ f\ M \in \{Lmap\ f\ N \mid N.\ N \in LList\ A\}$ **by** *blast*
  then show *?thesis*
  proof *coinduct*
    case $(LList\ L)$
    then obtain *N* **where** L: $L = Lmap\ f\ N$ **and** N: $N \in LList\ A$ **by** *blast*
    from *N* **show** *?case*
    proof *cases*
      case *NIL*
      with *L* **have** *?NIL* **by** *simp*
      then show *?thesis* **..**
    next
      case $(CONS\ K\ a)$
      with *f L* **have** *?CONS* **by** *auto*
      then show *?thesis* **..**
    qed
  qed
qed

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 –
  have $(?lhs\ M,\ ?rhs\ M) \in \{(?lhs\ N,\ ?rhs\ N) \mid N.\ N \in LList\ A\}$
    using *M* **by** *blast*
  then show *?thesis*
  proof (*coinduct taking*: *range* $(\lambda N.\ N)$ *rule*: *LList-equalityI*)
    case $(EqLList\ L\ M)$
    then obtain *N* **where** LM: $L = ?lhs\ N\ M = ?rhs\ N$ **and** N: $N \in LList\ A$
by *blast*
    from *N* **show** *?case*
    proof *cases*
      case *NIL*
      with *LM* **have** *?EqNIL* **by** *simp*
      then show *?thesis* **..**
    next
      case *CONS*
      with *LM* **have** *?EqCONS* **by** *auto*
      then show *?thesis* **..**
    qed
  qed
qed

lemma *Lmap-ident*:
  assumes M: $M \in LList\ A$
  shows $Lmap\ (\lambda x.\ x)\ M = M$ (**is** $?lmap\ M = -$)
proof –
  have $(?lmap\ M,\ M) \in \{(?lmap\ N,\ N) \mid N.\ N \in LList\ A\}$ **using** *M* **by** *blast*
  then show *?thesis*

proof (*coinduct taking*: *range* ($\lambda N.\ N$) *rule*: *LList-equalityI*)
case (*EqLList L M*)
then obtain N **where** *LM*: $L = ?lmap\ N\ M = N$ **and** N: $N \in LList\ A$ **by** *blast*
from N **show** *?case*
proof *cases*
case *NIL*
with *LM* **have** *?EqNIL* **by** *simp*
then show *?thesis* **..**
next
case *CONS*
with *LM* **have** *?EqCONS* **by** *auto*
then show *?thesis* **..**
qed
qed
qed

lemma *lmap-LNil* [*simp*]: *lmap f LNil* = *LNil*
and *lmap-LCons* [*simp*]: *lmap f* (*LCons M N*) = *LCons* (*f M*) (*lmap f N*)
by (*simp-all add*: *lmap-def llist-corec*)

lemma *lmap-compose* [*simp*]: *lmap* (*f o g*) *l* = *lmap f* (*lmap g l*)
by (*coinduct l rule*: *llist-fun-equalityI*) *auto*

lemma *lmap-ident* [*simp*]: *lmap* ($\lambda x.\ x$) *l* = *l*
by (*coinduct l rule*: *llist-fun-equalityI*) *auto*

13.5.3 *Lappend*

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

lemma *Lappend-NIL-NIL* [*simp*]:
Lappend NIL NIL = *NIL*
and *Lappend-NIL-CONS* [*simp*]:
Lappend NIL (*CONS N N'*) = *CONS N* (*Lappend NIL N'*)
and *Lappend-CONS* [*simp*]:
Lappend (*CONS M M'*) *N* = *CONS M* (*Lappend M' N*)
by (*simp-all add*: *Lappend-def LList-corec*)

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

by (*erule LList-fun-equalityI*) *auto*

lemma *Lappend-NIL2*: $M \in LList\ A \Longrightarrow Lappend\ M\ NIL = M$
by (*erule LList-fun-equalityI*) *auto*

```
lemma Lappend-type:
  assumes M: M ∈ LList A and N: N ∈ LList A
  shows Lappend M N ∈ LList A
proof −
  have Lappend M N ∈ {Lappend u v | u v. u ∈ LList A ∧ v ∈ LList A}
    using M N by blast
  then show ?thesis
  proof coinduct
    case (LList L)
    then obtain M N where L: L = Lappend M N
        and M: M ∈ LList A and N: N ∈ LList A
      by blast
    from M show ?case
    proof cases
      case NIL
      from N show ?thesis
      proof cases
        case NIL
        with L and ‹M = NIL› have ?NIL by simp
        then show ?thesis ..
      next
        case CONS
        with L and ‹M = NIL› have ?CONS by simp
        then show ?thesis ..
      qed
    next
      case CONS
      with L N have ?CONS by auto
      then show ?thesis ..
    qed
  qed
qed
```

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*)
by (*simp-all add*: *lappend-def llist-corec*)

lemma *lappend-LNil1* [*simp*]: *lappend LNil l* = *l*
by (*coinduct l rule*: *llist-fun-equalityI*) *auto*

lemma *lappend-LNil2* [*simp*]: *lappend l LNil* = *l*
by (*coinduct l rule*: *llist-fun-equalityI*) *auto*

lemma *lappend-assoc*: *lappend* (*lappend l1 l2*) *l3* = *lappend l1* (*lappend l2 l3*)
  by (*coinduct l1 rule*: *llist-fun-equalityI*) *auto*

lemma *lmap-lappend-distrib*: *lmap f* (*lappend l n*) = *lappend* (*lmap f l*) (*lmap f n*)
  by (*coinduct l rule*: *llist-fun-equalityI*) *auto*

13.6 iterates

llist-fun-equalityI cannot be used here!

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

lemma *iterates*: *iterates f x* = *LCons x* (*iterates f* (*f x*))
  apply (*unfold iterates-def*)
  apply (*subst llist-corec*)
  apply *simp*
  done

lemma *lmap-iterates*: *lmap f* (*iterates f x*) = *iterates f* (*f x*)
proof −
  have (*lmap f* (*iterates f x*), *iterates f* (*f x*)) ∈
    {(*lmap f* (*iterates f u*), *iterates f* (*f u*)) | *u*. *True*} **by** *blast*
  then show *?thesis*
  proof (*coinduct rule*: *llist-equalityI*)
    case (*Eqllist q*)
    then obtain *x* **where** *q*: *q* = (*lmap f* (*iterates f x*), *iterates f* (*f x*))
      by *blast*
    also have *iterates f* (*f x*) = *LCons* (*f x*) (*iterates f* (*f* (*f x*)))
      by (*subst iterates*) *rule*
    also have *iterates f x* = *LCons x* (*iterates f* (*f x*))
      by (*subst iterates*) *rule*
    finally have *?EqLCons* **by** *auto*
    then show *?case* **..**
  qed
qed

lemma *iterates-lmap*: *iterates f x* = *LCons x* (*lmap f* (*iterates f x*))
  by (*subst lmap-iterates*) (*rule iterates*)

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

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

```
lemma iterates-equality:
  assumes h: ⋀x. h x = LCons x (lmap f (h x))
  shows h = iterates f
proof
  fix x
  have (h x, iterates f x) ∈
      {((lmap f ^ n) (h u), (lmap f ^ n) (iterates f u)) | u n. True}
  proof −
    have (h x, iterates f x) = ((lmap f ^ 0) (h x), (lmap f ^ 0) (iterates f x))
      by simp
    then show ?thesis by blast
  qed
  then show h x = iterates f x
  proof (coinduct rule: llist-equalityI)
    case (Eqllist q)
    then obtain u n where q = ((lmap f ^ n) (h u), (lmap f ^ n) (iterates f u))
        (is - = (?q1, ?q2))
      by auto
    also have ?q1 = LCons ((f ^ n) u) ((lmap f ^ Suc n) (h u))
    proof −
      have ?q1 = (lmap f ^ n) (LCons u (lmap f (h u)))
        by (subst h) rule
      also have ... = LCons ((f ^ n) u) ((lmap f ^ n) (lmap f (h u)))
        by (rule funpow-lmap)
      also have (lmap f ^ n) (lmap f (h u)) = (lmap f ^ Suc n) (h u)
        by (simp add: funpow-swap1)
      finally show ?thesis .
    qed
    also have ?q2 = LCons ((f ^ n) u) ((lmap f ^ Suc n) (iterates f u))
    proof −
      have ?q2 = (lmap f ^ n) (LCons u (iterates f (f u)))
        by (subst iterates) rule
      also have ... = LCons ((f ^ n) u) ((lmap f ^ n) (iterates f (f u)))
        by (rule funpow-lmap)
      also have (lmap f ^ n) (iterates f (f u)) = (lmap f ^ Suc n) (iterates f u)
        by (simp add: lmap-iterates funpow-swap1)
      finally show ?thesis .
    qed
    finally have ?EqLCons by (auto simp del: funpow.simps)
    then show ?case ..
  qed
qed

lemma lappend-iterates: lappend (iterates f x) l = iterates f x
proof −
  have (lappend (iterates f x) l, iterates f x) ∈
    {(lappend (iterates f u) l, iterates f u) | u. True} by blast
  then show ?thesis
  proof (coinduct rule: llist-equalityI)
```

case (*Eqllist q*)
then obtain *x* **where** *q* = (*lappend* (*iterates f x*) *l*, *iterates f x*) **by** *blast*
also have *iterates f x* = *LCons x* (*iterates f* (*f x*)) **by** (*rule iterates*)
finally have *?EqLCons* **by** *auto*
then show *?case* **..**
qed
qed

end

14 Parity: Even and Odd for int and nat

theory *Parity*
imports *Main*
begin

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

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

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

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

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*
by *presburger*

lemma *neq-one-mod-two* [*simp*, *presburger*]:
((*x*::*int*) *mod 2* ~= *0*) = (*x mod 2* = *1*) **by** *presburger*

14.2 Behavior under integer arithmetic operations

lemma *even-times-anything*: *even* (*x*::*int*) ==> *even* (*x* * *y*)
by (*simp add*: *even-def zmod-zmult1-eq'*)

lemma *anything-times-even*: *even* (*y*::*int*) ==> *even* (*x* * *y*)
by (*simp add*: *even-def zmod-zmult1-eq*)

lemma *odd-times-odd*: *odd* (*x*::*int*) ==> *odd y* ==> *odd* (*x* * *y*)
by (*simp add*: *even-def zmod-zmult1-eq*)

lemma *even-product*[*presburger*]: *even*((*x*::*int*) $*$ *y*) = (*even x* | *even y*)
  apply (*auto simp add*: *even-times-anything anything-times-even*)
  apply (*rule ccontr*)
  apply (*auto simp add*: *odd-times-odd*)
  done

lemma *even-plus-even*: *even* (*x*::*int*) ==> *even y* ==> *even* (*x* + *y*)
  by *presburger*

lemma *even-plus-odd*: *even* (*x*::*int*) ==> *odd y* ==> *odd* (*x* + *y*)
  by *presburger*

lemma *odd-plus-even*: *odd* (*x*::*int*) ==> *even y* ==> *odd* (*x* + *y*)
  by *presburger*

lemma *odd-plus-odd*: *odd* (*x*::*int*) ==> *odd y* ==> *even* (*x* + *y*) **by** *presburger*

lemma *even-sum*[*presburger*]: *even* ((*x*::*int*) + *y*) = ((*even x* & *even y*) | (*odd x* & *odd y*))
  by *presburger*

lemma *even-neg*[*presburger*]: *even* (−(*x*::*int*)) = *even x* **by** *presburger*

lemma *even-difference*:
    even ((*x*::*int*) − *y*) = ((*even x* & *even y*) | (*odd x* & *odd y*)) **by** *presburger*

lemma *even-pow-gt-zero*:
    even (*x*::*int*) ==> *0* < *n* ==> *even* (*x*^*n*)
  by (*induct n*) (*auto simp add*: *even-product*)

lemma *odd-pow-iff*[*presburger*]: *odd* ((*x*::*int*) ^ *n*) ⟷ (*n* = *0* ∨ *odd x*)
  apply (*induct n*, *simp-all*)
  apply *presburger*
  apply (*case-tac n*, *auto*)
  apply (*simp-all add*: *even-product*)
  done

lemma *odd-pow*: *odd x* ==> *odd*((*x*::*int*)^*n*) **by** (*simp add*: *odd-pow-iff*)

lemma *even-power*[*presburger*]: *even* ((*x*::*int*)^*n*) = (*even x* & *0* < *n*)
  apply (*auto simp add*: *even-pow-gt-zero*)
  apply (*erule contrapos-pp*, *erule odd-pow*)
  apply (*erule contrapos-pp*, *simp add*: *even-def*)
  done

lemma *even-zero*[*presburger*]: *even* (*0*::*int*) **by** *presburger*

lemma *odd-one*[*presburger*]: *odd* (*1*::*int*) **by** *presburger*

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*: $even\ (x::int) ==> 2 * (x\ div\ 2) = x$
by *presburger*

lemma *two-times-odd-div-two-plus-one*: $odd\ (x::int) ==>$
$2 * (x\ div\ 2) + 1 = x$ **by** *presburger*

lemma *even-equiv-def*: $even\ (x::int) = (EX\ y.\ x = 2 * y)$ **by** *presburger*

lemma *odd-equiv-def*: $odd\ (x::int) = (EX\ y.\ x = 2 * y + 1)$ **by** *presburger*

14.4 even and odd for nats

lemma *pos-int-even-equiv-nat-even*: $0 \leq x ==> even\ x = even\ (nat\ x)$
by (*simp add*: *even-nat-def*)

lemma *even-nat-product*[*presburger*]: $even((x::nat) * y) = (even\ x \mid even\ y)$
by (*simp add*: *even-nat-def int-mult*)

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

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

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

lemma *even-nat-power*[*presburger*]: $even\ ((x::nat)\ \hat{}y) = (even\ x\ \&\ 0 < y)$
by (*simp add*: *even-nat-def int-power*)

lemma *even-nat-zero*[*presburger*]: $even\ (0::nat)$ **by** *presburger*

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*: $(x::nat) < Suc\ (Suc\ 0) ==>$
$x = 0 \mid x = Suc\ 0$ **by** *presburger*

lemma *even-nat-mod-two-eq-zero*: $even\ (x::nat) ==> x\ mod\ (Suc\ (Suc\ 0)) = 0$
by *presburger*

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

by *presburger*

lemma *even-nat-equiv-def*: *even* (*x*::*nat*) = (*x mod Suc* (*Suc 0*) = *0*)
  by *presburger*

lemma *odd-nat-equiv-def*: *odd* (*x*::*nat*) = (*x mod Suc* (*Suc 0*) = *Suc 0*)
  by *presburger*

lemma *even-nat-div-two-times-two*: *even* (*x*::*nat*) ==>
    Suc (*Suc 0*) ∗ (*x div Suc* (*Suc 0*)) = *x* **by** *presburger*

lemma *odd-nat-div-two-times-two-plus-one*: *odd* (*x*::*nat*) ==>
    Suc(*Suc* (*Suc 0*) ∗ (*x div Suc* (*Suc 0*))) = *x* **by** *presburger*

lemma *even-nat-equiv-def2*: *even* (*x*::*nat*) = (*EX y*. *x* = *Suc* (*Suc 0*) ∗ *y*)
  by *presburger*

lemma *odd-nat-equiv-def2*: *odd* (*x*::*nat*) = (*EX y*. *x* = *Suc*(*Suc* (*Suc 0*) ∗ *y*))
  by *presburger*

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*)
  apply (*induct x*)
  apply (*rule conjI*)
  apply *simp*
  apply (*insert even-nat-zero*, *blast*)
  apply (*simp add*: *power-Suc*)
  done

lemma *minus-one-even-power* [*simp*]:
    even x ==> (− *1*::′*a*::{*comm-ring-1*,*recpower*}) ˆ*x* = *1*
  using *minus-one-even-odd-power* **by** *blast*

lemma *minus-one-odd-power* [*simp*]:
    odd x ==> (− *1*::′*a*::{*comm-ring-1*,*recpower*}) ˆ*x* = − *1*
  using *minus-one-even-odd-power* **by** *blast*

lemma *neg-one-even-odd-power*:
    (*even x* −−> (−*1*::′*a*::{*number-ring*,*recpower*}) ˆ*x* = *1*) &
    (*odd x* −−> (−*1*::′*a*) ˆ*x* = −*1*)
  apply (*induct x*)
  apply (*simp*, *simp add*: *power-Suc*)
  done

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

using *neg-one-even-odd-power* **by** *blast*

lemma *neg-one-odd-power* [*simp*]:
odd x $==>$ $(-1::'a::\{number\text{-}ring,recpower\})$ ^ $x = -1$
using *neg-one-even-odd-power* **by** *blast*

lemma *neg-power-if*:
$(-x::'a::\{comm\text{-}ring\text{-}1,recpower\})$ ^ n =
(*if even n then* (x ^ n) *else* $-$(x ^ n))
apply (*induct n*)
apply (*simp-all split*: *split-if-asm add*: *power-Suc*)
done

lemma *zero-le-even-power*: *even n* $==>$
$0 <= (x::'a::\{recpower,ordered\text{-}ring\text{-}strict\})$ ^ n
apply (*simp add*: *even-nat-equiv-def2*)
apply (*erule exE*)
apply (*erule ssubst*)
apply (*subst power-add*)
apply (*rule zero-le-square*)
done

lemma *zero-le-odd-power*: *odd n* $==>$
$(0 <= (x::'a::\{recpower,ordered\text{-}idom\})$ ^ $n) = (0 <= x)$
apply (*simp add*: *odd-nat-equiv-def2*)
apply (*erule exE*)
apply (*erule ssubst*)
apply (*subst power-Suc*)
apply (*subst power-add*)
apply (*subst zero-le-mult-iff*)
apply *auto*
apply (*subgoal-tac* $x = 0$ & $y > 0$)
apply (*erule conjE*, *assumption*)
apply (*subst power-eq-0-iff* [*symmetric*])
apply (*subgoal-tac* $0 <= x$^$y * x$^y)
apply *simp*
apply (*rule zero-le-square*)+
done

lemma *zero-le-power-eq*[*presburger*]: $(0 <= (x::'a::\{recpower,ordered\text{-}idom\})$ ^ $n)$
=
(*even n* | (*odd n* & $0 <= x$))
apply *auto*
apply (*subst zero-le-odd-power* [*symmetric*])
apply *assumption+*
apply (*erule zero-le-even-power*)
apply (*subst zero-le-odd-power*)
apply *assumption+*
done

lemma *zero-less-power-eq*[*presburger*]: (*0* < (*x*::'*a*::{*recpower*,*ordered-idom*}) ^ *n*) =
  (*n* = *0* | (*even n* & *x* ~= *0*) | (*odd n* & *0* < *x*))
 apply (*rule iffI*)
 apply *clarsimp*
 apply (*rule conjI*)
 apply *clarsimp*
 apply (*rule ccontr*)
 apply (*subgoal-tac* ~ (*0* <= *x*^*n*))
 apply *simp*
 apply (*subst zero-le-odd-power*)
 apply *assumption*
 apply *simp*
 apply (*rule notI*)
 apply (*simp add*: *power-0-left*)
 apply (*rule notI*)
 apply (*simp add*: *power-0-left*)
 apply *auto*
 apply (*subgoal-tac* *0* <= *x*^*n*)
 apply (*frule order-le-imp-less-or-eq*)
 apply *simp*
 apply (*erule zero-le-even-power*)
 apply (*subgoal-tac* *0* <= *x*^*n*)
 apply (*frule order-le-imp-less-or-eq*)
 apply *auto*
 apply (*subst zero-le-odd-power*)
 apply *assumption*
 apply (*erule order-less-imp-le*)
 done

lemma *power-less-zero-eq*[*presburger*]: ((*x*::'*a*::{*recpower*,*ordered-idom*}) ^ *n* < *0*) =
  (*odd n* & *x* < *0*)
 apply (*subst linorder-not-le* [*symmetric*])+
 apply (*subst zero-le-power-eq*)
 apply *auto*
 done

lemma *power-le-zero-eq*[*presburger*]: ((*x*::'*a*::{*recpower*,*ordered-idom*}) ^ *n* <= *0*) =
  (*n* ~= *0* & ((*odd n* & *x* <= *0*) | (*even n* & *x* = *0*)))
 apply (*subst linorder-not-less* [*symmetric*])+
 apply (*subst zero-less-power-eq*)
 apply *auto*
 done

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

apply (*subst power-abs* [*symmetric*])
apply (*simp add*: *zero-le-even-power*)
done

lemma *zero-less-power-nat-eq*[*presburger*]: $(0 < (x::nat)\ \hat{}\ n) = (n = 0 \mid 0 < x)$
by (*induct n*) *auto*

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

lemma *power-minus-odd* [*simp*]: *odd n* ==>
$(-\ x)\,\hat{}n = -\ (x\hat{}n::'a::\{recpower,comm\text{-}ring\text{-}1\})$
apply (*subst power-minus*)
apply *simp*
done

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 $0 \leq a\hat{}n$

lemma *even-power-le-0-imp-0*:
$a\ \hat{}\ (2{*}k) \leq (0::'a::\{ordered\text{-}idom,recpower\})$ ==> $a{=}0$
by (*induct k*) (*auto simp add*: *zero-le-mult-iff mult-le-0-iff power-Suc*)

```
lemma zero-le-power-iff [presburger]:
  (0 ≤ a ^ n) = (0 ≤ (a::'a::{ordered-idom,recpower}) | even n)
proof cases
  assume even: even n
  then obtain k where n = 2*k
    by (auto simp add: even-nat-equiv-def2 numeral-2-eq-2)
  thus ?thesis by (simp add: zero-le-even-power even)
next
  assume odd: odd n
  then obtain k where n = Suc(2*k)
    by (auto simp add: odd-nat-equiv-def2 numeral-2-eq-2)
  thus ?thesis
    by (auto simp add: power-Suc zero-le-mult-iff zero-le-even-power
         dest!: even-power-le-0-imp-0)
qed
```

14.8 Miscellaneous

```
lemma [presburger]:(x + 1) div 2 = x div 2 ⟷ even (x::int) by presburger
lemma [presburger]: (x + 1) div 2 = x div 2 + 1 ⟷ odd (x::int) by presburger
lemma even-plus-one-div-two: even (x::int) ==> (x + 1) div 2 = x div 2 by
presburger
lemma odd-plus-one-div-two: odd (x::int) ==> (x + 1) div 2 = x div 2 + 1 by
presburger

lemma div-Suc: Suc a div c = a div c + Suc 0 div c +
    (a mod c + Suc 0 mod c) div c
  apply (subgoal-tac Suc a = a + Suc 0)
  apply (erule ssubst)
  apply (rule div-add1-eq, simp)
  done

lemma [presburger]: (Suc x) div Suc (Suc 0) = x div Suc (Suc 0) ⟷ even x by
presburger
lemma [presburger]: (Suc x) div Suc (Suc 0) = x div Suc (Suc 0) ⟷ even x by
presburger
lemma even-nat-plus-one-div-two: even (x::nat) ==>
    (Suc x) div Suc (Suc 0) = x div Suc (Suc 0) by presburger

lemma odd-nat-plus-one-div-two: odd (x::nat) ==>
    (Suc x) div Suc (Suc 0) = Suc (x div Suc (Suc 0)) by presburger

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)))
by pat-completeness auto
termination by (relation measure (λ(x, y). size x + size y)) auto

function
  mul :: 'a::{comm-ring,recpower} pol ⇒ 'a pol ⇒ 'a pol (infixl ⊗ 70)
where
    Pc a ⊗ Pc b = Pc (a ∗ b)
  | Pc c ⊗ Pinj i P =
      (if c = 0 then Pc 0 else mkPinj i (P ⊗ Pc c))
  | Pinj i P ⊗ Pc c =
      (if c = 0 then Pc 0 else mkPinj i (P ⊗ Pc c))
  | Pc c ⊗ PX P i Q =
      (if c = 0 then Pc 0 else mkPX (P ⊗ Pc c) i (Q ⊗ Pc c))
  | PX P i Q ⊗ Pc c =
      (if c = 0 then Pc 0 else mkPX (P ⊗ Pc c) i (Q ⊗ Pc c))
  | Pinj x P ⊗ Pinj y Q =
      (if x = y then mkPinj x (P ⊗ Q) else
```

```
      (if x > y then mkPinj y (Pinj (x−y) P ⊗ Q)
       else mkPinj x (Pinj (y − x) Q ⊗ P)))
  | Pinj x P ⊗ PX Q y R =
     (if x = 0 then P ⊗ PX Q y R else
       (if x = 1 then mkPX (Pinj x P ⊗ Q) y (R ⊗ P)
        else mkPX (Pinj x P ⊗ Q) y (R ⊗ Pinj (x − 1) P)))
  | PX P x R ⊗ Pinj y Q =
     (if y = 0 then PX P x R ⊗ Q else
       (if y = 1 then mkPX (Pinj y Q ⊗ P) x (R ⊗ Q)
        else mkPX (Pinj y Q ⊗ P) x (R ⊗ Pinj (y − 1) Q)))
  | PX P1 x P2 ⊗ PX Q1 y Q2 =
     mkPX (P1 ⊗ Q1) (x + y) (P2 ⊗ Q2) ⊕
      (mkPX (P1 ⊗ mkPinj 1 Q2) x (Pc 0) ⊕
       (mkPX (Q1 ⊗ mkPinj 1 P2) y (Pc 0)))
by pat-completeness auto
termination by (relation measure (λ(x, y). size x + size y))
  (auto simp add: mkPinj-def split: pol.split)
```

Negation

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

Substraction

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

Square for Fast Exponentation

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

Fast Exponentation

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

lemma *pow-if*:
  pow n P =
   (*if n = 0 then Pc 1 else if even n then pow* (*n div 2*) (*sqr P*)
    else P ⊗ *pow* (*n div 2*) (*sqr P*))
  by (*cases n*) *simp-all*

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*)
  by (*induct B*) (*auto simp add*: *mkPinj-def ring-simps*)

mkPX preserves semantics

lemma *mkPX-ci*: *Ipol l* (*mkPX A b C*) = *Ipol l* (*PX A b C*)
  by (*cases A*) (*auto simp add*: *mkPX-def mkPinj-ci power-add ring-simps*)

Correctness theorems for the implemented operations

Negation

lemma *neg-ci*: *Ipol l* (*neg P*) = −(*Ipol l P*)
  by (*induct P arbitrary*: *l*) *auto*

Addition

lemma *add-ci*: *Ipol l* (*P* ⊕ *Q*) = *Ipol l P* + *Ipol l Q*
proof (*induct P Q arbitrary*: *l rule*: *add.induct*)
  case (*6 x P y Q*)
  show *?case*
  proof (*rule linorder-cases*)
    assume $x < y$
    with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci ring-simps*)
  next
    assume $x = y$
    with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci*)
  next
    assume $x > y$
    with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci ring-simps*)
  qed
next
  case (*7 x P Q y R*)
  have $x = 0 \lor x = 1 \lor x > 1$ **by** *arith*

moreover
{ assume $x = 0$ **with** *7* **have** *?case* **by** *simp* **}**
moreover
{ assume $x = 1$ **with** *7* **have** *?case* **by** (*simp add*: *ring-simps*) **}**
moreover
{ assume $x > 1$ **from** *7* **have** *?case* **by** (*cases x*) *simp-all* **}**
ultimately show *?case* **by** *blast*
next
case (*8 P x R y Q*)
have $y = 0 \lor y = 1 \lor y > 1$ **by** *arith*
moreover
{ assume $y = 0$ **with** *8* **have** *?case* **by** *simp* **}**
moreover
{ assume $y = 1$ **with** *8* **have** *?case* **by** *simp* **}**
moreover
{ assume $y > 1$ **with** *8* **have** *?case* **by** *simp* **}**
ultimately show *?case* **by** *blast*
next
case (*9 P1 x P2 Q1 y Q2*)
show *?case*
proof (*rule linorder-cases*)
assume *a*: $x < y$ **hence** *EX d.* $d + x = y$ **by** *arith*
with *9 a* **show** *?case* **by** (*auto simp add*: *mkPX-ci power-add ring-simps*)
next
assume *a*: $y < x$ **hence** *EX d.* $d + y = x$ **by** *arith*
with *9 a* **show** *?case* **by** (*auto simp add*: *power-add mkPX-ci ring-simps*)
next
assume $x = y$
with *9* **show** *?case* **by** (*simp add*: *mkPX-ci ring-simps*)
qed
qed (*auto simp add*: *ring-simps*)

Multiplication

lemma *mul-ci*: *Ipol l* $(P \otimes Q)$ = *Ipol l P* $*$ *Ipol l Q*
by (*induct P Q arbitrary*: *l rule*: *mul.induct*)
(*simp-all add*: *mkPX-ci mkPinj-ci ring-simps add-ci power-add*)

Substraction

lemma *sub-ci*: *Ipol l* $(P \ominus Q)$ = *Ipol l P* $-$ *Ipol l Q*
by (*simp add*: *add-ci neg-ci sub-def*)

Square

lemma *sqr-ci*: *Ipol ls* (*sqr P*) = *Ipol ls P* $*$ *Ipol ls P*
by (*induct P arbitrary*: *ls*)
(*simp-all add*: *add-ci mkPinj-ci mkPX-ci mul-ci ring-simps power-add*)

Power

lemma *even-pow*:*even n* $\Longrightarrow$ *pow n P* = *pow* (*n div 2*) (*sqr P*)
by (*induct n*) *simp-all*

```
lemma pow-ci: Ipol ls (pow n P) = Ipol ls P ˆ n
proof (induct n arbitrary: P rule: nat-less-induct)
  case (1 k)
  show ?case
  proof (cases k)
    case 0
    then show ?thesis by simp
  next
    case (Suc l)
    show ?thesis
    proof cases
      assume even l
      then have Suc l div 2 = l div 2
        by (simp add: nat-number even-nat-plus-one-div-two)
      moreover
      from Suc have l < k by simp
      with 1 have ⋀P. Ipol ls (pow l P) = Ipol ls P ˆ l by simp
      moreover
      note Suc ‹even l› even-nat-plus-one-div-two
      ultimately show ?thesis by (auto simp add: mul-ci power-Suc even-pow)
    next
      assume odd l
      {
        fix p
        have Ipol ls (sqr P) ˆ (Suc l div 2) = Ipol ls P ˆ Suc l
        proof (cases l)
          case 0
          with ‹odd l› show ?thesis by simp
        next
          case (Suc w)
          with ‹odd l› have even w by simp
          have two-times: 2 ∗ (w div 2) = w
            by (simp only: numerals even-nat-div-two-times-two [OF ‹even w›])
          have Ipol ls P ∗ Ipol ls P = Ipol ls P ˆ Suc (Suc 0)
            by (simp add: power-Suc)
          then have Ipol ls P ∗ Ipol ls P = Ipol ls P ˆ 2
            by (simp add: numerals)
          with Suc show ?thesis
             by (auto simp add: power-mult [symmetric, of - 2 -] two-times mul-ci
sqr-ci)
        qed
      } with 1 Suc ‹odd l› show ?thesis by simp
    qed
  qed
qed
```

Normalization preserves semantics

```
lemma norm-ci: Ipolex l Pe = Ipol l (norm Pe)
```

by (*induct Pe*) (*simp-all add*: *add-ci sub-ci mul-ci neg-ci pow-ci*)

Reflection lemma: Key to the (incomplete) decision procedure

lemma *norm-eq*:
assumes *norm P1* = *norm P2*
shows *Ipolex l P1* = *Ipolex l P2*
proof −
from *prems* **have** *Ipol l* (*norm P1*) = *Ipol l* (*norm P2*) **by** *simp*
then show *?thesis* **by** (*simp only*: *norm-ci*)
qed

use *comm-ring.ML*
setup *CommRing.setup*

end

16 Continuity: Continuity and iterations (of set transformers)

theory *Continuity*
imports *Main*
begin

16.1 Continuity for complete lattices

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

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

lemma *SUP-nat-conv*:
(*SUP n*. *M n*) = *sup* (*M 0*) (*SUP n*. *M*(*Suc n*))
apply(*rule order-antisym*)
apply(*rule SUP-leI*)
apply(*case-tac n*)
apply *simp*
apply (*fast intro*:*le-SUPI le-supI2*)
apply(*simp*)
apply (*blast intro*:*SUP-leI le-SUPI*)
done

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

```
proof
  fix A B :: 'a assume A <= B
  let ?C = %i::nat. if i=0 then A else B
  have chain ?C using ‹A <= B› by(simp add:chain-def)
  have F B = sup (F A) (F B)
  proof −
    have sup A B = B using ‹A <= B› by (simp add:sup-absorb2)
    hence F B = F(SUP i. ?C i) by (subst SUP-nat-conv) simp
    also have ... = (SUP i. F(?C i))
      using ‹chain ?C› ‹continuous F› by(simp add:continuous-def)
    also have ... = sup (F A) (F B) by (subst SUP-nat-conv) simp
    finally show ?thesis .
  qed
  thus F A ≤ F B by(subst le-iff-sup, simp)
qed

lemma continuous-lfp:
 assumes continuous F shows lfp F = (SUP i. (F^i) bot)
proof −
  note mono = continuous-mono[OF ‹continuous F›]
  { fix i have (F^i) bot ≤ lfp F
    proof (induct i)
      show (F^0) bot ≤ lfp F by simp
    next
      case (Suc i)
      have (F^(Suc i)) bot = F((F^i) bot) by simp
      also have ... ≤ F(lfp F) by(rule monoD[OF mono Suc])
      also have ... = lfp F by(simp add:lfp-unfold[OF mono, symmetric])
      finally show ?case .
    qed }
  hence (SUP i. (F^i) bot) ≤ lfp F by (blast intro!:SUP-leI)
  moreover have lfp F ≤ (SUP i. (F^i) bot) (is - ≤ ?U)
  proof (rule lfp-lowerbound)
    have chain(%i. (F^i) bot)
    proof −
      { fix i have (F^i) bot ≤ (F^(Suc i)) bot
        proof (induct i)
          case 0 show ?case by simp
        next
          case Suc thus ?case using monoD[OF mono Suc] by auto
        qed }
      thus ?thesis by(auto simp add:chain-def)
    qed
      hence F ?U = (SUP i. (F^(i+1)) bot) using ‹continuous F› by (simp
add:continuous-def)
    also have ... ≤ ?U by(fast intro:SUP-leI le-SUPI)
    finally show F ?U ≤ ?U .
  qed
  ultimately show ?thesis by (blast intro:order-antisym)
```

qed

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 = ($\forall$ *i*. *F i* $\subseteq$ *F* (*Suc i*))

lemma *up-chainI*: (!!*i*. *F i* $\subseteq$ *F* (*Suc i*)) ==> *up-chain F*
  by (*simp add*: *up-chain-def*)

lemma *up-chainD*: *up-chain F* ==> *F i* $\subseteq$ *F* (*Suc i*)
  by (*simp add*: *up-chain-def*)

lemma *up-chain-less-mono*:
    up-chain F ==> *x* < *y* ==> *F x* $\subseteq$ *F y*
  apply (*induct y*)
   apply (*blast dest*: *up-chainD elim*: *less-SucE*)+
  done

lemma *up-chain-mono*: *up-chain F* ==> *x* $\leq$ *y* ==> *F x* $\subseteq$ *F y*
  apply (*drule le-imp-less-or-eq*)
  apply (*blast dest*: *up-chain-less-mono*)
  done

definition
  down-chain :: (*nat* => '*a set*) => *bool* **where**
  down-chain F = ($\forall$ *i*. *F* (*Suc i*) $\subseteq$ *F i*)

lemma *down-chainI*: (!!*i*. *F* (*Suc i*) $\subseteq$ *F i*) ==> *down-chain F*
  by (*simp add*: *down-chain-def*)

lemma *down-chainD*: *down-chain F* ==> *F* (*Suc i*) $\subseteq$ *F i*
  by (*simp add*: *down-chain-def*)

lemma *down-chain-less-mono*:
    down-chain F ==> *x* < *y* ==> *F y* $\subseteq$ *F x*
  apply (*induct y*)
   apply (*blast dest*: *down-chainD elim*: *less-SucE*)+
  done

lemma *down-chain-mono*: *down-chain F* ==> *x* $\leq$ *y* ==> *F y* $\subseteq$ *F x*
  apply (*drule le-imp-less-or-eq*)
  apply (*blast dest*: *down-chain-less-mono*)
  done

16.3 Continuity

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

lemma *up-contI*:
(!!*F*. *up-chain F* ==> *f* (⋃(*range F*)) = ⋃(*f* ' *range F*)) ==> *up-cont f*
apply (*unfold up-cont-def*)
apply *blast*
done

lemma *up-contD*:
up-cont f ==> *up-chain F* ==> *f* (⋃(*range F*)) = ⋃(*f* ' *range F*)
apply (*unfold up-cont-def*)
apply *auto*
done

lemma *up-cont-mono*: *up-cont f* ==> *mono f*
apply (*rule monoI*)
apply (*drule-tac F* = λi. *if i* = *0 then x else y* **in** *up-contD*)
apply (*rule up-chainI*)
apply *simp*
apply (*drule Un-absorb1*)
apply (*auto simp add*: *nat-not-singleton*)
done

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

lemma *down-contI*:
(!!*F*. *down-chain F* ==> *f* (*Inter* (*range F*)) = *Inter* (*f* ' *range F*)) ==>
down-cont f
apply (*unfold down-cont-def*)
apply *blast*
done

lemma *down-contD*: *down-cont f* ==> *down-chain F* ==>
f (*Inter* (*range F*)) = *Inter* (*f* ' *range F*)
apply (*unfold down-cont-def*)
apply *auto*
done

lemma *down-cont-mono*: *down-cont f* ==> *mono f*
apply (*rule monoI*)
apply (*drule-tac F* = λi. *if i* = *0 then y else x* **in** *down-contD*)

```
 apply (rule down-chainI)
 apply simp
apply (drule Int-absorb1)
apply auto
apply (auto simp add: nat-not-singleton)
done
```

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 = {}
  by (simp add: up-iterate-def)

lemma up-iterate-Suc [simp]: up-iterate f (Suc i) = f (up-iterate f i)
  by (simp add: up-iterate-def)

lemma up-iterate-chain: mono F ==> up-chain (up-iterate F)
  apply (rule up-chainI)
  apply (induct-tac i)
   apply simp+
  apply (erule (1) monoD)
  done

lemma UNION-up-iterate-is-fp:
  up-cont F ==>
    F (UNION UNIV (up-iterate F)) = UNION UNIV (up-iterate F)
  apply (frule up-cont-mono [THEN up-iterate-chain])
  apply (drule (1) up-contD)
  apply simp
  apply (auto simp del: up-iterate-Suc simp add: up-iterate-Suc [symmetric])
  apply (case-tac xa)
   apply auto
  done

lemma UNION-up-iterate-lowerbound:
    mono F ==> F P = P ==> UNION UNIV (up-iterate F) ⊆ P
  apply (subgoal-tac (!!i. up-iterate F i ⊆ P))
   apply fast
  apply (induct-tac i)
  prefer 2 apply (drule (1) monoD)
   apply auto
  done

lemma UNION-up-iterate-is-lfp:
    up-cont F ==> lfp F = UNION UNIV (up-iterate F)
  apply (rule set-eq-subset [THEN iffD2])
```

```
 apply (rule conjI)
  prefer 2
  apply (drule up-cont-mono)
  apply (rule UNION-up-iterate-lowerbound)
   apply assumption
  apply (erule lfp-unfold [symmetric])
 apply (rule lfp-lowerbound)
 apply (rule set-eq-subset [THEN iffD1, THEN conjunct2])
 apply (erule UNION-up-iterate-is-fp [symmetric])
 done


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
 by (simp add: down-iterate-def)

lemma down-iterate-Suc [simp]:
   down-iterate f (Suc i) = f (down-iterate f i)
 by (simp add: down-iterate-def)

lemma down-iterate-chain: mono F ==> down-chain (down-iterate F)
 apply (rule down-chainI)
 apply (induct-tac i)
  apply simp+
 apply (erule (1) monoD)
 done

lemma INTER-down-iterate-is-fp:
 down-cont F ==>
   F (INTER UNIV (down-iterate F)) = INTER UNIV (down-iterate F)
 apply (frule down-cont-mono [THEN down-iterate-chain])
 apply (drule (1) down-contD)
 apply simp
 apply (auto simp del: down-iterate-Suc simp add: down-iterate-Suc [symmetric])
 apply (case-tac xa)
  apply auto
 done

lemma INTER-down-iterate-upperbound:
   mono F ==> F P = P ==> P ⊆ INTER UNIV (down-iterate F)
 apply (subgoal-tac (!!i. P ⊆ down-iterate F i))
  apply fast
 apply (induct-tac i)
 prefer 2 apply (drule (1) monoD)
  apply auto
 done
```

lemma *INTER-down-iterate-is-gfp*:
  down-cont F ==> gfp F = INTER UNIV (down-iterate F)
 apply (*rule set-eq-subset* [*THEN iffD2*])
 apply (*rule conjI*)
  apply (*drule down-cont-mono*)
  apply (*rule INTER-down-iterate-upperbound*)
   apply *assumption*
  apply (*erule gfp-unfold* [*symmetric*])
 apply (*rule gfp-upperbound*)
 apply (*rule set-eq-subset* [*THEN iffD1*, *THEN conjunct2*])
 apply (*erule INTER-down-iterate-is-fp*)
 done

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* −)

setup ⟪
 fold (*fn target => CodeTarget.add-pretty-numeral target true*
  @{*const-name number-int-inst.number-of-int*}
  @{*const-name Numeral.B0*} @{*const-name Numeral.B1*}
  @{*const-name Numeral.Pls*} @{*const-name Numeral.Min*}
  @{*const-name Numeral.Bit*}
) [*SML*, *OCaml*, *Haskell*]
⟫

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
unfolding *int-of-nat-def* **by** *simp*

lemma *int-of-nat-add*:
int-of-nat $(m + n)$ = *int-of-nat* m + *int-of-nat* n
unfolding *int-of-nat-def* **by** (*rule of-nat-add*)

lemma *int-of-nat-mult*:
  int-of-nat $(m * n)$ = *int-of-nat* m * *int-of-nat* n
  unfolding *int-of-nat-def* **by** (*rule of-nat-mult*)

lemma *nat-of-int-of-number-of*:
  fixes k
  assumes $k \geq 0$
  shows *number-of* k = *nat-of-int* (*number-of* k)
  unfolding *nat-of-int-def* [*OF assms*] *nat-number-of-def number-of-is-id* **..**

lemma *nat-of-int-of-number-of-aux*:
  fixes k
  assumes *Numeral.Pls* $\leq k \equiv$ *True*
  shows $k \geq 0$
  using *assms* **unfolding** *Pls-def* **by** *simp*

lemma *nat-of-int-int*:
  nat-of-int (*int-of-nat* n) = n
  using *nat-of-int-def int-of-nat-def* **by** *simp*

lemma *eq-nat-of-int*: *int-of-nat* $n = x \Longrightarrow n$ = *nat-of-int* x
by (*erule subst*, *simp only*: *nat-of-int-int*)

code-datatype *nat-of-int*

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

lemma [*code unfold*, *code inline del*]:
  nat-case $\equiv$ ($\lambda f\ g\ n$. *if* $n = 0$ *then* f *else* g $(n - 1)$)
proof −
  have *rewrite*: $\bigwedge f\ g\ n$. *nat-case* $f\ g\ n$ = (*if* $n = 0$ *then* f *else* g $(n - 1)$)
  proof −
    fix $f\ g\ n$
    show *nat-case* $f\ g\ n$ = (*if* $n = 0$ *then* f *else* g $(n - 1)$)
      by (*cases* n) *simp-all*
  qed
  show *nat-case* $\equiv$ ($\lambda f\ g\ n$. *if* $n = 0$ *then* f *else* g $(n - 1)$)
    by (*rule eq-reflection ext rewrite*)+
qed

lemma [*code inline*]:
  nat-case = ($\lambda f\ g\ n$. *if* $n = 0$ *then* f *else* g (*nat-of-int* (*int-of-nat* $n - 1$)))
proof (*rule ext*)+
  fix $f\ g\ n$
  show *nat-case* $f\ g\ n$ = (*if* $n = 0$ *then* f *else* g (*nat-of-int* (*int-of-nat* $n - 1$)))
  by (*cases* n) (*simp-all add*: *nat-of-int-int*)
qed

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

lemma [*code func*]: *0 = nat-of-int 0*
by (*simp add*: *nat-of-int-def*)

lemma [*code func*, *code inline*]: *1 = nat-of-int 1*
by (*simp add*: *nat-of-int-def*)

lemma [*code func*]: *Suc n = nat-of-int* (*int-of-nat n + 1*)
by (*simp add*: *eq-nat-of-int*)

lemma [*code*]: *m + n = nat* (*int-of-nat m + int-of-nat n*)
by (*simp add*: *int-of-nat-def nat-eq-iff2*)

lemma [*code func*, *code inline*]: *m + n = nat-of-int* (*int-of-nat m + int-of-nat n*)
by (*simp add*: *eq-nat-of-int int-of-nat-add*)

lemma [*code*, *code inline*]: *m − n = nat* (*int-of-nat m − int-of-nat n*)
by (*simp add*: *int-of-nat-def nat-eq-iff2 of-nat-diff*)

lemma [*code*]: *m ∗ n = nat* (*int-of-nat m ∗ int-of-nat n*)
unfolding *int-of-nat-def*
by (*simp add*: *of-nat-mult* [*symmetric*] *del*: *of-nat-mult*)

lemma [*code func*, *code inline*]: *m ∗ n = nat-of-int* (*int-of-nat m ∗ int-of-nat n*)
by (*simp add*: *eq-nat-of-int int-of-nat-mult*)

lemma [*code*]: *m div n = nat* (*int-of-nat m div int-of-nat n*)
unfolding *int-of-nat-def zdiv-int* [*symmetric*] **by** *simp*

lemma *div-nat-code* [*code func*]:
m div k = nat-of-int (*fst* (*divAlg* (*int-of-nat m*, *int-of-nat k*)))
unfolding *div-def* [*symmetric*] *int-of-nat-def zdiv-int* [*symmetric*]
unfolding *int-of-nat-def* [*symmetric*] *nat-of-int-int* **..**

lemma [*code*]: *m mod n = nat* (*int-of-nat m mod int-of-nat n*)
unfolding *int-of-nat-def zmod-int* [*symmetric*] **by** *simp*

lemma *mod-nat-code* [*code func*]:
m mod k = nat-of-int (*snd* (*divAlg* (*int-of-nat m*, *int-of-nat k*)))
unfolding *mod-def* [*symmetric*] *int-of-nat-def zmod-int* [*symmetric*]
unfolding *int-of-nat-def* [*symmetric*] *nat-of-int-int* **..**

lemma [*code*, *code inline*]: (*m < n*) ⟷ (*int-of-nat m < int-of-nat n*)
unfolding *int-of-nat-def* **by** *simp*

lemma [*code func*, *code inline*]: (*m ≤ n*) ⟷ (*int-of-nat m ≤ int-of-nat n*)
unfolding *int-of-nat-def* **by** *simp*

lemma [*code func*, *code inline*]: *m = n* ⟷ *int-of-nat m = int-of-nat n*
unfolding *int-of-nat-def* **by** *simp*

lemma [*code func*]: *nat k* = (*if k* < *0 then 0 else nat-of-int k*)
  by (*cases k* < *0*) (*simp*, *simp add*: *nat-of-int-def*)

lemma [*code func*]:
  int-aux n i = (*if int-of-nat n* = *0 then i else int-aux* (*nat-of-int* (*int-of-nat n* − *1*)) (*i* + *1*))
proof −
  have *0* < *n* ⟹ *int-of-nat n* = *1* + *int-of-nat* (*nat-of-int* (*int-of-nat n* − *1*))
  proof −
    assume *prem*: *n* > *0*
    then have *int-of-nat n* − *1* ≥ *0* **unfolding** *int-of-nat-def* **by** *auto*
    then have *nat-of-int* (*int-of-nat n* − *1*) = *nat* (*int-of-nat n* − *1*) **by** (*simp add*: *nat-of-int-def*)
    with *prem* **show** *int-of-nat n* = *1* + *int-of-nat* (*nat-of-int* (*int-of-nat n* − *1*)) **unfolding** *int-of-nat-def* **by** *simp*
  qed
  then show *?thesis* **unfolding** *int-aux-def int-of-nat-def* **by** *auto*
qed

lemma *index-of-nat-code* [*code func*, *code inline*]:
  index-of-nat n = *index-of-int* (*int-of-nat n*)
  unfolding *index-of-nat-def int-of-nat-def* **by** *simp*

lemma *nat-of-index-code* [*code func*, *code inline*]:
  nat-of-index k = *nat* (*int-of-index k*)
  unfolding *nat-of-index-def* **by** *simp*

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 ⟪
fun nat-of-int-of-number-of thy cts =
  let
    val simplify-less = Simplifier.rewrite
    (HOL-basic-ss addsimps (@{thms less-numeral-code} @ @{thms less-eq-numeral-code}));
    fun mk-rew (t, ty) =
      if ty = HOLogic.natT andalso 0 <= HOLogic.dest-numeral t then
        Thm.capply @{cterm (op ≤) Numeral.Pls} (Thm.cterm-of thy t)
        |> simplify-less
        |> (fn thm => @{thm nat-of-int-of-number-of-aux} OF [thm])
        |> (fn thm => @{thm nat-of-int-of-number-of} OF [thm])
        |> (fn thm => @{thm eq-reflection} OF [thm])
        |> SOME
      else NONE
  in
    fold (HOLogic.add-numerals o Thm.term-of) cts []
    |> map-filter mk-rew
  end;
⟫

setup ⟪
```

Code.add-inline-proc (*nat-of-int-of-number-of*, *nat-of-int-of-number-of*)
⟫

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*: $(\bigwedge n.\ f\ (Suc\ n) = h\ n) \Longrightarrow f\ 0 = g \Longrightarrow$
$f\ n = (if\ n = 0\ then\ g\ else\ h\ (n - 1))$
by (*case-tac n*) *simp-all*

theorem *Suc-clause*: $(\bigwedge n.\ P\ n\ (Suc\ n)) \Longrightarrow n \neq 0 \Longrightarrow P\ (n - 1)\ n$
by (*case-tac n*) *simp-all*

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.

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* **..**
instance *bool* :: *ml-equiv* **..**
instance *list* :: (*ml-equiv*) *ml-equiv* **..**

```
oracle eval-witness-oracle (term * string list) = ⟪ fn thy => fn (goal, ws) =>
let
  fun check-type T =
    if Sorts.of-sort (Sign.classes-of thy) (T, [Eval-Witness.ml-equiv])
    then T
     else error (Type  ^ quote (Sign.string-of-typ thy T) ^  not allowed for ML
witnesses)

  fun dest-exs  0 t = t
    | dest-exs n (Const (Ex, -) $ Abs (v,T,b)) =
      Abs (v, check-type T, dest-exs (n − 1) b)
    | dest-exs - - = sys-error dest-exs;
  val t = dest-exs (length ws) (HOLogic.dest-Trueprop goal);
in
  if CodePackage.satisfies thy t ws
  then goal
  else HOLogic.Trueprop $ HOLogic.true-const (*dummy*)
end
⟫
```

method-setup *eval-witness* = ⟪

```
let

fun eval-tac ws thy =
  SUBGOAL (fn (t, i) => rtac (eval-witness-oracle thy (t, ws)) i)

in
  Method.simple-args (Scan.repeat Args.name) (fn ws => fn ctxt =>
    Method.SIMPLE-METHOD' (eval-tac ws (ProofContext.theory-of ctxt)))
end
⟩⟩ Evaluation with ML witnesses
```

19.1 Toy Examples

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

lemma $\exists\, n{::}nat.\ n = 1$
apply (*eval-witness Isabelle-Eval.Suc Isabelle-Eval.Zero-nat*)
done

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

lemma $\exists\, l.\ length\ (l{::}bool\ list) = 3$
apply (*eval-witness* [*true*,*true*,*true*])
done

Multiple witnesses

lemma $\exists\, k\ l.\ length\ (k{::}bool\ list) = length\ (l{::}bool\ list)$
apply (*eval-witness* [] [])
done

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*]:
$A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
by *blast*

lemma [*code*]:
$a \in A \longleftrightarrow (\exists x{\in}A.\ x = a)$
unfolding *bex-triv-one-point1* **..**

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* $(x\#xs)\ y \longleftrightarrow y = x \vee member\ xs\ y$
unfolding *member-def* **by** (*induct xs*) *simp-all*

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-first f xs*)
declare *drop-first.simps* [*code del*]
declare *drop-first.simps* [*code target*: *List*]

declare *remove1.simps* [*code del*]
lemma [*code target*: *List*]:
  remove1 x xs = (*if member xs x then drop-first* (λ*y*. *y* = *x*) *xs else xs*)
proof (*cases member xs x*)
  case *False* **thus** *?thesis* **unfolding** *member-def* **by** (*induct xs*) *auto*
next
  case *True*
  have *remove1 x xs* = *drop-first* (λ*y*. *y* = *x*) *xs* **by** (*induct xs*) *simp-all*
  with *True* **show** *?thesis* **by** *simp*
qed

lemma *member-nil* [*simp*]:
  member [] = (λ*x*. *False*)
proof
  fix *x*
  show *member* [] *x* = *False* **unfolding** *member-def* **by** *simp*
qed

lemma *member-insertl* [*simp*]:
  x ∈ *set* (*insertl x xs*)
  unfolding *insertl-def member-def mem-iff* **by** *simp*

lemma *insertl-member* [*simp*]:
  fixes *xs x*
  assumes *member*: *member xs x*
  shows *insertl x xs* = *xs*
  using *member* **unfolding** *insertl-def* **by** *simp*

lemma *insertl-not-member* [*simp*]:
  fixes *xs x*
  assumes *member*: ¬ (*member xs x*)
  shows *insertl x xs* = *x* # *xs*
  using *member* **unfolding** *insertl-def* **by** *simp*

lemma *foldr-remove1-empty* [*simp*]:
  foldr remove1 xs [] = []
  by (*induct xs*) *simp-all*

20.2.2 Derived definitions

function *unionl* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
  unionl [] *ys* = *ys*
| *unionl xs ys* = *foldr insertl xs ys*
by *pat-completeness auto*

termination by *lexicographic-order*

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*
by *pat-completeness auto*
termination by *lexicographic-order*

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*
by *pat-completeness auto*
termination by *lexicographic-order*

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

function *map-distinct* :: ($'a$ ⇒ $'b$) ⇒ $'a$ *list* ⇒ $'b$ *list*
where
  map-distinct f [] = []
| *map-distinct f xs* = *foldr* (*insertl o f*) *xs* []
by *pat-completeness auto*
termination by *lexicographic-order*

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

function *unions* :: $'a$ *list list* ⇒ $'a$ *list*
where
  unions [] = []
| *unions xs* = *foldr unionl xs* []
by *pat-completeness auto*
termination by *lexicographic-order*

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

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

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

definition
map-inter :: $'a$ *list* $\Rightarrow$ $('a \Rightarrow 'b$ *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*
unfolding *member-def mem-iff* **..**

lemma *iso-insert*:
set (*insertl x xs*) = *insert x* (*set xs*)
unfolding *insertl-def iso-member* **by** (*simp add*: *Set.insert-absorb*)

lemma *iso-remove1*:
assumes *distnct*: *distinct xs*
shows *set* (*remove1 x xs*) = *set xs* $-$ $\{x\}$
using *distnct set-remove1-eq* **by** *auto*

lemma *iso-union*:
set (*unionl xs ys*) = *set xs* $\cup$ *set ys*
unfolding *unionl-def*
by (*induct xs arbitrary*: *ys*) (*simp-all add*: *iso-insert*)

lemma *iso-intersect*:
set (*intersect xs ys*) = *set xs* $\cap$ *set ys*
unfolding *intersect-def Int-def* **by** (*simp add*: *Int-def iso-member*) *auto*

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*)
unfolding *subtract'-def flip-def subtract-def*
using *distnct* **by** (*induct xs arbitrary*: *ys*) *auto*

lemma *iso-map-distinct*:
set (*map-distinct f xs*) = *image f* (*set xs*)
unfolding *map-distinct-def* **by** (*induct xs*) (*simp-all add*: *iso-insert*)

lemma *iso-unions*:
set (*unions xss*) = $\bigcup$ *set* (*map set xss*)
unfolding *unions-def*
proof (*induct xss*)

case *Nil* **show** *?case* **by** *simp*
next
case (*Cons xs xss*) **thus** *?case* **by** (*induct xs*) (*simp-all add*: *iso-insert*)
qed

lemma *iso-intersects*:
set (*intersects* (*xs#xss*)) = $\bigcap$ *set* (*map set* (*xs#xss*))
by (*induct xss*) (*simp-all add*: *Int-def iso-member*, *auto*)

lemma *iso-UNION*:
set (*map-union xs f*) = *UNION* (*set xs*) (*set o f*)
unfolding *map-union-def iso-unions* **by** *simp*

lemma *iso-INTER*:
set (*map-inter* (*x#xs*) *f*) = *INTER* (*set* (*x#xs*)) (*set o f*)
unfolding *map-inter-def iso-intersects* **by** (*induct xs*) (*simp-all add*: *iso-member*, *auto*)

definition
Blall :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Blall = *flip list-all*
definition
Blex :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Blex = *flip list-ex*

lemma *iso-Ball*:
Blall xs f = *Ball* (*set xs*) *f*
unfolding *Blall-def flip-def* **by** (*induct xs*) *simp-all*

lemma *iso-Bex*:
Blex xs f = *Bex* (*set xs*) *f*
unfolding *Blex-def flip-def* **by** (*induct xs*) *simp-all*

lemma *iso-filter*:
set (*filter P xs*) = *filter-set P* (*set xs*)
unfolding *filter-set-def* **by** (*induct xs*) *auto*

20.4 code generator setup

ML ⟪
nonfix inter;
nonfix union;
nonfix subset;
⟫

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

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

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

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

21.1 Basic Properties of *Pi*

lemma *Pi-I*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B x*) ==> *f* ∈ *Pi A B*
by (*simp add*: *Pi-def*)

lemma *funcsetI*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B*) ==> *f* ∈ *A* −> *B*
by (*simp add*: *Pi-def*)

lemma *Pi-mem*: [|*f*: *Pi A B*; *x* ∈ *A*|] ==> *f x* ∈ *B x*
by (*simp add*: *Pi-def*)

lemma *funcset-mem*: [|*f* ∈ *A* −> *B*; *x* ∈ *A*|] ==> *f x* ∈ *B*
by (*simp add*: *Pi-def*)

lemma *funcset-image*: *f* ∈ *A*→*B* ==> *f* ‘ *A* ⊆ *B*
by (*auto simp add*: *Pi-def*)

lemma *Pi-eq-empty*: ((*PI x*: *A*. *B x*) = {}) = (∃ *x*∈*A*. *B*(*x*) = {})
apply (*simp add*: *Pi-def*, *auto*)

Converse direction requires Axiom of Choice to exhibit a function picking an element from each non-empty *B x*

apply (*drule-tac x* = %*u*. *SOME y*. *y* ∈ *B u* **in** *spec*, *auto*)

apply (*cut-tac P*= %*y*. *y* ∈ *B x* **in** *some-eq-ex*, *auto*)
done

lemma *Pi-empty* [*simp*]: *Pi* {} *B* = *UNIV*
  by (*simp add*: *Pi-def*)

lemma *Pi-UNIV* [*simp*]: *A* −> *UNIV* = *UNIV*
  by (*simp add*: *Pi-def*)

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: (!!*x*. *x* ∈ *A* ==> *B x* <= *C x*) ==> *Pi A B* <= *Pi A C*
  by (*simp add*: *Pi-def*, *blast*)

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: *A′* <= *A* ==> *Pi A B* <= *Pi A′ B*
  by (*simp add*: *Pi-def*, *blast*)

21.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*:
    [| *f* ∈ *A* −> *B*; *g* ∈ *B* −> *C* |]==> *compose A g f* ∈ *A* −> *C*
  by (*simp add*: *Pi-def compose-def restrict-def*)

lemma *compose-assoc*:
    [| *f* ∈ *A* −> *B*; *g* ∈ *B* −> *C*; *h* ∈ *C* −> *D* |]
      ==> *compose A h* (*compose A g f*) = *compose A* (*compose B h g*) *f*
  by (*simp add*: *expand-fun-eq Pi-def compose-def restrict-def*)

lemma *compose-eq*: *x* ∈ *A* ==> *compose A g f x* = *g*(*f*(*x*))
  by (*simp add*: *compose-def restrict-def*)

lemma *surj-compose*: [| *f* ‘ *A* = *B*; *g* ‘ *B* = *C* |] ==> *compose A g f* ‘ *A* = *C*
  by (*auto simp add*: *image-def compose-eq*)

21.3 Bounded Abstraction: *restrict*

lemma *restrict-in-funcset*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B*) ==> (λ*x*∈*A*. *f x*) ∈ *A* −>
B
  by (*simp add*: *Pi-def restrict-def*)

lemma *restrictI*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B x*) ==> (λ*x*∈*A*. *f x*) ∈ *Pi A B*
  by (*simp add*: *Pi-def restrict-def*)

lemma *restrict-apply* [*simp*]:
    (λ*y*∈*A*. *f y*) *x* = (*if x* ∈ *A then f x else arbitrary*)
  by (*simp add*: *restrict-def*)

lemma *restrict-ext*:
    (!!*x*. *x* ∈ *A* ==> *f x* = *g x*) ==> (λ*x*∈*A*. *f x*) = (λ*x*∈*A*. *g x*)
  by (*simp add*: *expand-fun-eq Pi-def Pi-def restrict-def*)

lemma *inj-on-restrict-eq* [*simp*]: *inj-on* (*restrict f A*) *A* = *inj-on f A*
by (*simp add*: *inj-on-def restrict-def*)

lemma *Id-compose*:
$[|f \in A \;{-}{>}\; B;\;\; f \in \mathit{extensional}\ A|] \;{=}{=}{>}\; \mathit{compose}\ A\ (\lambda y{\in}B.\ y)\ f = f$
by (*auto simp add*: *expand-fun-eq compose-def extensional-def Pi-def*)

lemma *compose-Id*:
$[|g \in A \;{-}{>}\; B;\;\; g \in \mathit{extensional}\ A|] \;{=}{=}{>}\; \mathit{compose}\ A\ g\ (\lambda x{\in}A.\ x) = g$
by (*auto simp add*: *expand-fun-eq compose-def extensional-def Pi-def*)

lemma *image-restrict-eq* [*simp*]: (*restrict f A*) ‘ *A* = *f* ‘ *A*
by (*auto simp add*: *restrict-def*)

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-betw :: [*'a* => *'b*, *'a set*, *'b set*] => *bool* **where** — bijective
bij-betw f A B = (*inj-on f A* & *f* ‘ *A* = *B*)

lemma *bij-betw-imp-inj-on*: *bij-betw f A B* $\Longrightarrow$ *inj-on f A*
by (*simp add*: *bij-betw-def*)

lemma *bij-betw-imp-funcset*: *bij-betw f A B* $\Longrightarrow f \in A \to B$
by (*auto simp add*: *bij-betw-def inj-on-Inv Pi-def*)

lemma *bij-betw-Inv*: *bij-betw f A B* $\Longrightarrow$ *bij-betw* (*Inv A f*) *B A*
apply (*auto simp add*: *bij-betw-def inj-on-Inv Inv-mem*)
apply (*simp add*: *image-compose* [*symmetric*] *o-def*)
apply (*simp add*: *image-def Inv-f-f*)
done

lemma *inj-on-compose*:
[| *bij-betw f A B*; *inj-on g B* |] ==> *inj-on* (*compose A g f*) *A*
by (*auto simp add*: *bij-betw-def inj-on-def compose-eq*)

lemma *bij-betw-compose*:
[| *bij-betw f A B*; *bij-betw g B C* |] ==> *bij-betw* (*compose A g f*) *A C*
apply (*simp add*: *bij-betw-def compose-eq inj-on-compose*)
apply (*auto simp add*: *compose-def image-def*)
done

lemma *bij-betw-restrict-eq* [*simp*]:
bij-betw (*restrict f A*) *A B* = *bij-betw f A B*
by (*simp add*: *bij-betw-def*)

21.5 Extensionality

lemma *extensional-arb*: $[|f \in extensional\ A;\ x \notin A|] ==> f\ x = arbitrary$
  by (*simp add*: *extensional-def*)

lemma *restrict-extensional* [*simp*]: $restrict\ f\ A \in extensional\ A$
  by (*simp add*: *restrict-def extensional-def*)

lemma *compose-extensional* [*simp*]: $compose\ A\ f\ g \in extensional\ A$
  by (*simp add*: *compose-def*)

lemma *extensionalityI*:
    $[|\ f \in extensional\ A;\ g \in extensional\ A;$
      $!!x.\ x{\in}A ==> f\ x = g\ x\ |] ==> f = g$
  by (*force simp add*: *expand-fun-eq extensional-def*)

lemma *Inv-funcset*: $f\ `\ A = B ==> (\lambda x{\in}B.\ Inv\ A\ f\ x) : B -> A$
  by (*unfold Inv-def*) (*fast intro*: *restrict-in-funcset someI2*)

lemma *compose-Inv-id*:
    $bij\text{-}betw\ f\ A\ B ==> compose\ A\ (\lambda y{\in}B.\ Inv\ A\ f\ y)\ f = (\lambda x{\in}A.\ x)$
  apply (*simp add*: *bij-betw-def compose-def*)
  apply (*rule restrict-ext*, *auto*)
  apply (*erule subst*)
  apply (*simp add*: *Inv-f-f*)
  done

lemma *compose-id-Inv*:
    $f\ `\ A = B ==> compose\ B\ f\ (\lambda y{\in}B.\ Inv\ A\ f\ y) = (\lambda x{\in}B.\ x)$
  apply (*simp add*: *compose-def*)
  apply (*rule restrict-ext*)
  apply (*simp add*: *f-Inv-f*)
  done

21.6 Cardinality

lemma *card-inj*: $[|f \in A{\to}B;\ inj\text{-}on\ f\ A;\ finite\ B|] ==> card(A) \le card(B)$
  apply (*rule card-inj-on-le*)
    apply (*auto simp add*: *Pi-def*)
  done

lemma *card-bij*:
    $[|f \in A{\to}B;\ inj\text{-}on\ f\ A;$
      $g \in B{\to}A;\ inj\text{-}on\ g\ B;\ finite\ A;\ finite\ B|] ==> card(A) = card(B)$
  by (*blast intro*: *card-inj order-antisym*)

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 ==> $S \neq \{\}$
  by *auto*

lemma *infinite-remove*:
  infinite $S \Longrightarrow$ *infinite* $(S - \{a\})$
  by *simp*

lemma *Diff-infinite-finite*:
  assumes T: *finite* T **and** S: *infinite* S
  shows *infinite* $(S - T)$
  using T
proof *induct*
  from S
  show *infinite* $(S - \{\})$ **by** *auto*
next
  fix T x
  assume *ih*: *infinite* $(S - T)$
  have $S - ($*insert* x $T) = (S - T) - \{x\}$
    by (*rule Diff-insert*)
  with *ih*

```
  show infinite (S − (insert x T))
    by (simp add: infinite-remove)
qed

lemma Un-infinite: infinite S ⟹ infinite (S ∪ T)
  by simp

lemma infinite-super:
  assumes T: S ⊆ T and S: infinite S
  shows infinite T
proof
  assume finite T
  with T have finite S by (simp add: finite-subset)
  with S show False by simp
qed
```

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 ∃k. S ⊆ {..<k}  (is ∃k. ?bounded S k)
using S
proof induct
  have ?bounded {} 0 by simp
  then show ∃k. ?bounded {} k ..
next
  fix S x
  assume ∃k. ?bounded S k
  then obtain k where k: ?bounded S k ..
  show ∃k. ?bounded (insert x S) k
  proof (cases x < k)
    case True
    with k show ?thesis by auto
  next
    case False
    with k have ?bounded S (Suc x) by auto
    then show ?thesis by auto
  qed
qed

lemma finite-nat-iff-bounded:
  finite (S::nat set) = (∃k. S ⊆ {..<k})  (is ?lhs = ?rhs)
proof
  assume ?lhs
  then show ?rhs by (rule finite-nat-bounded)
next
  assume ?rhs
  then obtain k where S ⊆ {..<k} ..
  then show finite S
```

  by (*rule finite-subset*) *simp*
qed

lemma *finite-nat-iff-bounded-le*:
  finite (S::*nat set*) = ($\exists\, k.\ S \subseteq \{..k\}$) (**is** *?lhs* = *?rhs*)
proof
  assume *?lhs*
  then obtain k **where** $S \subseteq \{..{<}k\}$
    by (*blast dest*: *finite-nat-bounded*)
  then have $S \subseteq \{..k\}$ **by** *auto*
  then show *?rhs* **..**
next
  assume *?rhs*
  then obtain k **where** $S \subseteq \{..k\}$ **..**
  then show *finite S*
    by (*rule finite-subset*) *simp*
qed

lemma *infinite-nat-iff-unbounded*:
  infinite (S::*nat set*) = ($\forall\, m.\ \exists\, n.\ m{<}n \wedge n{\in}S$)
  (**is** *?lhs* = *?rhs*)
proof
  assume *?lhs*
  show *?rhs*
  proof (*rule ccontr*)
    assume ¬ *?rhs*
    then obtain m **where** m: $\forall\, n.\ m{<}n \longrightarrow n{\notin}S$ **by** *blast*
    then have $S \subseteq \{..m\}$
      by (*auto simp add*: *sym* [*OF linorder-not-less*])
    with ‹*?lhs*› **show** *False*
      by (*simp add*: *finite-nat-iff-bounded-le*)
  qed
next
  assume *?rhs*
  show *?lhs*
  proof
    assume *finite S*
    then obtain m **where** $S \subseteq \{..m\}$
      by (*auto simp add*: *finite-nat-iff-bounded-le*)
    then have $\forall\, n.\ m{<}n \longrightarrow n{\notin}S$ **by** *auto*
    with ‹*?rhs*› **show** *False* **by** *blast*
  qed
qed

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
  assume *?lhs*

```
  show ?rhs
  proof
    fix m
    from ‹?lhs› obtain n where m<n ∧ n∈S
      by (auto simp add: infinite-nat-iff-unbounded)
    then have m≤n ∧ n∈S by simp
    then show ∃ n. m ≤ n ∧ n ∈ S ..
  qed
next
  assume ?rhs
  show ?lhs
  proof (auto simp add: infinite-nat-iff-unbounded)
    fix m
    from ‹?rhs› obtain n where Suc m ≤ n ∧ n∈S
      by blast
    then have m<n ∧ n∈S by simp
    then show ∃ n. m < n ∧ n ∈ S ..
  qed
qed
```

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: ∀ m. k<m ⟶ (∃ n. m<n ∧ n∈S)
  shows infinite (S::nat set)
proof −
  {
    fix m have ∃ n. m<n ∧ n∈S
    proof (cases k<m)
      case True
      with k show ?thesis by blast
    next
      case False
      from k obtain n where Suc k < n ∧ n∈S by auto
      with False have m<n ∧ n∈S by auto
      then show ?thesis ..
    qed
  }
  then show ?thesis
    by (auto simp add: infinite-nat-iff-unbounded)
qed

lemma nat-infinite [simp]: infinite (UNIV :: nat set)
  by (auto simp add: infinite-nat-iff-unbounded)

lemma nat-not-finite [elim]: finite (UNIV::nat set) ⟹ R
  by simp
```

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S.

```
lemma range-inj-infinite:
  inj (f::nat ⇒ 'a) ⟹ infinite (range f)
proof
  assume inj f
    and  finite (range f)
  then have finite (UNIV::nat set)
    by (auto intro: finite-imageD simp del: nat-infinite)
  then show False by simp
qed

lemma int-infinite [simp]:
  shows infinite (UNIV::int set)
proof −
  from inj-int have infinite (range int) by (rule range-inj-infinite)
  moreover
  have range int ⊆ (UNIV::int set) by simp
  ultimately show infinite (UNIV::int set) by (simp add: infinite-super)
qed
```

The "only if" direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

```
lemma linorder-injI:
  assumes hyp: !!x y. x < (y::'a::linorder) ==> f x ≠ f y
  shows inj f
proof (rule inj-onI)
  fix x y
  assume f-eq: f x = f y
  show x = y
  proof (rule linorder-cases)
    assume x < y
    with hyp have f x ≠ f y by blast
    with f-eq show ?thesis by simp
  next
    assume x = y
    then show ?thesis .
  next
    assume y < x
    with hyp have f y ≠ f x by blast
    with f-eq show ?thesis by simp
  qed
qed

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

proof –
  def *Sseq* $\equiv$ *nat-rec S* ($\lambda n\ T.\ T - \{SOME\ e.\ e \in T\}$)
  def *pick* $\equiv \lambda n.$ (*SOME e. e* $\in$ *Sseq n*)
  have *Sseq-inf*: $\bigwedge n.$ *infinite* (*Sseq n*)
  proof –
    fix *n*
    show *infinite* (*Sseq n*)
    proof (*induct n*)
      from *inf* **show** *infinite* (*Sseq 0*)
        by (*simp add*: *Sseq-def*)
    next
      fix *n*
      assume *infinite* (*Sseq n*) **then show** *infinite* (*Sseq* (*Suc n*))
        by (*simp add*: *Sseq-def infinite-remove*)
    qed
  qed
  have *Sseq-S*: $\bigwedge n.$ *Sseq n* $\subseteq$ *S*
  proof –
    fix *n*
    show *Sseq n* $\subseteq$ *S*
      by (*induct n*) (*auto simp add*: *Sseq-def*)
  qed
  have *Sseq-pick*: $\bigwedge n.$ *pick n* $\in$ *Sseq n*
  proof –
    fix *n*
    show *pick n* $\in$ *Sseq n*
    proof (*unfold pick-def*, *rule someI-ex*)
      from *Sseq-inf* **have** *infinite* (*Sseq n*) **.**
      then have *Sseq n* $\neq \{\}$ **by** *auto*
      then show $\exists x.\ x \in$ *Sseq n* **by** *auto*
    qed
  qed
  with *Sseq-S* **have** *rng*: *range pick* $\subseteq$ *S*
    by *auto*
  have *pick-Sseq-gt*: $\bigwedge n\ m.$ *pick n* $\notin$ *Sseq* (*n* + *Suc m*)
  proof –
    fix *n m*
    show *pick n* $\notin$ *Sseq* (*n* + *Suc m*)
      by (*induct m*) (*auto simp add*: *Sseq-def pick-def*)
  qed
  have *pick-pick*: $\bigwedge n\ m.$ *pick n* $\neq$ *pick* (*n* + *Suc m*)
  proof –
    fix *n m*
    from *Sseq-pick* **have** *pick* (*n* + *Suc m*) $\in$ *Sseq* (*n* + *Suc m*) **.**
    moreover from *pick-Sseq-gt*
    have *pick n* $\notin$ *Sseq* (*n* + *Suc m*) **.**
    ultimately show *pick n* $\neq$ *pick* (*n* + *Suc m*)
      by *auto*
  qed

have *inj*: *inj pick*
proof (*rule linorder-injI*)
fix *i j* :: *nat*
assume $i < j$
show *pick* $i \neq$ *pick* j
proof
assume *eq*: *pick* $i =$ *pick* j
from $\langle i < j \rangle$ **obtain** k **where** $j = i + Suc\ k$
by (*auto simp add*: *less-iff-Suc-add*)
with *pick-pick* **have** *pick* $i \neq$ *pick* j **by** *simp*
with *eq* **show** *False* **by** *simp*
qed
qed
from *rng inj* **show** *?thesis* **by** *auto*
qed

lemma *infinite-iff-countable-subset*:
infinite $S = (\exists f.\ inj\ (f{::}nat \Rightarrow {'a}) \wedge range\ f \subseteq S)$
by (*auto simp add*: *infinite-countable-subset range-inj-infinite infinite-super*)

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 (*rule ccontr*)
assume $\neg$ *?thesis*
with *img* **have** *finite* $(UN\ y{:}f`A.\ f\ -`\ \{y\})$ **by** (*blast intro*: *finite-UN-I*)
moreover have $A \subseteq (UN\ y{:}f`A.\ f\ -`\ \{y\})$ **by** *auto*
moreover note *dom*
ultimately show *False* **by** (*simp add*: *infinite-super*)
qed

lemma *inf-img-fin-domE*:
assumes *finite* $(f`A)$ **and** *infinite* A
obtains y **where** $y \in f`A$ **and** *infinite* $(f\ -`\ \{y\})$
using *assms* **by** (*blast dest*: *inf-img-fin-dom*)

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

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

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

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

lemma *INF-EX*:
$(\exists_\infty x.\ P\ x) \Longrightarrow (\exists\, x.\ P\ x)$
unfolding *Inf-many-def*
proof (*rule ccontr*)
assume *inf*: *infinite* $\{x.\ P\ x\}$
assume $\neg$ *?thesis* **then have** $\{x.\ P\ x\} = \{\}$ **by** *simp*
then have *finite* $\{x.\ P\ x\}$ **by** *simp*
with *inf* **show** *False* **by** *simp*
qed

lemma *MOST-iff-finiteNeg*: $(\forall_\infty x.\ P\ x) =$ *finite* $\{x.\ \neg\ P\ x\}$
by (*simp add*: *Alm-all-def Inf-many-def*)

lemma *ALL-MOST*: $\forall\, x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$
by (*simp add*: *MOST-iff-finiteNeg*)

lemma *INF-mono*:
assumes *inf*: $\exists_\infty x.\ P\ x$ **and** *q*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\exists_\infty x.\ Q\ x$
proof –
from *inf* **have** *infinite* $\{x.\ P\ x\}$ **unfolding** *Inf-many-def* **.**
moreover from *q* **have** $\{x.\ P\ x\} \subseteq \{x.\ Q\ x\}$ **by** *auto*
ultimately show *?thesis*
by (*simp add*: *Inf-many-def infinite-super*)
qed

lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$
unfolding *Alm-all-def* **by** (*blast intro*: *INF-mono*)

lemma *INF-nat*: $(\exists_\infty n.\ P\ (n{::}nat)) = (\forall\, m.\ \exists\, n.\ m<n \wedge P\ n)$
by (*simp add*: *Inf-many-def infinite-nat-iff-unbounded*)

lemma *INF-nat-le*: $(\exists_\infty n.\ P\ (n{::}nat)) = (\forall\, m.\ \exists\, n.\ m\le n \wedge P\ n)$
by (*simp add*: *Inf-many-def infinite-nat-iff-unbounded-le*)

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

by (*simp add*: *Alm-all-def INF-nat*)

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n{::}nat)) = (\exists\, m.\ \forall\, n.\ m \le n \longrightarrow P\ n)$
by (*simp add*: *Alm-all-def INF-nat-le*)

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*
by *simp*

lemma *enumerate-in-set*: *infinite S* $\Longrightarrow$ *enumerate S n* : *S*
apply (*induct n arbitrary*: *S*)
apply (*fastsimp intro*: *LeastI dest*!: *infinite-imp-nonempty*)
apply (*fastsimp iff*: *finite-Diff-singleton*)
done

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

lemma *enumerate-step*: *infinite S* $\Longrightarrow$ *enumerate S n* $<$ *enumerate S (Suc n)*
apply (*induct n arbitrary*: *S*)
apply (*rule order-le-neq-trans*)
apply (*simp add*: *enumerate-0 Least-le enumerate-in-set*)
apply (*simp only*: *enumerate-Suc'*)
apply (*subgoal-tac enumerate* $(S - \{$*enumerate S 0*$\})$ *0* : $S - \{$*enumerate S 0*$\}$)
apply (*blast intro*: *sym*)
apply (*simp add*: *enumerate-in-set del*: *Diff-iff*)
apply (*simp add*: *enumerate-Suc'*)
done

lemma *enumerate-mono*: $m<n \Longrightarrow$ *infinite S* $\Longrightarrow$ *enumerate S m* $<$ *enumerate S n*
apply (*erule less-Suc-induct*)
apply (*auto intro*: *enumerate-step*)
done

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* ⇒ *bool* **where**
atmost-one S = (∀ *x y*. *x*∈*S* ∧ *y*∈*S* ⟶ *x*=*y*)

lemma *atmost-one-empty*: *S* = {} ⟹ *atmost-one S*
by (*simp add*: *atmost-one-def*)

lemma *atmost-one-singleton*: *S* = {*x*} ⟹ *atmost-one S*
by (*simp add*: *atmost-one-def*)

lemma *atmost-one-unique* [*elim*]: *atmost-one S* ⟹ *x* ∈ *S* ⟹ *y* ∈ *S* ⟹ *y* = *x*
by (*simp add*: *atmost-one-def*)

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
show (λ*x*. *0*::*nat*) ∈ *?multiset* **by** *simp*
qed

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*) **..**

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*
by (*simp add*: *multiset-def*)

lemma *only1-in-multiset* [*simp*]: (λ*b*. *if b* = *a then 1 else 0*) ∈ *multiset*
by (*simp add*: *multiset-def*)

lemma *union-preserves-multiset* [*simp*]:
M ∈ *multiset* ==> *N* ∈ *multiset* ==> (λ*a*. *M a* + *N a*) ∈ *multiset*
apply (*simp add*: *multiset-def*)
apply (*drule* (*1*) *finite-UnI*)
apply (*simp del*: *finite-Un add*: *Un-def*)
done

lemma *diff-preserves-multiset* [*simp*]:
M ∈ *multiset* ==> (λ*a*. *M a* − *N a*) ∈ *multiset*
apply (*simp add*: *multiset-def*)
apply (*rule finite-subset*)
apply *auto*
done

23.2 Algebraic properties of multisets

23.2.1 Union

lemma *union-empty* [*simp*]: *M* + {#} = *M* ∧ {#} + *M* = *M*
by (*simp add*: *union-def Mempty-def*)

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$
  by (*simp add*: *union-def add-ac*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
  by (*simp add*: *union-def add-ac*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
proof −
  have $M + (N + K) = (N + K) + M$
    by (*rule union-commute*)
  also have $\ldots = N + (K + M)$
    by (*rule union-assoc*)
  also have $K + M = M + K$
    by (*rule union-commute*)
  finally show *?thesis* **.**
qed

lemmas *union-ac* = *union-assoc union-commute union-lcomm*

instance *multiset* :: (*type*) *comm-monoid-add*
proof
  fix $a\ b\ c$ :: *'a multiset*
  show $(a + b) + c = a + (b + c)$ **by** (*rule union-assoc*)
  show $a + b = b + a$ **by** (*rule union-commute*)
  show $0 + a = a$ **by** *simp*
qed

23.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
  by (*simp add*: *Mempty-def diff-def*)

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
  by (*simp add*: *union-def diff-def*)

23.2.3 Count of elements

lemma *count-empty* [*simp*]: $count\ \{\#\}\ a = 0$
  by (*simp add*: *count-def Mempty-def*)

lemma *count-single* [*simp*]: $count\ \{\#b\#\}\ a = (if\ b = a\ then\ 1\ else\ 0)$
  by (*simp add*: *count-def single-def*)

lemma *count-union* [*simp*]: $count\ (M + N)\ a = count\ M\ a + count\ N\ a$
  by (*simp add*: *count-def union-def*)

lemma *count-diff* [*simp*]: $count\ (M - N)\ a = count\ M\ a - count\ N\ a$
  by (*simp add*: *count-def diff-def*)

23.2.4 Set of elements

lemma *set-of-empty* [*simp*]: *set-of* $\{\#\} = \{\}$
by (*simp add*: *set-of-def*)

lemma *set-of-single* [*simp*]: *set-of* $\{\#b\#\} = \{b\}$
by (*simp add*: *set-of-def*)

lemma *set-of-union* [*simp*]: *set-of* $(M + N) =$ *set-of* $M \cup$ *set-of* N
by (*auto simp add*: *set-of-def*)

lemma *set-of-eq-empty-iff* [*simp*]: (*set-of* $M = \{\}) = (M = \{\#\})$
by (*auto simp add*: *set-of-def Mempty-def count-def expand-fun-eq*)

lemma *mem-set-of-iff* [*simp*]: $(x \in$ *set-of* $M) = (x :\# M)$
by (*auto simp add*: *set-of-def*)

23.2.5 Size

lemma *size-empty* [*simp*]: *size* $\{\#\} = 0$
by (*simp add*: *size-def*)

lemma *size-single* [*simp*]: *size* $\{\#b\#\} = 1$
by (*simp add*: *size-def*)

lemma *finite-set-of* [*iff*]: *finite* (*set-of* M)
using *Rep-multiset* [*of* M]
by (*simp add*: *multiset-def set-of-def count-def*)

lemma *setsum-count-Int*:
finite A ==> *setsum* (*count* N) ($A \cap$ *set-of* N) = *setsum* (*count* N) A
apply (*induct rule*: *finite-induct*)
apply *simp*
apply (*simp add*: *Int-insert-left set-of-def*)
done

lemma *size-union* [*simp*]: *size* $(M + N::'a$ *multiset*$) =$ *size* $M +$ *size* N
apply (*unfold size-def*)
apply (*subgoal-tac count* $(M + N) = (\lambda a.$ *count* $M\ a +$ *count* $N\ a))$
prefer *2*
apply (*rule ext*, *simp*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-Un-nat setsum-addf setsum-count-Int*)
apply (*subst Int-commute*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-count-Int*)
done

lemma *size-eq-0-iff-empty* [*iff*]: (*size* $M = 0) = (M = \{\#\})$
apply (*unfold size-def Mempty-def count-def*, *auto*)
apply (*simp add*: *set-of-def count-def expand-fun-eq*)
done

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ==> $\exists$ *a*. *a* :# *M*
  apply (*unfold size-def*)
  apply (*drule setsum-SucD*, *auto*)
  done

23.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: (*M* = *N*) = ($\forall$ *a*. *count M a* = *count N a*)
  by (*simp add*: *count-def expand-fun-eq*)

lemma *single-not-empty* [*simp*]: {#*a*#} $\neq$ {#} $\wedge$ {#} $\neq$ {#*a*#}
  by (*simp add*: *single-def Mempty-def expand-fun-eq*)

lemma *single-eq-single* [*simp*]: ({#*a*#} = {#*b*#}) = (*a* = *b*)
  by (*auto simp add*: *single-def expand-fun-eq*)

lemma *union-eq-empty* [*iff*]: (*M* + *N* = {#}) = (*M* = {#} $\wedge$ *N* = {#})
  by (*auto simp add*: *union-def Mempty-def expand-fun-eq*)

lemma *empty-eq-union* [*iff*]: ({#} = *M* + *N*) = (*M* = {#} $\wedge$ *N* = {#})
  by (*auto simp add*: *union-def Mempty-def expand-fun-eq*)

lemma *union-right-cancel* [*simp*]: (*M* + *K* = *N* + *K*) = (*M* = (*N*::'*a multiset*))
  by (*simp add*: *union-def expand-fun-eq*)

lemma *union-left-cancel* [*simp*]: (*K* + *M* = *K* + *N*) = (*M* = (*N*::'*a multiset*))
  by (*simp add*: *union-def expand-fun-eq*)

lemma *union-is-single*:
    (*M* + *N* = {#*a*#}) = (*M* = {#*a*#} $\wedge$ *N*={#} $\vee$ *M* = {#} $\wedge$ *N* = {#*a*#})
  apply (*simp add*: *Mempty-def single-def union-def add-is-1 expand-fun-eq*)
  apply *blast*
  done

lemma *single-is-union*:
     ({#*a*#} = *M* + *N*) = ({#*a*#} = *M* $\wedge$ *N* = {#} $\vee$ *M* = {#} $\wedge$ {#*a*#} = *N*)
  apply (*unfold Mempty-def single-def union-def*)
  apply (*simp add*: *add-is-1 one-is-add expand-fun-eq*)
  apply (*blast dest*: *sym*)
  done

lemma *add-eq-conv-diff*:
  (*M* + {#*a*#} = *N* + {#*b*#}) =
    (*M* = *N* $\wedge$ *a* = *b* $\vee$ *M* = *N* − {#*a*#} + {#*b*#} $\wedge$ *N* = *M* − {#*b*#} + {#*a*#})
  using [[*simproc del*: *neq*]]
  apply (*unfold single-def union-def diff-def*)

apply (*simp* (*no-asm*) *add*: *expand-fun-eq*)
apply (*rule conjI*, *force*, *safe*, *simp-all*)
apply (*simp add*: *eq-sym-conv*)
done

declare *Rep-multiset-inject* [*symmetric*, *simp del*]

instance *multiset* :: (*type*) *cancel-ab-semigroup-add*
proof
fix $a\ b\ c$ :: *'a multiset*
show $a + b = a + c \Longrightarrow b = c$ **by** *simp*
qed

23.2.7 Intersection

lemma *multiset-inter-count*:
count (A #∩ B) x = *min* (*count* A x) (*count* B x)
by (*simp add*: *multiset-inter-def min-def*)

lemma *multiset-inter-commute*: A #∩ B = B #∩ A
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count*
min-max.inf-commute)

lemma *multiset-inter-assoc*: A #∩ (B #∩ C) = A #∩ B #∩ C
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count*
min-max.inf-assoc)

lemma *multiset-inter-left-commute*: A #∩ (B #∩ C) = B #∩ (A #∩ C)
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def*)

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *multiset-union-diff-commute*: B #∩ C = {#} $\Longrightarrow$ $A + B - C = A - C + B$
apply (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def*
split: *split-if-asm*)
apply *clarsimp*
apply (*erule-tac* $x = a$ **in** *allE*)
apply *auto*
done

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* $f\ F$)
apply (*induct rule*: *finite-induct*)

```
   apply auto
  apply (drule-tac a = a in mk-disjoint-insert, auto)
  done

lemma rep-multiset-induct-aux:
  assumes 1: P (λa. (0::nat))
    and 2: !!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))
  shows ∀f. f ∈ multiset --> setsum f {x. f x ≠ 0} = n --> P f
  apply (unfold multiset-def)
  apply (induct-tac n, simp, clarify)
   apply (subgoal-tac f = (λa.0))
    apply simp
    apply (rule 1)
   apply (rule ext, force, clarify)
  apply (frule setsum-SucD, clarify)
  apply (rename-tac a)
  apply (subgoal-tac finite {x. (f (a := f a - 1)) x > 0})
   prefer 2
   apply (rule finite-subset)
    prefer 2
    apply assumption
   apply simp
   apply blast
  apply (subgoal-tac f = (f (a := f a - 1))(a := (f (a := f a - 1)) a + 1))
   prefer 2
   apply (rule ext)
   apply (simp (no-asm-simp))
   apply (erule ssubst, rule 2 [unfolded multiset-def], blast)
  apply (erule allE, erule impE, erule-tac [2] mp, blast)
  apply (simp (no-asm-simp) add: setsum-decr del: fun-upd-apply One-nat-def)
  apply (subgoal-tac {x. x ≠ a --> f x ≠ 0} = {x. f x ≠ 0})
   prefer 2
   apply blast
  apply (subgoal-tac {x. x ≠ a ∧ f x ≠ 0} = {x. f x ≠ 0} - {a})
   prefer 2
   apply blast
  apply (simp add: le-imp-diff-is-add setsum-diff1-nat cong: conj-cong)
  done

theorem rep-multiset-induct:
  f ∈ multiset ==> P (λa. 0) ==>
    (!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))) ==> P f
  using rep-multiset-induct-aux by blast

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 -
```

```
  note defns = union-def single-def Mempty-def
  show ?thesis
    apply (rule Rep-multiset-inverse [THEN subst])
    apply (rule Rep-multiset [THEN rep-multiset-induct])
     apply (rule empty [unfolded defns])
    apply (subgoal-tac f(b := f b + 1) = (λa. f a + (if a=b then 1 else 0)))
     prefer 2
     apply (simp add: expand-fun-eq)
    apply (erule ssubst)
    apply (erule Abs-multiset-inverse [THEN subst])
    apply (erule add [unfolded defns, simplified])
    done
qed

lemma MCollect-preserves-multiset:
    M ∈ multiset ==> (λx. if P x then M x else 0) ∈ multiset
  apply (simp add: multiset-def)
  apply (rule finite-subset, auto)
  done

lemma count-MCollect [simp]:
    count {# x:M. P x #} a = (if P a then count M a else 0)
  by (simp add: count-def MCollect-def MCollect-preserves-multiset)

lemma set-of-MCollect [simp]: set-of {# x:M. P x #} = set-of M ∩ {x. P x}
  by (auto simp add: set-of-def)

lemma multiset-partition: M = {# x:M. P x #} + {# x:M. ¬ P x #}
  by (subst multiset-eq-conv-count-eq, auto)

lemma add-eq-conv-ex:
  (M + {#a#} = N + {#b#}) =
    (M = N ∧ a = b ∨ (∃ K. M = K + {#b#} ∧ N = K + {#a#}))
  by (auto simp add: add-eq-conv-diff)

declare multiset-typedef [simp del]
```

23.4 Multiset orderings

23.4.1 Well-foundedness

```
definition
  mult1 :: ('a × 'a) set => ('a multiset × 'a multiset) set where
  mult1 r =
    {(N, M). ∃ a M0 K. M = M0 + {#a#} ∧ N = M0 + K ∧
      (∀ b. b :# K --> (b, a) ∈ r)}

definition
  mult :: ('a × 'a) set => ('a multiset × 'a multiset) set where
  mult r = (mult1 r)^+
```

```
lemma not-less-empty [iff]: (M, {#}) ∉ mult1 r
  by (simp add: mult1-def)

lemma less-add: (N, M0 + {#a#}) ∈ mult1 r ==>
    (∃ M. (M, M0) ∈ mult1 r ∧ N = M + {#a#}) ∨
    (∃ K. (∀ b. b :# K --> (b, a) ∈ r) ∧ N = M0 + K)
  (is - ⟹ ?case1 (mult1 r) ∨ ?case2)
proof (unfold mult1-def)
  let ?r = λK a. ∀ b. b :# K --> (b, a) ∈ r
  let ?R = λN M. ∃ a M0 K. M = M0 + {#a#} ∧ N = M0 + K ∧ ?r K a
  let ?case1 = ?case1 {(N, M). ?R N M}

  assume (N, M0 + {#a#}) ∈ {(N, M). ?R N M}
  then have ∃ a' M0' K.
      M0 + {#a#} = M0' + {#a'#} ∧ N = M0' + K ∧ ?r K a' by simp
  then show ?case1 ∨ ?case2
  proof (elim exE conjE)
    fix a' M0' K
    assume N: N = M0' + K and r: ?r K a'
    assume M0 + {#a#} = M0' + {#a'#}
    then have M0 = M0' ∧ a = a' ∨
        (∃ K'. M0 = K' + {#a'#} ∧ M0' = K' + {#a#})
      by (simp only: add-eq-conv-ex)
    then show ?thesis
    proof (elim disjE conjE exE)
      assume M0 = M0' a = a'
      with N r have ?r K a ∧ N = M0 + K by simp
      then have ?case2 .. then show ?thesis ..
    next
      fix K'
      assume M0' = K' + {#a#}
      with N have n: N = K' + K + {#a#} by (simp add: union-ac)

      assume M0 = K' + {#a'#}
      with r have ?R (K' + K) M0 by blast
      with n have ?case1 by simp then show ?thesis ..
    qed
  qed
qed

lemma all-accessible: wf r ==> ∀ M. M ∈ acc (mult1 r)
proof
  let ?R = mult1 r
  let ?W = acc ?R
  {
    fix M M0 a
    assume M0: M0 ∈ ?W
      and wf-hyp: !!b. (b, a) ∈ r ==> (∀ M ∈ ?W. M + {#b#} ∈ ?W)
```

```
    and acc-hyp: ∀ M. (M, M0) ∈ ?R --> M + {#a#} ∈ ?W
  have M0 + {#a#} ∈ ?W
  proof (rule accI [of M0 + {#a#}])
    fix N
    assume (N, M0 + {#a#}) ∈ ?R
    then have ((∃ M. (M, M0) ∈ ?R ∧ N = M + {#a#}) ∨
        (∃ K. (∀ b. b :# K --> (b, a) ∈ r) ∧ N = M0 + K))
      by (rule less-add)
    then show N ∈ ?W
    proof (elim exE disjE conjE)
      fix M assume (M, M0) ∈ ?R and N: N = M + {#a#}
      from acc-hyp have (M, M0) ∈ ?R --> M + {#a#} ∈ ?W ..
      from this and ‹(M, M0) ∈ ?R› have M + {#a#} ∈ ?W ..
      then show N ∈ ?W by (simp only: N)
    next
      fix K
      assume N: N = M0 + K
      assume ∀ b. b :# K --> (b, a) ∈ r
      then have M0 + K ∈ ?W
      proof (induct K)
        case empty
        from M0 show M0 + {#} ∈ ?W by simp
      next
        case (add K x)
        from add.prems have (x, a) ∈ r by simp
        with wf-hyp have ∀ M ∈ ?W. M + {#x#} ∈ ?W by blast
        moreover from add have M0 + K ∈ ?W by simp
        ultimately have (M0 + K) + {#x#} ∈ ?W ..
        then show M0 + (K + {#x#}) ∈ ?W by (simp only: union-assoc)
      qed
      then show N ∈ ?W by (simp only: N)
    qed
  qed
} note tedious-reasoning = this

assume wf: wf r
fix M
show M ∈ ?W
proof (induct M)
  show {#} ∈ ?W
  proof (rule accI)
    fix b assume (b, {#}) ∈ ?R
    with not-less-empty show b ∈ ?W by contradiction
  qed

  fix M a assume M ∈ ?W
  from wf have ∀ M ∈ ?W. M + {#a#} ∈ ?W
  proof induct
    fix a
```

```
      assume r: !!b. (b, a) ∈ r ==> (∀ M ∈ ?W. M + {#b#} ∈ ?W)
      show ∀ M ∈ ?W. M + {#a#} ∈ ?W
      proof
        fix M assume M ∈ ?W
        then show M + {#a#} ∈ ?W
          by (rule acc-induct) (rule tedious-reasoning [OF - r])
      qed
    qed
    from this and ‹M ∈ ?W› show M + {#a#} ∈ ?W ..
  qed
qed
```

```
theorem wf-mult1: wf r ==> wf (mult1 r)
  by (rule acc-wfI) (rule all-accessible)
```

```
theorem wf-mult: wf r ==> wf (mult r)
  unfolding mult-def by (rule wf-trancl) (rule wf-mult1)
```

23.4.2 Closure-free presentation

```
lemma diff-union-single-conv: a :# J ==> I + J − {#a#} = I + (J − {#a#})
  by (simp add: multiset-eq-conv-count-eq)
```

One direction.

```
lemma mult-implies-one-step:
  trans r ==> (M, N) ∈ mult r ==>
    ∃ I J K. N = I + J ∧ M = I + K ∧ J ≠ {#} ∧
    (∀ k ∈ set-of K. ∃ j ∈ set-of J. (k, j) ∈ r)
  apply (unfold mult-def mult1-def set-of-def)
  apply (erule converse-trancl-induct, clarify)
   apply (rule-tac x = M0 in exI, simp, clarify)
  apply (case-tac a :# K)
   apply (rule-tac x = I in exI)
   apply (simp (no-asm))
   apply (rule-tac x = (K − {#a#}) + Ka in exI)
   apply (simp (no-asm-simp) add: union-assoc [symmetric])
   apply (drule-tac f = λM. M − {#a#} in arg-cong)
   apply (simp add: diff-union-single-conv)
   apply (simp (no-asm-use) add: trans-def)
   apply blast
  apply (subgoal-tac a :# I)
   apply (rule-tac x = I − {#a#} in exI)
   apply (rule-tac x = J + {#a#} in exI)
   apply (rule-tac x = K + Ka in exI)
   apply (rule conjI)
    apply (simp add: multiset-eq-conv-count-eq split: nat-diff-split)
   apply (rule conjI)
    apply (drule-tac f = λM. M − {#a#} in arg-cong, simp)
    apply (simp add: multiset-eq-conv-count-eq split: nat-diff-split)
   apply (simp (no-asm-use) add: trans-def)
```

apply *blast*
apply (*subgoal-tac* $a :\# (M0 + \{\#a\#\})$)
apply *simp*
apply (*simp* (*no-asm*))
done

lemma *elem-imp-eq-diff-union*: $a :\# M ==> M = M - \{\#a\#\} + \{\#a\#\}$
by (*simp add*: *multiset-eq-conv-count-eq*)

lemma *size-eq-Suc-imp-eq-union*: $\mathit{size}\ M = \mathit{Suc}\ n ==> \exists\, a\ N.\ M = N + \{\#a\#\}$
apply (*erule size-eq-Suc-imp-elem* [*THEN exE*])
apply (*drule elem-imp-eq-diff-union*, *auto*)
done

lemma *one-step-implies-mult-aux*:
$\mathit{trans}\ r ==>$
$\forall\, I\ J\ K.\ (\mathit{size}\ J = n \wedge J \neq \{\#\} \wedge (\forall\, k \in \mathit{set\text{-}of}\ K.\ \exists\, j \in \mathit{set\text{-}of}\ J.\ (k,\ j) \in r))$
$--> (I + K,\ I + J) \in \mathit{mult}\ r$
apply (*induct-tac n*, *auto*)
apply (*frule size-eq-Suc-imp-eq-union*, *clarify*)
apply (*rename-tac* J', *simp*)
apply (*erule notE*, *auto*)
apply (*case-tac* $J' = \{\#\}$)
apply (*simp add*: *mult-def*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def set-of-def*, *blast*)

Now we know $J' \neq \{\#\}$.

apply (*cut-tac* $M = K$ **and** $P = \lambda x.\ (x,\ a) \in r$ **in** *multiset-partition*)
apply (*erule-tac* $P = \forall\, k \in \mathit{set\text{-}of}\ K.\ ?P\ k$ **in** *rev-mp*)
apply (*erule ssubst*)
apply (*simp add*: *Ball-def*, *auto*)
apply (*subgoal-tac*
$((I + \{\#\ x : K.\ (x,\ a) \in r\ \#\}) + \{\#\ x : K.\ (x,\ a) \notin r\ \#\},$
$(I + \{\#\ x : K.\ (x,\ a) \in r\ \#\}) + J') \in \mathit{mult}\ r$)
prefer *2*
apply *force*
apply (*simp* (*no-asm-use*) *add*: *union-assoc* [*symmetric*] *mult-def*)
apply (*erule trancl-trans*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def set-of-def*)
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = I + J'$ **in** *exI*)
apply (*simp add*: *union-ac*)
done

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

23.4.3 Partial-order properties

instance *multiset* :: (*type*) *ord* **..**

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*: $trans\ \{(x', x).\ x' < (x::'a::order)\}$
unfolding *trans-def* **by** (*blast intro*: *order-less-trans*)

Irreflexivity.

lemma *mult-irrefl-aux*:
$finite\ A ==> (\forall x \in A.\ \exists y \in A.\ x < (y::'a::order)) \Longrightarrow A = \{\}$
by (*induct rule*: *finite-induct*) (*auto intro*: *order-less-trans*)

lemma *mult-less-not-refl*: $\neg M < (M::'a::order\ multiset)$
apply (*unfold less-multiset-def*, *auto*)
apply (*drule trans-base-order* [*THEN mult-implies-one-step*], *auto*)
apply (*drule finite-set-of* [*THEN mult-irrefl-aux* [*rule-format* (*no-asm*)]])
apply (*simp add*: *set-of-eq-empty-iff*)
done

lemma *mult-less-irrefl* [*elim*!]: $M < (M::'a::order\ multiset) ==> R$
using *insert mult-less-not-refl* **by** *fast*

Transitivity.

theorem *mult-less-trans*: $K < M ==> M < N ==> K < (N::'a::order\ multiset)$
unfolding *less-multiset-def mult-def* **by** (*blast intro*: *trancl-trans*)

Asymmetry.

theorem *mult-less-not-sym*: $M < N ==> \neg N < (M::'a::order\ multiset)$
apply *auto*
apply (*rule mult-less-not-refl* [*THEN notE*])
apply (*erule mult-less-trans*, *assumption*)
done

theorem *mult-less-asym*:
$M < N ==> (\neg P ==> N < (M::'a::order\ multiset)) ==> P$
by (*insert mult-less-not-sym*, *blast*)

theorem *mult-le-refl* [*iff*]: $M <= (M::'a::order\ multiset)$
unfolding *le-multiset-def* **by** *auto*

Anti-symmetry.

theorem *mult-le-antisym*:
$M <= N ==> N <= M ==> M = (N::'a::order\ multiset)$
unfolding *le-multiset-def* **by** (*blast dest*: *mult-less-not-sym*)

Transitivity.

```
theorem mult-le-trans:
    K <= M ==> M <= N ==> K <= (N::'a::order multiset)
  unfolding le-multiset-def by (blast intro: mult-less-trans)

theorem mult-less-le: (M < N) = (M <= N ∧ M ≠ (N::'a::order multiset))
  unfolding le-multiset-def by auto
```

Partial order.

```
instance multiset :: (order) order
  apply intro-classes
  apply (rule mult-less-le)
  apply (rule mult-le-refl)
  apply (erule mult-le-trans, assumption)
  apply (erule mult-le-antisym, assumption)
  done
```

23.4.4 Monotonicity of multiset union

```
lemma mult1-union:
    (B, D) ∈ mult1 r ==> trans r ==> (C + B, C + D) ∈ mult1 r
  apply (unfold mult1-def, auto)
  apply (rule-tac x = a in exI)
  apply (rule-tac x = C + M0 in exI)
  apply (simp add: union-assoc)
  done

lemma union-less-mono2: B < D ==> C + B < C + (D::'a::order multiset)
  apply (unfold less-multiset-def mult-def)
  apply (erule trancl-induct)
   apply (blast intro: mult1-union transI order-less-trans r-into-trancl)
  apply (blast intro: mult1-union transI order-less-trans r-into-trancl trancl-trans)
  done

lemma union-less-mono1: B < D ==> B + C < D + (C::'a::order multiset)
  apply (subst union-commute [of B C])
  apply (subst union-commute [of D C])
  apply (erule union-less-mono2)
  done

lemma union-less-mono:
    A < C ==> B < D ==> A + B < C + (D::'a::order multiset)
  apply (blast intro!: union-less-mono1 union-less-mono2 mult-less-trans)
  done

lemma union-le-mono:
    A <= C ==> B <= D ==> A + B <= C + (D::'a::order multiset)
  unfolding le-multiset-def
  by (blast intro: union-less-mono union-less-mono1 union-less-mono2)

lemma empty-leI [iff]: {#} <= (M::'a::order multiset)
```

apply (*unfold le-multiset-def less-multiset-def*)
apply (*case-tac* $M = \{\#\}$)
prefer *2*
apply (*subgoal-tac* $(\{\#\} + \{\#\}, \{\#\} + M) \in$ *mult* (*Collect* (*split op* <)))
prefer *2*
apply (*rule one-step-implies-mult*)
apply (*simp only*: *trans-def*, *auto*)
done

lemma *union-upper1*: $A <= A + (B::'a::order\ multiset)$
proof −
have $A + \{\#\} <= A + B$ **by** (*blast intro*: *union-le-mono*)
then show *?thesis* **by** *simp*
qed

lemma *union-upper2*: $B <= A + (B::'a::order\ multiset)$
by (*subst union-commute*) (*rule union-upper1*)

instance *multiset* :: (*order*) *pordered-ab-semigroup-add*
apply *intro-classes*
apply(*erule union-le-mono*[*OF mult-le-refl*])
done

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* = [])
by (*induct x*) *auto*

lemma *multiset-of-zero-iff-right*[*simp*]: ({#} = *multiset-of x*) = (*x* = [])
by (*induct x*) *auto*

lemma *set-of-multiset-of*[*simp*]: *set-of*(*multiset-of x*) = *set x*
by (*induct x*) *auto*

lemma *mem-set-multiset-eq*: $x \in$ *set xs* = (*x* :# *multiset-of xs*)
by (*induct xs*) *auto*

lemma *multiset-of-append* [*simp*]:
multiset-of (*xs* @ *ys*) = *multiset-of xs* + *multiset-of ys*
by (*induct xs arbitrary*: *ys*) (*auto simp*: *union-ac*)

lemma *surj-multiset-of*: *surj multiset-of*
apply (*unfold surj-def*, *rule allI*)

apply (*rule-tac* $M{=}y$ **in** *multiset-induct*, *auto*)
apply (*rule-tac* $x = x \# xa$ **in** *exI*, *auto*)
done

lemma *set-count-greater-0*: *set* $x = \{a.\ count\ (multiset\text{-}of\ x)\ a > 0\}$
by (*induct* x) *auto*

lemma *distinct-count-atmost-1*:
distinct $x = (!\ a.\ count\ (multiset\text{-}of\ x)\ a = (if\ a \in set\ x\ then\ 1\ else\ 0))$
apply (*induct* x, *simp*, *rule iffI*, *simp-all*)
apply (*rule conjI*)
apply (*simp-all add*: *set-of-multiset-of* [*THEN sym*] *del*: *set-of-multiset-of*)
apply (*erule-tac* $x{=}a$ **in** *allE*, *simp*, *clarify*)
apply (*erule-tac* $x{=}aa$ **in** *allE*, *simp*)
done

lemma *multiset-of-eq-setD*:
multiset-of xs = *multiset-of* $ys \Longrightarrow$ *set* xs = *set* ys
by (*rule*) (*auto simp add*:*multiset-eq-conv-count-eq set-count-greater-0*)

lemma *set-eq-iff-multiset-of-eq-distinct*:
⟦*distinct* x; *distinct* y⟧
$\Longrightarrow$ (*set* x = *set* y) = (*multiset-of* x = *multiset-of* y)
by (*auto simp*: *multiset-eq-conv-count-eq distinct-count-atmost-1*)

lemma *set-eq-iff-multiset-of-remdups-eq*:
(*set* x = *set* y) = (*multiset-of* (*remdups* x) = *multiset-of* (*remdups* y))
apply (*rule iffI*)
apply (*simp add*: *set-eq-iff-multiset-of-eq-distinct*[*THEN iffD1*])
apply (*drule distinct-remdups*[*THEN distinct-remdups*
[*THEN set-eq-iff-multiset-of-eq-distinct*[*THEN iffD2*]]])
apply *simp*
done

lemma *multiset-of-compl-union* [*simp*]:
multiset-of $[x \leftarrow xs.\ P\ x]$ + *multiset-of* $[x \leftarrow xs.\ \neg P\ x]$ = *multiset-of* xs
by (*induct* xs) (*auto simp*: *union-ac*)

lemma *count-filter*:
count (*multiset-of* xs) x = *length* $[y \leftarrow xs.\ y = x]$
by (*induct* xs) *auto*

23.6 Pointwise ordering induced by count

definition
mset-le :: $'a$ *multiset* $\Rightarrow$ $'a$ *multiset* $\Rightarrow$ *bool* (**infix** $\le\#$ *50*) **where**
$(A \le\# B) = (\forall a.\ count\ A\ a \le count\ B\ a)$
definition
mset-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*]: $A \leq\# A$
  unfolding *mset-le-def* **by** *auto*

lemma *mset-le-trans*: $[\![A \leq\# B; B \leq\# C]\!] \Longrightarrow A \leq\# C$
  unfolding *mset-le-def* **by** (*fast intro*: *order-trans*)

lemma *mset-le-antisym*: $[\![A \leq\# B; B \leq\# A]\!] \Longrightarrow A = B$
  apply (*unfold mset-le-def*)
  apply (*rule multiset-eq-conv-count-eq*[*THEN iffD2*])
  apply (*blast intro*: *order-antisym*)
  done

lemma *mset-le-exists-conv*:
  $(A \leq\# B) = (\exists C. B = A + C)$
  apply (*unfold mset-le-def*, *rule iffI*, *rule-tac* $x = B - A$ **in** *exI*)
  apply (*auto intro*: *multiset-eq-conv-count-eq* [*THEN iffD2*])
  done

lemma *mset-le-mono-add-right-cancel*[*simp*]: $(A + C \leq\# B + C) = (A \leq\# B)$
  unfolding *mset-le-def* **by** *auto*

lemma *mset-le-mono-add-left-cancel*[*simp*]: $(C + A \leq\# C + B) = (A \leq\# B)$
  unfolding *mset-le-def* **by** *auto*

lemma *mset-le-mono-add*: $[\![A \leq\# B; C \leq\# D]\!] \Longrightarrow A + C \leq\# B + D$
  apply (*unfold mset-le-def*)
  apply *auto*
  apply (*erule-tac* $x=a$ **in** *allE*)+
  apply *auto*
  done

lemma *mset-le-add-left*[*simp*]: $A \leq\# A + B$
  unfolding *mset-le-def* **by** *auto*

lemma *mset-le-add-right*[*simp*]: $B \leq\# A + B$
  unfolding *mset-le-def* **by** *auto*

lemma *multiset-of-remdups-le*: *multiset-of* (*remdups xs*) $\leq\#$ *multiset-of xs*
apply (*induct xs*)
 apply *auto*
apply (*rule mset-le-trans*)
 apply *auto*
done

interpretation *mset-order*:
  order [*op* $\leq\#$ *op* $<\#$]
  by (*auto intro*: *order.intro mset-le-refl mset-le-antisym*

```
  mset-le-trans simp: mset-less-def)

interpretation mset-order-cancel-semigroup:
  pordered-cancel-ab-semigroup-add [op ≤# op <# op +]
  by unfold-locales (erule mset-le-mono-add [OF mset-le-refl])

interpretation mset-order-semigroup-cancel:
  pordered-ab-semigroup-add-imp-le [op ≤# op <# op +]
  by (unfold-locales) simp

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) ⇒ 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 (cases 2 dvd a)
  case True
  then show ?thesis by (rule dvd-mult2)
next
  case False
  then have Suc (a mod 2) = 2 by (simp add: dvd-eq-mod-eq-0)
  then have Suc a mod 2 = 0 by (simp add: mod-Suc)
  then have 2 dvd Suc a by (simp only:dvd-eq-mod-eq-0)
  then show ?thesis by (rule dvd-mult)
qed

lemma
  assumes eq: nat2-to-nat (u,v) = nat2-to-nat (x,y)
  shows nat2-to-nat-help: u+v ≤ x+y
proof (rule classical)
  assume ¬ ?thesis
  then have contrapos: x+y < u+v
    by simp
  have nat2-to-nat (x,y) < (x+y) ∗ Suc (x+y) div 2 + Suc (x + y)
    by (unfold nat2-to-nat-def) (simp add: Let-def)
  also have . . . = (x+y)∗Suc(x+y) div 2 + 2 ∗ Suc(x+y) div 2
    by (simp only: div-mult-self1-is-m)
  also have . . . = (x+y)∗Suc(x+y) div 2 + 2 ∗ Suc(x+y) div 2
```

```
    + ((x+y)*Suc(x+y) mod 2 + 2 * Suc(x+y) mod 2) div 2
  proof -
    have 2 dvd (x+y)*Suc(x+y)
      by (rule dvd2-a-x-suc-a)
    then have (x+y)*Suc(x+y) mod 2 = 0
      by (simp only: dvd-eq-mod-eq-0)
    also
    have 2 * Suc(x+y) mod 2 = 0
      by (rule mod-mult-self1-is-0)
    ultimately have
      ((x+y)*Suc(x+y) mod 2 + 2 * Suc(x+y) mod 2) div 2 = 0
      by simp
    then show ?thesis
      by simp
  qed
  also have ... = ((x+y)*Suc(x+y) + 2*Suc(x+y)) div 2
    by (rule div-add1-eq [symmetric])
  also have ... = ((x+y+2)*Suc(x+y)) div 2
    by (simp only: add-mult-distrib [symmetric])
  also from contrapos have ... ≤ ((Suc(u+v))*(u+v)) div 2
    by (simp only: mult-le-mono div-le-mono)
  also have ... ≤ nat2-to-nat (u,v)
    by (unfold nat2-to-nat-def) (simp add: Let-def)
  finally show ?thesis
    by (simp only: eq)
qed

theorem nat2-to-nat-inj: inj nat2-to-nat
proof -
  {
    fix u v x y
    assume eq1: nat2-to-nat (u,v) = nat2-to-nat (x,y)
    then have u+v ≤ x+y by (rule nat2-to-nat-help)
    also from eq1 [symmetric] have x+y ≤ u+v
      by (rule nat2-to-nat-help)
    finally have eq2: u+v = x+y .
    with eq1 have ux: u=x
      by (simp add: nat2-to-nat-def Let-def)
    with eq2 have vy: v=y by simp
    with ux have (u,v) = (x,y) by simp
  }
  then have ⋀x y. nat2-to-nat x = nat2-to-nat y ⟹ x=y by fast
  then show ?thesis unfolding inj-on-def by simp
qed

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*} **..**

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*
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Infty-ne-i0* [*simp*]: $\infty \neq$ *0*
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *i0-ne-Infty* [*simp*]: *0* $\neq \infty$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iSuc-Fin* [*simp*]: $iSuc\ (Fin\ n) = Fin\ (Suc\ n)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iSuc-Infty* [*simp*]: $iSuc\ \infty = \infty$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iSuc-ne-0* [*simp*]: $iSuc\ n \neq 0$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iSuc-inject* [*simp*]: $(iSuc\ x = iSuc\ y) = (x = y)$
by (*simp add*: *inat-defs split*:*inat-splits*)

25.3 Ordering relations

lemma *Infty-ilessE* [*elim*!]: $\infty < Fin\ m ==> R$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iless-linear*: $m < n \vee m = n \vee n < (m::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith*)

lemma *iless-not-refl* [*simp*]: $\neg\ n < (n::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iless-trans*: $i < j ==> j < k ==> i < (k::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iless-not-sym*: $n < m ==> \neg\ m < (n::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Fin-iless-mono* [*simp*]: $(Fin\ n < Fin\ m) = (n < m)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Fin-iless-Infty* [*simp*]: $Fin\ n < \infty$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Infty-eq* [*simp*]: $(n < \infty) = (n \neq \infty)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *i0-eq* [*simp*]: $((0::inat) < n) = (n \neq 0)$
by (*fastsimp simp*: *inat-defs split*:*inat-splits*)

lemma *i0-iless-iSuc* [*simp*]: $0 < iSuc\ n$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *not-ilessi0* [*simp*]: $\neg\ n < (0::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Fin-iless*: $n < Fin\ m ==> \exists\, k.\ n = Fin\ k$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iSuc-mono* [*simp*]: $(iSuc\ n < iSuc\ m) = (n < m)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *ile-def2*: $(m \leq n) = (m < n \vee m = (n::inat))$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith*)

lemma *ile-refl* [*simp*]: $n \leq (n::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *ile-trans*: $i \leq j ==> j \leq k ==> i \leq (k::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *ile-iless-trans*: $i \leq j ==> j < k ==> i < (k::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iless-ile-trans*: $i < j ==> j \leq k ==> i < (k::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Infty-ub* [*simp*]: $n \leq \infty$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *i0-lb* [*simp*]: $(0::inat) \leq n$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Infty-ileE* [*elim*!]: $\infty \leq Fin\ m ==> R$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Fin-ile-mono* [*simp*]: $(Fin\ n \leq Fin\ m) = (n \leq m)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith*)

lemma *ilessI1*: $n \leq m ==> n \neq m ==> n < (m::inat)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *ileI1*: $m < n ==> iSuc\ m \leq n$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *Suc-ile-eq*: $(Fin\ (Suc\ m) \leq n) = (Fin\ m < n)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith*)

lemma *iSuc-ile-mono* [*simp*]: $(iSuc\ n \leq iSuc\ m) = (n \leq m)$
by (*simp add*: *inat-defs split*:*inat-splits*)

lemma *iless-Suc-eq* [*simp*]: $(Fin\ m < iSuc\ n) = (Fin\ m \leq n)$
by (*simp add*: *inat-defs split*:*inat-splits*, *arith*)

lemma *not-iSuc-ilei0* [*simp*]: $\neg\ iSuc\ n \leq 0$

```
by (simp add: inat-defs split:inat-splits)

lemma ile-iSuc [simp]: n ≤ iSuc n
by (simp add: inat-defs split:inat-splits)

lemma Fin-ile: n ≤ Fin m ==> ∃ k. n = Fin k
by (simp add: inat-defs split:inat-splits)

lemma chain-incr: ∀ i. ∃ j. Y i < Y j ==> ∃ j. Fin k < Y j
apply (induct-tac k)
 apply (simp (no-asm) only: Fin-0)
 apply (fast intro: ile-iless-trans i0-lb)
apply (erule exE)
apply (drule spec)
apply (erule exE)
apply (drule ileI1)
apply (rule iSuc-Fin [THEN subst])
apply (rule exI)
apply (erule (1) ile-iless-trans)
done

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
  by (cases e) simp-all

theorem lookup-val-cons: lookup (Val a) (x # xs) = None
  by simp

theorem lookup-env-cons:
  lookup (Env b es) (x # xs) =
    (case es x of
      None => None
    | Some e => lookup e xs)
  by (cases es x) simp-all

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)))
  by (simp split: list.split env.split)
```

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
  using assms
proof (induct xs arbitrary: env)
  case Nil
  then have False by simp
  then show ?case ..
next
  case (Cons x xs)
  show ?case
  proof (cases env)
    case Val
    then show ?thesis by simp
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      with Env show ?thesis by simp
    next
      case (Some e)
      note es = ‹es x = Some e›
      show ?thesis
      proof (cases lookup e xs)
        case None
        then have lookup e (xs @ ys) = None by (rule Cons.hyps)
        with Env Some show ?thesis by simp
      next
        case Some
        with Env es have False using Cons.prems by simp
        then show ?thesis ..
      qed
    qed
  qed
qed

theorem lookup-append-some:
  assumes lookup env xs = Some e
  shows lookup env (xs @ ys) = lookup e ys
  using assms
proof (induct xs arbitrary: env e)
```

```
  case Nil
  then have env = e by simp
  then show lookup env ([] @ ys) = lookup e ys by simp
next
  case (Cons x xs)
  note asm = ‹lookup env (x # xs) = Some e›
  show lookup env ((x # xs) @ ys) = lookup e ys
  proof (cases env)
    case (Val a)
    with asm have False by simp
    then show ?thesis ..
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      with asm Env have False by simp
      then show ?thesis ..
    next
      case (Some e')
      note es = ‹es x = Some e'›
      show ?thesis
      proof (cases lookup e' xs)
        case None
        with asm Env es have False by simp
        then show ?thesis ..
      next
        case Some
        with asm Env es have lookup e' xs = Some e
          by simp
        then have lookup e' (xs @ ys) = lookup e ys by (rule Cons.hyps)
        with Env es show ?thesis by simp
      qed
    qed
  qed
qed
```

Successful *lookup* deeper down an environment structure means we are able to peek further up as well. Note that this is basically just the contrapositive statement of *lookup-append-none* above.

```
theorem lookup-some-append:
  assumes lookup env (xs @ ys) = Some e
  shows ∃ e. lookup env xs = Some e
proof −
  from assms have lookup env (xs @ ys) ≠ None by simp
  then have lookup env xs ≠ None
    by (rule contrapos-nn) (simp only: lookup-append-none)
  then show ?thesis by (simp)
qed
```

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

```
theorem lookup-some-upper:
  assumes lookup env (xs @ y # ys) = Some e
  shows ∃ b' es' env'.
    lookup env xs = Some (Env b' es') ∧
    es' y = Some env' ∧
    lookup env' ys = Some e
  using assms
proof (induct xs arbitrary: env e)
  case Nil
  from Nil.prems have lookup env (y # ys) = Some e
    by simp
  then obtain b' es' env' where
      env: env = Env b' es' and
      es': es' y = Some env' and
      look': lookup env' ys = Some e
    by (auto simp add: lookup-eq split: option.splits env.splits)
  from env have lookup env [] = Some (Env b' es') by simp
  with es' look' show ?case by blast
next
  case (Cons x xs)
  from Cons.prems
  obtain b' es' env' where
      env: env = Env b' es' and
      es': es' x = Some env' and
      look': lookup env' (xs @ y # ys) = Some e
    by (auto simp add: lookup-eq split: option.splits env.splits)
  from Cons.hyps [OF look'] obtain b'' es'' env'' where
      upper': lookup env' xs = Some (Env b'' es'') and
      es'': es'' y = Some env'' and
      look'': lookup env'' ys = Some e
    by blast
  from env es' upper' have lookup env (x # xs) = Some (Env b'' es'')
    by simp
  with es'' look'' show ?case by blast
qed
```

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-option :: 'c list => ('a, 'b, 'c) env option
```

```
  => ('a, 'b, 'c) env option => ('a, 'b, 'c) env option

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

hide const update-option
```

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

```
theorem update-nil-none: update [] None env = env
  by (cases env) simp-all

theorem update-nil-some: update [] (Some e) env = e
  by (cases env) simp-all

theorem update-cons-val: update (x # xs) opt (Val a) = Val a
  by simp

theorem update-cons-nil-env:
    update [x] opt (Env b es) = Env b (es (x := opt))
  by (cases es x) simp-all

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))))
  by (cases es x) simp-all

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)))))))
by (simp split: list.split env.split option.split)
```

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'
  using assms
proof (induct xs arbitrary: env e)
  case Nil
  then have env = e by simp
  then show ?case by simp
next
  case (Cons x xs)
  note hyp = Cons.hyps
    and asm = ‹lookup env (x # xs) = Some e›
  show ?case
  proof (cases env)
    case (Val a)
    with asm have False by simp
    then show ?thesis ..
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      with asm Env have False by simp
      then show ?thesis ..
    next
      case (Some e')
      note es = ‹es x = Some e'›
      show ?thesis
      proof (cases xs)
        case Nil
        with Env show ?thesis by simp
```

```
    next
      case (Cons x' xs')
      from asm Env es have lookup e' xs = Some e by simp
     then have lookup (update xs (Some env') e') xs = Some env' by (rule hyp)
      with Env es Cons show ?thesis by simp
    qed
  qed
 qed
qed
```

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
  using assms
proof (induct xs arbitrary: env)
  case Nil
  then have False by simp
  then show ?case ..
next
  case (Cons x xs)
  note hyp = Cons.hyps
    and asm = ‹lookup env (x # xs) = None›
  show update ((x # xs) @ y # ys) opt env = env
  proof (cases env)
    case (Val a)
    then show ?thesis by simp
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      note es = ‹es x = None›
      show ?thesis
        by (cases xs) (simp-all add: es Env fun-upd-idem-iff)
    next
      case (Some e)
      note es = ‹es x = Some e›
      show ?thesis
      proof (cases xs)
        case Nil
        with asm Env Some have False by simp
        then show ?thesis ..
      next
        case (Cons x' xs')
        from asm Env es have lookup e xs = None by simp
```

```
      then have update (xs @ y # ys) opt e = e by (rule hyp)
      with Env es Cons show update ((x # xs) @ y # ys) opt env = env
        by (simp add: fun-upd-idem-iff)
    qed
  qed
 qed
qed

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)
 using assms
proof (induct xs arbitrary: env e)
 case Nil
 then have env = e by simp
 then show ?case by simp
next
 case (Cons x xs)
 note hyp = Cons.hyps
  and asm = ‹lookup env (x # xs) = Some e›
 show lookup (update ((x # xs) @ y # ys) opt env) (x # xs) =
     Some (update (y # ys) opt e)
 proof (cases env)
  case (Val a)
  with asm have False by simp
  then show ?thesis ..
 next
  case (Env b es)
  show ?thesis
  proof (cases es x)
   case None
   with asm Env have False by simp
   then show ?thesis ..
  next
   case (Some e')
   note es = ‹es x = Some e'›
   show ?thesis
   proof (cases xs)
    case Nil
    with asm Env es have e = e' by simp
    with Env es Nil show ?thesis by simp
   next
    case (Cons x' xs')
    from asm Env es have lookup e' xs = Some e by simp
    then have lookup (update (xs @ y # ys) opt e') xs =
      Some (update (y # ys) opt e) by (rule hyp)
    with Env es Cons show ?thesis by simp
   qed
```

```
    qed
  qed
qed
```

Apparently, *update* does not affect the result of subsequent *lookup* operations at independent positions, i.e. in case that the paths for *update* and *lookup* fork at a certain point.

```
theorem lookup-update-other:
  assumes neq: y ≠ (z::'c)
  shows lookup (update (xs @ z # zs) opt env) (xs @ y # ys) =
    lookup env (xs @ y # ys)
proof (induct xs arbitrary: env)
  case Nil
  show ?case
  proof (cases env)
    case Val
    then show ?thesis by simp
  next
    case Env
    show ?thesis
    proof (cases zs)
      case Nil
      with neq Env show ?thesis by simp
    next
      case Cons
      with neq Env show ?thesis by simp
    qed
  qed
next
  case (Cons x xs)
  note hyp = Cons.hyps
  show ?case
  proof (cases env)
    case Val
    then show ?thesis by simp
  next
    case (Env y es)
    show ?thesis
    proof (cases xs)
      case Nil
      show ?thesis
      proof (cases es x)
        case None
        with Env Nil show ?thesis by simp
      next
        case Some
        with neq hyp and Env Nil show ?thesis by simp
      qed
    next
```

```
      case (Cons x' xs')
      show ?thesis
      proof (cases es x)
        case None
        with Env Cons show ?thesis by simp
      next
        case Some
        with neq hyp and Env Cons show ?thesis by simp
      qed
    qed
  qed
qed
```

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 ⟷ e1 = e2 ..

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 ⟷
  x = y ∧ (∀ z∈UNIV. case f z
   of None ⇒ (case g z
        of None ⇒ True | Some - ⇒ False)
    | Some a ⇒ (case g z
        of None ⇒ False | Some b ⇒ a = b)) (is ?env)
    and Val a = Val b ⟷ a = b
    and Val a = Env y g ⟷ False
    and Env x f = Val b ⟷ False
proof −
  have f = g ⟷ (∀ z. 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 ?lhs = ?rhs)
  proof
    assume ?lhs
    then show ?rhs by (auto split: option.splits)
  next
    assume assm: ?rhs (is ∀ z. ?prop z)
    show ?lhs
    proof
      fix z
      from assm have ?prop z ..
      then show f z = g z by (auto split: option.splits)
    qed
  qed
  then show ?env by simp
```

qed *simp-all*

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*
  by (*rule inj-onI*, *simp*)

lemma *inj-Inr* [*simp*]: *inj-on Inr A*
  by (*rule inj-onI*, *simp*)

lemma *inj-Some* [*simp*]: *inj-on Some A*
  by (*rule inj-onI*, *simp*)

lemma *card-Plus*:
  [| *finite A*; *finite B* |] ==> *card* (*A* <+> *B*) = *card A* + *card B*
  unfolding *Plus-def*
  apply (*subgoal-tac Inl* ' *A* ∩ *Inr* ' *B* = {})
  apply (*simp add*: *card-Un-disjoint card-image*)
  apply *fast*
  done

lemma (**in** *type-definition*) *univ*:
  UNIV = *Abs* ' *A*
proof
  show *Abs* ' *A* ⊆ *UNIV* **by** (*rule subset-UNIV*)
  show *UNIV* ⊆ *Abs* ' *A*
  proof
    fix *x* :: *'b*
    have *x* = *Abs* (*Rep x*) **by** (*rule Rep-inverse* [*symmetric*])
    moreover have *Rep x* ∈ *A* **by** (*rule Rep*)
    ultimately show *x* ∈ *Abs* ' *A* **by** (*rule image-eqI*)
  qed
qed

lemma (**in** *type-definition*) *card*: *card* (*UNIV* :: *'b set*) = *card A*
  by (*simp add*: *univ card-image inj-on-def Abs-inject*)

27.2 Cardinalities of types

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

translations *CARD*(*t*) => *card* (*UNIV*::*t set*)

typed-print-translation ⟪
let
  fun card-univ-tr′ show-sorts - [Const (@{const-name UNIV}, Type(-,[T]))] =
    Syntax.const -type-card $ Syntax.term-of-typ show-sorts T;
in [(card, card-univ-tr′)]
end
⟫

lemma *card-unit*: *CARD*(*unit*) = *1*
  unfolding *univ-unit* **by** *simp*

lemma *card-bool*: *CARD*(*bool*) = *2*
  unfolding *univ-bool* **by** *simp*

lemma *card-prod*: *CARD*(′*a*::*finite* × ′*b*::*finite*) = *CARD*(′*a*) ∗ *CARD*(′*b*)
  unfolding *univ-prod* **by** (*simp only*: *card-cartesian-product*)

lemma *card-sum*: *CARD*(′*a*::*finite* + ′*b*::*finite*) = *CARD*(′*a*) + *CARD*(′*b*)
  unfolding *univ-sum* **by** (*simp only*: *finite card-Plus*)

lemma *card-option*: *CARD*(′*a*::*finite option*) = *Suc CARD*(′*a*)
  unfolding *univ-option*
  apply (*subgoal-tac* (*None*::′*a option*) ∉ *range Some*)
  apply (*simp add*: *finite card-image*)
  apply *fast*
  done

lemma *card-set*: *CARD*(′*a*::*finite set*) = *2* ˆ *CARD*(′*a*)
  unfolding *univ-set*
  by (*simp only*: *card-Pow finite numeral-2-eq-2*)

27.3 Numeral Types

typedef (**open**) *num0* = *UNIV* :: *nat set* **..**
typedef (**open**) *num1* = *UNIV* :: *unit set* **..**
typedef (**open**) ′*a bit0* = *UNIV* :: (*bool* ∗ ′*a*) *set* **..**
typedef (**open**) ′*a bit1* = *UNIV* :: (*bool* ∗ ′*a*) *option set* **..**

instance *num1* :: *finite*
proof
  show *finite* (*UNIV*::*num1 set*)
    unfolding *type-definition.univ* [*OF type-definition-num1*]
    using *finite* **by** (*rule finite-imageI*)
qed

instance *bit0* :: (*finite*) *finite*

proof
  show *finite* (*UNIV*::′*a bit0 set*)
    unfolding *type-definition.univ* [*OF type-definition-bit0*]
    using *finite* **by** (*rule finite-imageI*)
qed

instance *bit1* :: (*finite*) *finite*
proof
  show *finite* (*UNIV*::′*a bit1 set*)
    unfolding *type-definition.univ* [*OF type-definition-bit1*]
    using *finite* **by** (*rule finite-imageI*)
qed

lemma *card-num1*: *CARD*(*num1*) = *1*
  unfolding *type-definition.card* [*OF type-definition-num1*]
  by (*simp only*: *card-unit*)

lemma *card-bit0*: *CARD*(′*a*::*finite bit0*) = *2* ∗ *CARD*(′*a*)
  unfolding *type-definition.card* [*OF type-definition-bit0*]
  by (*simp only*: *card-prod card-bool*)

lemma *card-bit1*: *CARD*(′*a*::*finite bit1*) = *Suc* (*2* ∗ *CARD*(′*a*))
  unfolding *type-definition.card* [*OF type-definition-bit1*]
  by (*simp only*: *card-prod card-option card-bool*)

lemma *card-num0*: *CARD* (*num0*) = *0*
  by (*simp add*: *type-definition.card* [*OF type-definition-num0*])

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

parse-translation ⟪
let

val num1-const = Syntax.const Numeral-Type.num1;
val num0-const = Syntax.const Numeral-Type.num0;
val B0-const = Syntax.const Numeral-Type.bit0;
val B1-const = Syntax.const Numeral-Type.bit1;

fun mk-bintype n =
  let
    fun mk-bit n = if n = 0 then B0-const else B1-const;
    fun bin-of n =
      if n = 1 then num1-const
      else if n = 0 then num0-const
      else if n = ~1 then raise TERM (negative type numeral, [])
      else
        let val (q, r) = Integer.div-mod n 2;
        in mk-bit r $ bin-of q end;
  in bin-of n end;

fun numeral-tr (*-NumeralType*) [Const (str, -)] =
      mk-bintype (valOf (Int.fromString str))
  | numeral-tr (*-NumeralType*) ts = raise TERM (numeral-tr, ts);

in [(-NumeralType, numeral-tr)] end;
⟫

print-translation ⟪
let
fun int-of [] = 0
  | int-of (b :: bs) = b + 2 * int-of bs;

fun bin-of (Const (num0, -)) = []
  | bin-of (Const (num1, -)) = [1]
  | bin-of (Const (bit0, -) $ bs) = 0 :: bin-of bs
  | bin-of (Const (bit1, -) $ bs) = 1 :: bin-of bs
  | bin-of t = raise TERM(bin-of, [t]);

fun bit-tr' b [t] =
  let
    val rev-digs = b :: bin-of t handle TERM - => raise Match
    val i = int-of rev-digs;
    val num = string-of-int (abs i);
  in
    Syntax.const -NumeralType $ Syntax.free num
  end
  | bit-tr' b - = raise Match;
```

in [(*bit0*, *bit-tr′ 0*), (*bit1*, *bit-tr′ 1*)] *end*;
⟫

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 (*cases CARD*(′*a*::*finite*) = *0*)
  case *False* **thus** *?thesis* **by** (*simp del*: *card-0-eq*)
next
  case *True*
  thus *?thesis* **by** (*simp add*: *finite*)
qed

lemma *one-le-card-finite* [*simp*]:
  Suc 0 <= *CARD*(′*a*::*finite*)
  by (*simp add*: *less-Suc-eq-le* [*symmetric*] *zero-less-card-finite*)

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*)
  using *two-le-card* [**where** ′*a*=′*a*] **by** *simp*

instance *bit0* :: (*finite*) *card2*
  by *intro-classes* (*simp add*: *one-le-card-finite*)

instance *bit1* :: (*finite*) *card2*
  by *intro-classes* (*simp add*: *one-le-card-finite*)

27.6 Examples

term *TYPE*(*10*)

lemma *CARD*(*0*) = *0* **by** *simp*
lemma *CARD*(*17*) = *17* **by** *simp*

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
  by (induct l) auto
```

28.1 Some examples of rule induction on permutations

```
lemma xperm-empty-imp: [] <~~> ys ==> ys = []
  by (induct xs == []::'a list ys pred: perm) simp-all
```

This more general theorem is easier to understand!

```
lemma perm-length: xs <~~> ys ==> length xs = length ys
  by (induct pred: perm) simp-all

lemma perm-empty-imp: [] <~~> xs ==> xs = []
  by (drule perm-length) auto

lemma perm-sym: xs <~~> ys ==> ys <~~> xs
  by (induct pred: perm) auto
```

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
  by (induct xs) auto

lemma perm-append-swap: xs @ ys <~~> ys @ xs
  apply (induct xs)
   apply simp-all
  apply (blast intro: perm-append-Cons)
  done

lemma perm-append-single: a # xs <~~> xs @ [a]
  by (rule perm.trans [OF - perm-append-swap]) simp

lemma perm-rev: rev xs <~~> xs
  apply (induct xs)
   apply simp-all
  apply (blast intro!: perm-append-single intro: perm-sym)
  done
```

lemma *perm-append1*: *xs* <~~> *ys* ==> *l* @ *xs* <~~> *l* @ *ys*
  by (*induct l*) *auto*

lemma *perm-append2*: *xs* <~~> *ys* ==> *xs* @ *l* <~~> *ys* @ *l*
  by (*blast intro*!: *perm-append-swap perm-append1*)

28.3 Further results

lemma *perm-empty* [*iff*]: ([] <~~> *xs*) = (*xs* = [])
  by (*blast intro*: *perm-empty-imp*)

lemma *perm-empty2* [*iff*]: (*xs* <~~> []) = (*xs* = [])
  apply *auto*
  apply (*erule perm-sym* [*THEN perm-empty-imp*])
  done

lemma *perm-sing-imp*: *ys* <~~> *xs* ==> *xs* = [*y*] ==> *ys* = [*y*]
  by (*induct pred*: *perm*) *auto*

lemma *perm-sing-eq* [*iff*]: (*ys* <~~> [*y*]) = (*ys* = [*y*])
  by (*blast intro*: *perm-sing-imp*)

lemma *perm-sing-eq2* [*iff*]: ([*y*] <~~> *ys*) = (*ys* = [*y*])
  by (*blast dest*: *perm-sym*)

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*
  by (*induct ys*) *auto*

lemma *remove-commute*: *remove x* (*remove y l*) = *remove y* (*remove x l*)
  by (*induct l*) *auto*

lemma *multiset-of-remove* [*simp*]:
    multiset-of (*remove a x*) = *multiset-of x* − {#*a*#}
  apply (*induct x*)
   apply (*auto simp*: *multiset-eq-conv-count-eq*)
  done

Congruence rule

lemma *perm-remove-perm*: *xs* <~~> *ys* ==> *remove z xs* <~~> *remove z ys*
  by (*induct pred*: *perm*) *auto*

lemma *remove-hd* [*simp*]: *remove z* (*z* # *xs*) = *xs*
by *auto*

lemma *cons-perm-imp-perm*: *z* # *xs* <~~> *z* # *ys* ==> *xs* <~~> *ys*
by (*drule-tac z* = *z* **in** *perm-remove-perm*) *auto*

lemma *cons-perm-eq* [*iff*]: (*z*#*xs* <~~> *z*#*ys*) = (*xs* <~~> *ys*)
by (*blast intro*: *cons-perm-imp-perm*)

lemma *append-perm-imp-perm*: *zs* @ *xs* <~~> *zs* @ *ys* ==> *xs* <~~> *ys*
apply (*induct zs arbitrary*: *xs ys rule*: *rev-induct*)
apply (*simp-all* (*no-asm-use*))
apply *blast*
done

lemma *perm-append1-eq* [*iff*]: (*zs* @ *xs* <~~> *zs* @ *ys*) = (*xs* <~~> *ys*)
by (*blast intro*: *append-perm-imp-perm perm-append1*)

lemma *perm-append2-eq* [*iff*]: (*xs* @ *zs* <~~> *ys* @ *zs*) = (*xs* <~~> *ys*)
apply (*safe intro*!: *perm-append2*)
apply (*rule append-perm-imp-perm*)
apply (*rule perm-append-swap* [*THEN perm.trans*])
— the previous step helps this *blast* call succeed quickly
apply (*blast intro*: *perm-append-swap*)
done

lemma *multiset-of-eq-perm*: (*multiset-of xs* = *multiset-of ys*) = (*xs* <~~> *ys*)
apply (*rule iffI*)
apply (*erule-tac* [*2*] *perm.induct*, *simp-all add*: *union-ac*)
apply (*erule rev-mp*, *rule-tac x*=*ys* **in** *spec*)
apply (*induct-tac xs*, *auto*)
apply (*erule-tac x* = *remove a x* **in** *allE*, *drule sym*, *simp*)
apply (*subgoal-tac a* ∈ *set x*)
apply (*drule-tac z*=*a* **in** *perm.Cons*)
apply (*erule perm.trans*, *rule perm-sym*, *erule perm-remove*)
apply (*drule-tac f*=*set-of* **in** *arg-cong*, *simp*)
done

lemma *multiset-of-le-perm-append*:
(*multiset-of xs* ≤# *multiset-of ys*) = (∃ *zs*. *xs* @ *zs* <~~> *ys*)
apply (*auto simp*: *multiset-of-eq-perm*[*THEN sym*] *mset-le-exists-conv*)
apply (*insert surj-multiset-of*, *drule surjD*)
apply (*blast intro*: *sym*)+
done

lemma *perm-set-eq*: *xs* <~~> *ys* ==> *set xs* = *set ys*
by (*metis multiset-of-eq-perm multiset-of-eq-setD*)

```
lemma perm-distinct-iff: xs <~~> ys ==> distinct xs = distinct ys
  apply (induct pred: perm)
     apply simp-all
   apply fastsimp
  apply (metis perm-set-eq)
  done

lemma eq-set-perm-remdups: set xs = set ys ==> remdups xs <~~> remdups ys
  apply (induct xs arbitrary: ys rule: length-induct)
  apply (case-tac remdups xs, simp, simp)
  apply (subgoal-tac a : set (remdups ys))
   prefer 2 apply (metis set.simps(2) insert-iff set-remdups)
  apply (drule split-list) apply(elim exE conjE)
  apply (drule-tac x=list in spec) apply(erule impE) prefer 2
   apply (drule-tac x=ysa@zs in spec) apply(erule impE) prefer 2
    apply simp
    apply (subgoal-tac a#list <~~> a#ysa@zs)
     apply (metis Cons-eq-appendI perm-append-Cons trans)
    apply (metis Cons Cons-eq-appendI distinct.simps(2)
      distinct-remdups distinct-remdups-id perm-append-swap perm-distinct-iff)
   apply (subgoal-tac set (a#list) = set (ysa@a#zs) & distinct (a#list) & distinct
(ysa@a#zs))
    apply (fastsimp simp add: insert-ident)
   apply (metis distinct-remdups set-remdups)
 apply (metis Nat.le-less-trans Suc-length-conv le-def length-remdups-leq less-Suc-eq)
  done

lemma perm-remdups-iff-eq-set: remdups x <~~> remdups y = (set x = set y)
  by (metis List.set-remdups perm-set-eq eq-set-perm-remdups)

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)

setup ⟪
let
  val charr = @{const-name Char}
```

```
  val nibbles = [@{const-name Nibble0}, @{const-name Nibble1},
    @{const-name Nibble2}, @{const-name Nibble3},
    @{const-name Nibble4}, @{const-name Nibble5},
    @{const-name Nibble6}, @{const-name Nibble7},
    @{const-name Nibble8}, @{const-name Nibble9},
    @{const-name NibbleA}, @{const-name NibbleB},
    @{const-name NibbleC}, @{const-name NibbleD},
    @{const-name NibbleE}, @{const-name NibbleF}];
in
  fold (fn target => CodeTarget.add-pretty-char target charr nibbles)
    [SML, OCaml, Haskell]
  #> CodeTarget.add-pretty-list-string Haskell
    @{const-name Nil} @{const-name Cons} charr nibbles
end
⟫
```

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*
  unfolding *int-of-char-def* **by** (*simp add*: *expand-fun-eq*)

definition

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

lemma [*code func*]:
char-of-nat = char-of-int o int
unfolding *char-of-int-def* **by** (*simp add*: *expand-fun-eq*)

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))
by (*cases c*) (*auto simp add*: *nibble-pair-of-nat-char*)

lemma [*code func*, *code inline*]:
char-case f c = split f (nibble-pair-of-nat (nat-of-char c))
by (*cases c*) (*auto simp add*: *nibble-pair-of-nat-char*)

lemma [*code func*]:
size (c::char) = 0
by (*cases c*) *auto*

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 = ($1 < p \wedge (\forall m.$ *m dvd p* −−> $m = 1 \vee m = p$))

lemma *two-is-prime*: *prime 2*
apply (*auto simp add*: *prime-def*)
apply (*case-tac m*)
apply (*auto dest*!: *dvd-imp-le*)
done

lemma *prime-imp-relprime*: *prime p* ==> ¬ *p dvd n* ==> *gcd* (*p*, *n*) = *1*
  apply (*auto simp add*: *prime-def*)
  apply (*metis One-nat-def gcd-dvd1 gcd-dvd2*)
  done

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*
  by (*blast intro*: *relprime-dvd-mult prime-imp-relprime*)

lemma *prime-dvd-square*: *prime p* ==> *p dvd m*ˆ*Suc* (*Suc 0*) ==> *p dvd m*
  by (*auto dest*: *prime-dvd-mult*)

lemma *prime-dvd-power-two*: *prime p* ==> *p dvd* m^2 ==> *p dvd m*
  by (*rule prime-dvd-square*) (*simp-all add*: *power2-eq-square*)

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*])
by *pat-completeness auto*

termination
by (*relation measure size*)
  (*auto simp*: *length-filter-le*[*THEN order-class.le-less-trans*])

end
context *linorder*
begin

lemma *quicksort-permutes* [*simp*]:
  multiset-of (*quicksort xs*) = *multiset-of xs*
by (*induct xs rule*: *quicksort.induct*) (*auto simp*: *union-ac*)

lemma *set-quicksort* [*simp*]: *set* (*quicksort xs*) = *set xs*
by(*simp add*: *set-count-greater-0*)

lemma *sorted-quicksort*: *sorted*(*quicksort xs*)
apply (*induct xs rule*: *quicksort.induct*)
apply *simp*
apply (*simp add*:*sorted-Cons sorted-append not-le less-imp-le*)
apply (*metis leD le-cases le-less-trans*)
done

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 $\sim$ :: *'a => 'a => bool.*

class *eqv* = *type* +
fixes *eqv* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\sim$ *50*)

class *equiv* = *eqv* +
assumes *equiv-refl* [*intro*]: $x \sim x$
assumes *equiv-trans* [*trans*]: $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$
assumes *equiv-sym* [*sym*]: $x \sim y \Longrightarrow y \sim x$

lemma *equiv-not-sym* [*sym*]: $\neg$ ($x \sim y$) ==> $\neg$ ($y \sim$ (x::*'a*::*equiv*))
proof −
assume $\neg$ ($x \sim y$) **then show** $\neg$ ($y \sim x$)
by (*rule contrapos-nn*) (*rule equiv-sym*)
qed

lemma *not-equiv-trans1* [*trans*]: $\neg$ ($x \sim y$) ==> $y \sim z$ ==> $\neg$ ($x \sim$ (z::*'a*::*equiv*))
proof −
assume $\neg$ ($x \sim y$) **and** $y \sim z$
show $\neg$ ($x \sim z$)
proof
assume $x \sim z$
also from ‹$y \sim z$› **have** $z \sim y$ **..**
finally have $x \sim y$ **.**
with ‹$\neg$ ($x \sim y$)› **show** *False* **by** *contradiction*
qed

qed

lemma *not-equiv-trans2* [*trans*]: $x \sim y ==> \neg (y \sim z) ==> \neg (x \sim (z::'a::equiv))$
proof –
  assume $\neg (y \sim z)$ **then have** $\neg (z \sim y)$ **..**
  also assume $x \sim y$ **then have** $y \sim x$ **..**
  finally have $\neg (z \sim x)$ **. then show** $(\neg x \sim z)$ **..**
qed

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

typedef $'a\ quot = \{\{x.\ a \sim x\} \mid a::'a::eqv.\ True\}$
  by *blast*

lemma *quotI* [*intro*]: $\{x.\ a \sim x\} \in quot$
  unfolding *quot-def* **by** *blast*

lemma *quotE* [*elim*]: $R \in quot ==> (!!a.\ R = \{x.\ a \sim x\} ==> C) ==> C$
  unfolding *quot-def* **by** *blast*

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 (*cases A*)
  fix R **assume** R: $A = Abs\text{-}quot\ R$
  assume $R \in quot$ **then have** $\exists\, a.\ R = \{x.\ a \sim x\}$ **by** *blast*
  with R **have** $\exists\, a.\ A = Abs\text{-}quot\ \{x.\ a \sim x\}$ **by** *blast*
  then show *?thesis* **unfolding** *class-def* **.**
qed

lemma *quot-cases* [*cases type*: *quot*]: $(!!a.\ A = \lfloor a \rfloor ==> C) ==> C$
  using *quot-exhaust* **by** *blast*

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
  assume *eq*: $\lfloor a \rfloor = \lfloor b \rfloor$
  show $a \sim b$
  proof –
    from *eq* **have** $\{x.\ a \sim x\} = \{x.\ b \sim x\}$
      by (*simp only*: *class-def Abs-quot-inject quotI*)
    moreover have $a \sim a$ **..**

```
    ultimately have a ∈ {x. b ∼ x} by blast
    then have b ∼ a by blast
    then show ?thesis ..
  qed
next
  assume ab: a ∼ b
  show ⌊a⌋ = ⌊b⌋
  proof −
    have {x. a ∼ x} = {x. b ∼ x}
    proof (rule Collect-cong)
      fix x show (a ∼ x) = (b ∼ x)
      proof
        from ab have b ∼ a ..
        also assume a ∼ x
        finally show b ∼ x .
      next
        note ab
        also assume b ∼ x
        finally show a ∼ x .
      qed
    qed
    then show ?thesis by (simp only: class-def)
  qed
qed
```

33.3 Picking representing elements

```
definition
  pick :: 'a::equiv quot => 'a where
  pick A = (SOME a. A = ⌊a⌋)

theorem pick-equiv [intro]: pick ⌊a⌋ ∼ a
proof (unfold pick-def)
  show (SOME x. ⌊a⌋ = ⌊x⌋) ∼ a
  proof (rule someI2)
    show ⌊a⌋ = ⌊a⌋ ..
    fix x assume ⌊a⌋ = ⌊x⌋
    then have a ∼ x .. then show x ∼ a ..
  qed
qed

theorem pick-inverse [intro]: ⌊pick A⌋ = A
proof (cases A)
  fix a assume a: A = ⌊a⌋
  then have pick A ∼ a by (simp only: pick-equiv)
  then have ⌊pick A⌋ = ⌊a⌋ ..
  with a show ?thesis by simp
qed
```

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 −
  from *eq* **and** *P* **have** $f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ (pick\ \lfloor a\rfloor)\ (pick\ \lfloor b\rfloor)$ **by** (*simp only*:)
  also have ... $= g\ a\ b$
  proof (*rule cong*)
    show $\lfloor pick\ \lfloor a\rfloor\rfloor = \lfloor a\rfloor$ **..**
    moreover
    show $\lfloor pick\ \lfloor b\rfloor\rfloor = \lfloor b\rfloor$ **..**
    moreover
    show $P\ \lfloor a\rfloor\ \lfloor b\rfloor$ **by** (*rule P*)
    ultimately show $P\ \lfloor pick\ \lfloor a\rfloor\rfloor\ \lfloor pick\ \lfloor b\rfloor\rfloor$ **by** (*simp only*:)
  qed
  finally show *?thesis* **.**
qed

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 $f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ a\ b$
  using *assms* **and** *TrueI*
  by (*rule quot-cond-function*)

theorem *quot-function'*:
  $(!!X\ Y.\ f\ X\ Y == g\ (pick\ X)\ (pick\ Y)) ==>$
    $(!!x\ x'\ y\ y'.\ x \sim x' ==> y \sim y' ==> g\ x\ y = g\ x'\ y') ==>$
    $f\ \lfloor a\rfloor\ \lfloor b\rfloor = g\ a\ b$
  by (*rule quot-function*) (*simp-all only*: *quot-equality*)

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*) ∈ *r*)

lemma *choice-n*:
  assumes *P0*: *P x0*
    and *Pstep*: !!*x*. *P x* ==> ∃ *y*. *P y* & (*x*,*y*) ∈ *r*
  shows *P* (*choice P r n*)
proof (*induct n*)
  case *0* **show** *?case* **by** (*force intro*: *someI P0*)
next
  case *Suc* **thus** *?case* **by** (*auto intro*: *someI2-ex* [*OF Pstep*])
qed

lemma *dependent-choice*:
  assumes *trans*: *trans r*
    and *P0*: *P x0*
    and *Pstep*: !!*x*. *P x* ==> ∃ *y*. *P y* & (*x*,*y*) ∈ *r*
  obtains *f* :: *nat* => *'a* **where**
   !!*n*. *P* (*f n*) **and** !!*n m*. *n* < *m* ==> (*f n*, *f m*) ∈ *r*
proof
  fix *n*
  show *P* (*choice P r n*) **by** (*blast intro*: *choice-n* [*OF P0 Pstep*])
next
  have *PSuc*: ∀ *n*. (*choice P r n*, *choice P r* (*Suc n*)) ∈ *r*
   using *Pstep* [*OF choice-n* [*OF P0 Pstep*]]
   by (*auto intro*: *someI2-ex*)
  fix *n m* :: *nat*
  assume *less*: *n* < *m*
  show (*choice P r n*, *choice P r m*) ∈ *r* **using** *PSuc*
   by (*auto intro*: *less-Suc-induct* [*OF less*] *transD* [*OF trans*])
qed

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 = (∀ *X*. *X* ⊆ *Y* & *finite X* & *card X* = *r* --> *f X* < *s*)

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

lemma *part-Suc-imp-part*:
   [| *infinite Y*; *part* (*Suc r*) *s Y f*; *y* ∈ *Y* |]
   ==> *part r s* (*Y* − {*y*}) (%*u*. *f* (*insert y u*))
  apply(*simp add*: *part-def*, *clarify*)
  apply(*drule-tac x*=*insert y X* **in** *spec*)
  apply(*force*)

done

lemma *part-subset*: *part r s YY f* ==> $Y \subseteq YY$ ==> *part r s Y f*
unfolding *part-def* **by** *blast*

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 --> f X = t')
proof (*induct r*)
case *0*
thus *?case* **by** (*auto simp add*: *part-def card-eq-0-iff cong*: *conj-cong*)
next
case (*Suc r*)
show *?case*
proof −
from *Suc.prems infinite-imp-nonempty* **obtain** *yy* **where** *yy*: $yy \in YY$ **by** *blast*
let *?ramr* = {((y,Y,t),(y',Y',t')). $y' \in Y$ & $Y' \subseteq Y$}
let *?propr* = %(y,Y,t).
$y \in YY$ & $y \notin Y$ & $Y \subseteq YY$ & *infinite* Y & $t < s$
& ($\forall$ X. $X \subseteq Y$ & *finite X* & *card X* = r --> (*f o insert y*) X = t)
have *infYY'*: *infinite* ($YY - \{yy\}$) **using** *Suc.prems* **by** *auto*
have *partf'*: *part r s* ($YY - \{yy\}$) ($f \circ$ *insert yy*)
by (*simp add*: *o-def part-Suc-imp-part yy Suc.prems*)
have *transr*: *trans ?ramr* **by** (*force simp add*: *trans-def*)
from *Suc.hyps* [*OF infYY' partf'*]
obtain *Y0* **and** *t0*
where $Y0 \subseteq YY - \{yy\}$ *infinite Y0* $t0 < s$
$\forall X. X \subseteq Y0 \wedge$ *finite X* $\wedge$ *card X* = $r \longrightarrow$ ($f \circ$ *insert yy*) X = $t0$
by *blast*
with *yy* **have** *propr0*: *?propr*(*yy*,*Y0*,*t0*) **by** *blast*
have *proprstep*: $\bigwedge x$. *?propr x* $\Longrightarrow$ $\exists$ y. *?propr y* $\wedge$ $(x, y) \in$ *?ramr*
proof −
fix *x*
assume *px*: *?propr x* **thus** *?thesis x*
proof (*cases x*)
case (*fields yx Yx tx*)
then obtain *yx'* **where** *yx'*: $yx' \in Yx$ **using** *px*
by (*blast dest*: *infinite-imp-nonempty*)
have *infYx'*: *infinite* ($Yx - \{yx'\}$) **using** *fields px* **by** *auto*
with *fields px yx' Suc.prems*
have *partfx'*: *part r s* ($Yx - \{yx'\}$) ($f \circ$ *insert yx'*)
by (*simp add*: *o-def part-Suc-imp-part part-subset* [**where** *?YY*=*YY*])

```
    from Suc.hyps [OF infYx′ partfx′]
    obtain Y′ and t′
    where Y′: Y′ ⊆ Yx − {yx′}  infinite Y′  t′ < s
          ∀X. X⊆Y′ ∧ finite X ∧ card X = r ⟶ (f ∘ insert yx′) X = t′
      by blast
    show ?thesis
    proof
      show ?propr (yx′,Y′,t′) & (x, (yx′,Y′,t′)) ∈ ?ramr
        using fields Y′ yx′ px by blast
    qed
  qed
qed
from dependent-choice [OF transr propr0 proprstep]
obtain g where pg: !!n::nat.  ?propr (g n)
  and rg: !!n m. n<m ==> (g n, g m) ∈ ?ramr by blast
let ?gy = (λn. let (y,Y,t) = g n in y)
let ?gt = (λn. let (y,Y,t) = g n in t)
have rangeg: ∃k. range ?gt ⊆ {..<k}
proof (intro exI subsetI)
  fix x
  assume x ∈ range ?gt
  then obtain n where x = ?gt n ..
  with pg [of n] show x ∈ {..<s} by (cases g n) auto
qed
have finite (range ?gt)
  by (simp add: finite-nat-iff-bounded rangeg)
then obtain s′ and n′
  where s′: s′ = ?gt n′
    and infeqs′: infinite {n. ?gt n = s′}
  by (rule inf-img-fin-domE) (auto simp add: vimage-def intro: nat-infinite)
with pg [of n′] have less′: s′<s by (cases g n′) auto
have inj-gy: inj ?gy
proof (rule linorder-injI)
  fix m m′ :: nat assume less: m < m′ show ?gy m ≠ ?gy m′
    using rg [OF less] pg [of m] by (cases g m, cases g m′) auto
qed
show ?thesis
proof (intro exI conjI)
  show ?gy ‘ {n. ?gt n = s′} ⊆ YY using pg
    by (auto simp add: Let-def split-beta)
  show infinite (?gy ‘ {n. ?gt n = s′}) using infeqs′
    by (blast intro: inj-gy [THEN subset-inj-on] dest: finite-imageD)
  show s′ < s by (rule less′)
  show ∀X. X ⊆ ?gy ‘ {n. ?gt n = s′} & finite X & card X = Suc r
      --> f X = s′
  proof −
    {fix X
     assume X ⊆ ?gy ‘ {n. ?gt n = s′}
       and cardX: finite X card X = Suc r
```

```
        then obtain AA where AA: AA ⊆ {n. ?gt n = s'} and Xeq: X = ?gy‘AA
          by (auto simp add: subset-image-iff)
        with cardX have AA≠{} by auto
        hence AAleast: (LEAST x. x ∈ AA) ∈ AA by (auto intro: LeastI-ex)
        have f X = s'
        proof (cases g (LEAST x. x ∈ AA))
          case (fields ya Ya ta)
          with AAleast Xeq
          have ya: ya ∈ X by (force intro!: rev-image-eqI)
          hence f X = f (insert ya (X − {ya})) by (simp add: insert-absorb)
          also have ... = ta
          proof −
            have X − {ya} ⊆ Ya
            proof
              fix x assume x: x ∈ X − {ya}
              then obtain a' where xeq: x = ?gy a' and a': a' ∈ AA
                by (auto simp add: Xeq)
              hence a' ≠ (LEAST x. x ∈ AA) using x fields by auto
              hence lessa': (LEAST x. x ∈ AA) < a'
                using Least-le [of %x. x ∈ AA, OF a'] by arith
              show x ∈ Ya using xeq fields rg [OF lessa'] by auto
            qed
            moreover
            have card (X − {ya}) = r
              by (simp add: cardX ya)
            ultimately show ?thesis
              using pg [of LEAST x. x ∈ AA] fields cardX
              by (clarsimp simp del:insert-Diff-single)
          qed
          also have ... = s' using AA AAleast fields by auto
          finally show ?thesis .
        qed}
      thus ?thesis by blast
    qed
  qed
 qed
qed


theorem Ramsey:
  fixes s r :: nat and Z::'a set and f::'a set => nat
  shows
   [|infinite Z;
      ∀X. X ⊆ Z & finite X & card X = r −−> f X < s|]
  ==> ∃Y t. Y ⊆ Z & infinite Y & t < s
          & (∀X. X ⊆ Y & finite X & card X = r −−> f X = t)
by (blast intro: Ramsey-induction [unfolded part-def])
```

corollary *Ramsey2*:
  fixes s::*nat* **and** Z::$'a$ *set* **and** f::$'a$ *set* => *nat*
  assumes *infZ*: *infinite* Z
    and *part*: $\forall x \in Z.\ \forall y \in Z.\ x \neq y$ --> $f\{x,y\} < s$
  shows
  $\exists\, Y\ t.\ Y \subseteq Z$ & *infinite* Y & $t < s$ & $(\forall x \in Y.\ \forall y \in Y.\ x \neq y$ --> $f\{x,y\} = t)$
proof −
  have *part2*: $\forall X.\ X \subseteq Z$ & *finite* X & *card* $X = 2$ --> $f\ X < s$
    using *part* **by** (*fastsimp simp add*: *nat-number card-Suc-eq*)
  obtain $Y\ t$
    where $Y \subseteq Z$ *infinite* $Y\ t < s$
      $(\forall X.\ X \subseteq Y$ & *finite* X & *card* $X = 2$ --> $f\ X = t)$
    by (*insert Ramsey* [*OF infZ part2*]) *auto*
  moreover from *this* **have** $\forall x \in Y.\ \forall y \in Y.\ x \neq y \longrightarrow f\ \{x,\ y\} = t$ **by** *auto*
  ultimately show *?thesis* **by** *iprover*
qed

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-wf $r = (\exists\, T.\ \exists\, n$::*nat*. $(\forall i<n.\ wf(T\ i))$ & $r = (\bigcup i<n.\ T\ i))$

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

lemma *transition-idx-less*:
  [| $i<j$; $(s\ j,\ s\ i) \in T\ k$; $k<n$ |] ==> *transition-idx* $s\ T\ \{i,j\} < n$
apply (*subgoal-tac transition-idx* $s\ T\ \{i,\ j\} \leq k$, *simp*)
apply (*simp add*: *transition-idx-def*, *blast intro*: *Least-le*)
done

lemma *transition-idx-in*:
  [| $i<j$; $(s\ j,\ s\ i) \in T\ k$ |] ==> $(s\ j,\ s\ i) \in T$ (*transition-idx* $s\ T\ \{i,j\}$)
apply (*simp add*: *transition-idx-def doubleton-eq-iff conj-disj-distribR*
      cong: *conj-cong*)
apply (*erule LeastI*)
done

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

lemma *disj-wf*:

```
    disj-wf(r) = (∃ T. ∃ n::nat. (∀ i<n. wf(T i)) & r ⊆ (⋃ i<n. T i))
apply (auto simp add: disj-wf-def)
apply (rule-tac x=%i. T i Int r in exI)
apply (rule-tac x=n in exI)
apply (force simp add: wf-Int1)
done

theorem trans-disj-wf-implies-wf:
  assumes transr: trans r
      and dwf:    disj-wf(r)
  shows wf r
proof (simp only: wf-iff-no-infinite-down-chain, rule notI)
  assume ∃ s. ∀ i. (s (Suc i), s i) ∈ r
  then obtain s where sSuc: ∀ i. (s (Suc i), s i) ∈ r ..
  have s: !!i j. i < j ==> (s j, s i) ∈ r
  proof −
    fix i and j::nat
    assume less: i<j
    thus (s j, s i) ∈ r
    proof (rule less-Suc-induct)
      show ⋀i. (s (Suc i), s i) ∈ r by (simp add: sSuc)
      show ⋀i j k. ⟦(s j, s i) ∈ r; (s k, s j) ∈ r⟧ ⟹ (s k, s i) ∈ r
        using transr by (unfold trans-def, blast)
    qed
  qed
  from dwf
  obtain T and n::nat where wfT: ∀ k<n. wf(T k) and r: r = (⋃ k<n. T k)
    by (auto simp add: disj-wf-def)
  have s-in-T: ⋀i j. i<j ==> ∃ k. (s j, s i) ∈ T k & k<n
  proof −
    fix i and j::nat
    assume less: i<j
    hence (s j, s i) ∈ r by (rule s [of i j])
    thus ∃ k. (s j, s i) ∈ T k & k<n by (auto simp add: r)
  qed
  have trless: !!i j. i≠j ==> transition-idx s T {i,j} < n
    apply (auto simp add: linorder-neq-iff)
    apply (blast dest: s-in-T transition-idx-less)
    apply (subst insert-commute)
    apply (blast dest: s-in-T transition-idx-less)
    done
  have
   ∃ K k. K ⊆ UNIV & infinite K & k < n &
          (∀ i∈K. ∀ j∈K. i≠j --> transition-idx s T {i,j} = k)
    by (rule Ramsey2) (auto intro: trless nat-infinite)
  then obtain K and k
    where infK: infinite K and less: k < n and
          allk: ∀ i∈K. ∀ j∈K. i≠j --> transition-idx s T {i,j} = k
    by auto
```

```
  have ∀ m. (s (enumerate K (Suc m)), s(enumerate K m)) ∈ T k
  proof
    fix m::nat
    let ?j = enumerate K (Suc m)
    let ?i = enumerate K m
    have jK: ?j ∈ K by (simp add: enumerate-in-set infK)
    have iK: ?i ∈ K by (simp add: enumerate-in-set infK)
    have ij: ?i < ?j by (simp add: enumerate-step infK)
    have ijk: transition-idx s T {?i,?j} = k using iK jK ij
      by (simp add: allk)
    obtain k' where (s ?j, s ?i) ∈ T k' k'<n
      using s-in-T [OF ij] by blast
    thus (s ?j, s ?i) ∈ T k
      by (simp add: ijk [symmetric] transition-idx-in ij)
  qed
  hence ~ wf(T k) by (force simp add: wf-iff-no-infinite-down-chain)
  thus False using wfT less by blast
qed

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*

print-ast-translation ⟪
[(@{*const-syntax run*}, *fn* (*f*::*ts*) => *Syntax.mk-appl f ts*)]
⟫

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*]:
$\bigwedge f.\ run\ (run\ f) = run\ f$
$\bigwedge f\ g.\ run\ f \gg= g = f \gg= g$
$\bigwedge f\ g.\ run\ f \gg g = f \gg g$
$\bigwedge f\ g.\ f \gg= (\lambda x.\ run\ g) = f \gg= (\lambda x.\ g)$
$\bigwedge f\ g.\ f \gg run\ g = f \gg g$
$\bigwedge f.\ f = run\ f \longleftrightarrow True$
$\bigwedge f.\ run\ f = f \longleftrightarrow True$
unfolding *run-def* **by** *rule+*

35.4 Monad laws

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

lemma
return-mbind [*simp*]: $return\ x \gg= f = f\ x$
unfolding *mbind-def* **by** (*simp add*: *expand-fun-eq*)

lemma
mbind-return [*simp*]: $x \gg= return = x$
unfolding *mbind-def* **by** (*simp add*: *expand-fun-eq split-Pair*)

lemma
id-fcomp [*simp*]: $id \gg f = f$
unfolding *fcomp-def* **by** *simp*

lemma
fcomp-id [*simp*]: $f \gg id = f$
unfolding *fcomp-def* **by** *simp*

lemma
mbind-mbind [*simp*]: $(f \gg= g) \gg= h = f \gg= (\lambda x.\ g\ x \gg= h)$
unfolding *mbind-def* **by** (*simp add*: *split-def expand-fun-eq*)

lemma
mbind-fcomp [*simp*]: $(f \gg= g) \gg h = f \gg= (\lambda x.\ g\ x \gg h)$
unfolding *mbind-def fcomp-def* **by** (*simp add*: *split-def expand-fun-eq*)

lemma
fcomp-mbind [*simp*]: $(f \gg g) \gg= h = f \gg (g \gg= h)$
unfolding *mbind-def fcomp-def* **by** (*simp add*: *split-def expand-fun-eq*)

lemma
fcomp-fcomp [*simp*]: $(f \gg g) \gg h = f \gg (g \gg h)$
unfolding *fcomp-def o-assoc* **..**

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 => $f \gg= (\lambda x.\ g)$

```
  -fcomp f g => f ≫ g
  -let x t f => CONST Let t (λx. f)
  -nil f => f
```

print-translation ⟪

```
let
  fun dest-abs-eta (Abs (abs as (-, ty, -))) =
        let
          val (v, t) = Syntax.variant-abs abs;
        in ((v, ty), t) end
    | dest-abs-eta t =
        let
          val (v, t) = Syntax.variant-abs (, dummyT, t $ Bound 0);
        in ((v, dummyT), t) end
  fun unfold-monad (Const (@{const-syntax mbind}, -) $ f $ g) =
        let
          val ((v, ty), g') = dest-abs-eta g;
        in Const (-mbind, dummyT) $ Free (v, ty) $ f $ unfold-monad g' end
    | unfold-monad (Const (@{const-syntax fcomp}, -) $ f $ g) =
        Const (-fcomp, dummyT) $ f $ unfold-monad g
    | unfold-monad (Const (@{const-syntax Let}, -) $ f $ g) =
        let
          val ((v, ty), g') = dest-abs-eta g;
        in Const (-let, dummyT) $ Free (v, ty) $ f $ unfold-monad g' end
    | unfold-monad (Const (@{const-syntax Pair}, -) $ f) =
        Const (return, dummyT) $ f
    | unfold-monad f = f;
  fun tr' (f::ts) =
    list-comb (Const (-do, dummyT) $ unfold-monad f, ts)
in [(@{const-syntax run}, tr')] end;
```
⟫

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 ≫ list f ys

by (*induct xs*) (*simp-all del*: *id-apply*)

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 (*induct f xs arbitrary*: *g ys rule*: *list.induct*)
case *1* **then show** *?case* **by** *simp*
next
case (*2 f x xs g*)
from *2* **have** $\bigwedge y.\ y \in set\ (x\ \#\ xs) \Longrightarrow f\ y = g\ y$ **by** *auto*
then have $\bigwedge y.\ y \in set\ xs \Longrightarrow f\ y = g\ y$ **by** *auto*
with *2* **have** *list f xs* = *list g xs* **by** *auto*
with *2* **have** *list f* (*x* # *xs*) = *list g* (*x* # *xs*) **by** *auto*
with *2* **show** *list f* (*x* # *xs*) = *list g ys* **by** *auto*
qed

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 (*induct f xs arbitrary*: *g ys rule*: *list-yield.induct*)
case *1* **then show** *?case* **by** *simp*
next
case (*2 f x xs g*)
from *2* **have** $\bigwedge y.\ y \in set\ (x\ \#\ xs) \Longrightarrow f\ y = g\ y$ **by** *auto*
then have $\bigwedge y.\ y \in set\ xs \Longrightarrow f\ y = g\ y$ **by** *auto*
with *2* **have** *list-yield f xs* = *list-yield g xs* **by** *auto*
with *2* **have** *list-yield f* (*x* # *xs*) = *list-yield g* (*x* # *xs*) **by** *auto*
with *2* **show** *list-yield f* (*x* # *xs*) = *list-yield g ys* **by** *auto*
qed

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*)
by *auto*

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 −
  have *f x* = *while test do x* **using** *fdef* **by** *simp*
  also have ... = (*if test x then while test do* (*do x*) *else x*)
    by(*rule while-unfold*)
  also have ... = (*if test x then f*(*do x*) *else x*) **by**(*simp add*:*fdef*[*symmetric*])
  finally show *?thesis* **.**
qed

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

theorem *while-rule-lemma*:
  assumes *invariant*: !!*s*. *P s* ==> *b s* ==> *P* (*c s*)
    and *terminate*: !!*s*. *P s* ==> ¬ *b s* ==> *Q s*
    and *wf*: *wf* {(*t*, *s*). *P s* ∧ *b s* ∧ *t* = *c s*}
  shows *P s* ⟹ *Q* (*while b c s*)
  using *wf*
  apply (*induct s*)
  apply *simp*
  apply (*subst while-unfold*)
  apply (*simp add*: *invariant terminate*)
  done

theorem *while-rule*:
  [| *P s*;
      !!*s*. [| *P s*; *b s* |] ==> *P* (*c s*);
      !!*s*. [| *P s*; ¬ *b s* |] ==> *Q s*;
      wf r;
      !!*s*. [| *P s*; *b s* |] ==> (*c s*, *s*) ∈ *r* |] ==>
   Q (*while b c s*)
  apply (*rule while-rule-lemma*)
     prefer *4* **apply** *assumption*
    apply *blast*
   apply *blast*
  apply (*erule wf-subset*)
  apply *blast*
  done

An application: computation of the *lfp* on finite sets via iteration.

theorem *lfp-conv-while*:
  [| *mono f*; *finite U*; *f U* = *U* |] ==>
    lfp f = *fst* (*while* (λ(*A*, *fA*). *A* ≠ *fA*) (λ(*A*, *fA*). (*fA*, *f fA*)) ({}, *f* {}))
apply (*rule-tac P* = λ(*A*, *B*). (*A* ⊆ *U* ∧ *B* = *f A* ∧ *A* ⊆ *B* ∧ *B* ⊆ *lfp f*) **and**
                r = ((*Pow U* × *UNIV*) × (*Pow U* × *UNIV*)) ∩
                    inv-image finite-psubset (*op* − *U o fst*) **in** *while-rule*)

```
   apply (subst lfp-unfold)
    apply assumption
   apply (simp add: monoD)
  apply (subst lfp-unfold)
   apply assumption
  apply clarsimp
  apply (blast dest: monoD)
 apply (fastsimp intro!: lfp-lowerbound)
 apply (blast intro: wf-finite-psubset Int-lower2 [THEN [2] wf-subset])
apply (clarsimp simp add: finite-psubset-def order-less-le)
apply (blast intro!: finite-Diff dest: monoD)
done
```

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 (λN::int set. {0} ∪ {(n + 2) mod 6 | n. n ∈ N})) =
  P {0, 4, 2}
proof −
  have seteq: !!A B. (A = B) = ((!a : A. a:B) & (!b:B. b:A))
    by blast
  have aux: !!f A B. {f n | n. A n ∨ B n} = {f n | n. A n} ∪ {f n | n. B n}
    apply blast
    done
  show ?thesis
    apply (subst lfp-conv-while [where ?U = {0, 1, 2, 3, 4, 5}])
       apply (rule monoI)
      apply blast
     apply simp
    apply (simp add: aux set-eq-subset)
```

The fixpoint computation is performed purely by rewriting:

```
    apply (simp add: while-unfold aux seteq del: subset-empty)
    done
qed

end
```

37 Word: Binary Words

```
theory Word
imports Main
begin
```

37.1 Auxilary Lemmas

```
lemma max-le [intro!]: [| x ≤ z; y ≤ z |] ==> max x y ≤ z
```

by (*simp add*: *max-def*)

lemma *max-mono*:
fixes *x* :: *'a*::*linorder*
assumes *mf*: *mono f*
shows $max\ (f\ x)\ (f\ y) \le f\ (max\ x\ y)$
proof −
from *mf* **and** *le-maxI1* [*of x y*]
have *fx*: $f\ x \le f\ (max\ x\ y)$ **by** (*rule monoD*)
from *mf* **and** *le-maxI2* [*of y x*]
have *fy*: $f\ y \le f\ (max\ x\ y)$ **by** (*rule monoD*)
from *fx* **and** *fy*
show $max\ (f\ x)\ (f\ y) \le f\ (max\ x\ y)$ **by** *auto*
qed

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

lemma *int-nat-two-exp*: *2 ^ k = int (2 ^ k)*
by (*simp add*: *zpower-int* [*symmetric*])

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*
by (*cases* *b*) *simp-all*

lemma *bitand-cancel* [*simp*]: (*b* *bitand* *b*) = *b*
by (*cases* *b*) *simp-all*

lemma *bitor-cancel* [*simp*]: (*b* *bitor* *b*) = *b*
by (*cases* *b*) *simp-all*

lemma *bitxor-cancel* [*simp*]: (*b* *bitxor* *b*) = **0**
by (*cases* *b*) *simp-all*

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 (*cases* *w*)
assume *w* = []
thus *?thesis* **by** (*rule* *empty*)
next
fix *b bs*
assume [*simp*]: *w* = *b* # *bs*
show *P w*
proof (*cases* *b*)
assume *b* = **0**
hence *w* = **0** # *bs* **by** *simp*

```
    thus ?thesis by (rule zero)
  next
    assume b = 1
    hence w = 1 # bs by simp
    thus ?thesis by (rule one)
  qed
qed

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 (induct w, simp-all add: empty)
  fix b bs
  assume P bs
  then show P (b#bs)
    by (cases b) (auto intro!: zero one)
qed

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
  by (simp add: bv-extend-def)

lemma bv-not-Nil [simp]: bv-not [] = []
  by (simp add: bv-not-def)

lemma bv-not-Cons [simp]: bv-not (b#bs) = (bitnot b) # bv-not bs
  by (simp add: bv-not-def)

lemma bv-not-bv-not [simp]: bv-not (bv-not w) = w
  by (rule bit-list-induct [of - w]) simp-all

lemma bv-msb-Nil [simp]: bv-msb [] = 0
  by (simp add: bv-msb-def)

lemma bv-msb-Cons [simp]: bv-msb (b#bs) = b
  by (simp add: bv-msb-def)
```

lemma *bv-msb-bv-not* [*simp*]: *0* < *length w* ==> *bv-msb* (*bv-not w*) = (*bitnot* (*bv-msb w*))
  by (*cases w*) *simp-all*

lemma *bv-msb-one-length* [*simp*,*intro*]: *bv-msb w* = **1** ==> *0* < *length w*
  by (*cases w*) *simp-all*

lemma *length-bv-not* [*simp*]: *length* (*bv-not w*) = *length w*
  by (*induct w*) *simp-all*

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*
  by (*simp add*: *bv-to-nat-def*)

lemma *bv-to-nat-helper* [*simp*]: *bv-to-nat* (*b* # *bs*) = *bitval b* * *2* ^ *length bs* + *bv-to-nat bs*
proof −
  let *?bv-to-nat'* = *foldl* (λ*bn b*. *2* * *bn* + *bitval b*)
  have *helper*: ⋀*base*. *?bv-to-nat' base bs* = *base* * *2* ^ *length bs* + *?bv-to-nat' 0 bs*
  proof (*induct bs*)
    case *Nil*
    show *?case* **by** *simp*
  next
    case (*Cons x xs base*)
    show *?case*
      apply (*simp only*: *foldl.simps*)
      apply (*subst Cons* [*of 2* * *base* + *bitval x*])
      apply *simp*
      apply (*subst Cons* [*of bitval x*])
      apply (*simp add*: *add-mult-distrib*)
      done
  qed
  show *?thesis* **by** (*simp add*: *bv-to-nat-def*) (*rule helper*)
qed

lemma *bv-to-nat0* [*simp*]: *bv-to-nat* (**0**#*bs*) = *bv-to-nat bs*
  by *simp*

lemma *bv-to-nat1* [*simp*]: *bv-to-nat* (**1**#*bs*) = *2* ^ *length bs* + *bv-to-nat bs*
  by *simp*

lemma *bv-to-nat-upper-range*: *bv-to-nat w* < *2* ^ *length w*
proof (*induct w*, *simp-all*)
  fix *b bs*

```
  assume bv-to-nat bs < 2 ^ length bs
  show bitval b * 2 ^ length bs + bv-to-nat bs < 2 * 2 ^ length bs
  proof (cases b, simp-all)
    have bv-to-nat bs < 2 ^ length bs by fact
    also have ... < 2 * 2 ^ length bs by auto
    finally show bv-to-nat bs < 2 * 2 ^ length bs by simp
  next
    have bv-to-nat bs < 2 ^ length bs by fact
    hence 2 ^ length bs + bv-to-nat bs < 2 ^ length bs + 2 ^ length bs by arith
    also have ... = 2 * (2 ^ length bs) by simp
    finally show bv-to-nat bs < 2 ^ length bs by simp
  qed
qed

lemma bv-extend-longer [simp]:
  assumes wn: n ≤ length w
  shows     bv-extend n b w = w
  by (simp add: bv-extend-def wn)

lemma bv-extend-shorter [simp]:
  assumes wn: length w < n
  shows     bv-extend n b w = bv-extend n b (b#w)
proof -
  from wn
  have s: n - Suc (length w) + 1 = n - length w
    by arith
  have bv-extend n b w = replicate (n - length w) b @ w
    by (simp add: bv-extend-def)
  also have ... = replicate (n - Suc (length w) + 1) b @ w
    by (subst s) rule
  also have ... = (replicate (n - Suc (length w)) b @ replicate 1 b) @ w
    by (subst replicate-add) rule
  also have ... = replicate (n - Suc (length w)) b @ b # w
    by simp
  also have ... = bv-extend n b (b#w)
    by (simp add: bv-extend-def)
  finally show bv-extend n b w = bv-extend n b (b#w) .
qed

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
  by (rule bit-list-induct [of - w],simp-all (no-asm),safe,simp-all)

lemma rem-initial-equal:
```

```
  assumes p: length (rem-initial b w) = length w
  shows      rem-initial b w = w
proof -
  have length (rem-initial b w) = length w --> rem-initial b w = w
  proof (induct w, simp-all, clarify)
    fix xs
    assume length (rem-initial b xs) = length xs --> rem-initial b xs = xs
    assume f: length (rem-initial b xs) = Suc (length xs)
    with rem-initial-length [of b xs]
    show rem-initial b xs = b#xs
      by auto
  qed
  from this and p show ?thesis ..
qed

lemma bv-extend-rem-initial: bv-extend (length w) b (rem-initial b w) = w
proof (induct w, simp-all, safe)
  fix xs
  assume ind: bv-extend (length xs) b (rem-initial b xs) = xs
  from rem-initial-length [of b xs]
  have [simp]: Suc (length xs) - length (rem-initial b xs) =
      1 + (length xs - length (rem-initial b xs))
    by arith
  have bv-extend (Suc (length xs)) b (rem-initial b xs) =
      replicate (Suc (length xs) - length (rem-initial b xs)) b @ (rem-initial b xs)
    by (simp add: bv-extend-def)
  also have ... =
      replicate (1 + (length xs - length (rem-initial b xs))) b @ rem-initial b xs
    by simp
  also have ... =
     (replicate 1 b @ replicate (length xs - length (rem-initial b xs)) b) @ rem-initial
b xs
    by (subst replicate-add) (rule refl)
  also have ... = b # bv-extend (length xs) b (rem-initial b xs)
    by (auto simp add: bv-extend-def [symmetric])
  also have ... = b # xs
    by (simp add: ind)
  finally show bv-extend (Suc (length xs)) b (rem-initial b xs) = b # xs .
qed

lemma rem-initial-append1:
  assumes rem-initial b xs ~= []
  shows   rem-initial b (xs @ ys) = rem-initial b xs @ ys
  using assms by (induct xs) auto

lemma rem-initial-append2:
  assumes rem-initial b xs = []
  shows   rem-initial b (xs @ ys) = rem-initial b ys
  using assms by (induct xs) auto
```

definition
norm-unsigned :: *bit list* => *bit list* **where**
norm-unsigned = *rem-initial* **0**

lemma *norm-unsigned-Nil* [*simp*]: *norm-unsigned* [] = []
by (*simp add*: *norm-unsigned-def*)

lemma *norm-unsigned-Cons0* [*simp*]: *norm-unsigned* (**0**#*bs*) = *norm-unsigned bs*
by (*simp add*: *norm-unsigned-def*)

lemma *norm-unsigned-Cons1* [*simp*]: *norm-unsigned* (**1**#*bs*) = **1**#*bs*
by (*simp add*: *norm-unsigned-def*)

lemma *norm-unsigned-idem* [*simp*]: *norm-unsigned* (*norm-unsigned w*) = *norm-unsigned w*
by (*rule bit-list-induct* [*of - w*],*simp-all*)

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* = []
by (*simp add*: *nat-to-bv-def*)

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 (*cases n* = *0*)
assume *n* = *0*
thus *R* **by** (*rule zero*)
next
assume *n* ~= *0*
hence *0* < *n* **by** *simp*
hence *n div 2* < *n* **by** *arith*
from *this* **and** ‹*0* < *n*› **show** *R* **by** (*rule div*)
qed

lemma *int-wf-ge-induct*:

```
  assumes ind : !!i::int. (!!j. [| k ≤ j ; j < i |] ==> P j) ==> P i
  shows       P i
proof (rule wf-induct-rule [OF wf-int-ge-less-than])
  fix x
  assume ih: (⋀y::int. (y, x) ∈ int-ge-less-than k ⟹ P y)
  thus P x
    by (rule ind) (simp add: int-ge-less-than-def)
qed

lemma unfold-nat-to-bv-helper:
  nat-to-bv-helper b l = nat-to-bv-helper b [] @ l
proof −
  have ∀ l. nat-to-bv-helper b l = nat-to-bv-helper b [] @ l
  proof (induct b rule: less-induct)
    fix n
    assume ind: !!j. j < n ⟹ ∀ l. nat-to-bv-helper j l = nat-to-bv-helper j [] @ l
    show ∀ l. nat-to-bv-helper n l = nat-to-bv-helper n [] @ l
    proof
      fix l
      show nat-to-bv-helper n l = nat-to-bv-helper n [] @ l
      proof (cases n < 0)
        assume n < 0
        thus ?thesis
          by (simp add: nat-to-bv-helper.simps)
      next
        assume ~n < 0
        show ?thesis
        proof (rule n-div-2-cases [of n])
          assume [simp]: n = 0
          show ?thesis
            apply (simp only: nat-to-bv-helper.simps [of n])
            apply simp
            done
        next
          assume n2n: n div 2 < n
          assume [simp]: 0 < n
          hence n20: 0 ≤ n div 2
            by arith
          from ind [of n div 2] and n2n n20
          have ind': ∀ l. nat-to-bv-helper (n div 2) l = nat-to-bv-helper (n div 2) []
@ l
            by blast
          show ?thesis
            apply (simp only: nat-to-bv-helper.simps [of n])
            apply (cases n=0)
            apply simp
            apply (simp only: if-False)
            apply simp
            apply (subst spec [OF ind',of 0#l])
```

```
        apply (subst spec [OF ind',of 1#l])
        apply (subst spec [OF ind',of [1]])
        apply (subst spec [OF ind',of [0]])
        apply simp
        done
      qed
    qed
  qed
 qed
 thus ?thesis ..
qed
```

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 -
 assume [simp]: n≠0
 show ?thesis
  apply (subst nat-to-bv-def [of n])
  apply (simp only: nat-to-bv-helper.simps [of n])
  apply (subst unfold-nat-to-bv-helper)
  using prems
  apply (simp)
  apply (subst nat-to-bv-def [of n div 2])
  apply auto
  done
qed
```

lemma *bv-to-nat-dist-append*:
 bv-to-nat (*l1* @ *l2*) = *bv-to-nat l1* ∗ *2* ˆ *length l2* + *bv-to-nat l2*

```
proof -
 have ∀ l2. bv-to-nat (l1 @ l2) = bv-to-nat l1 * 2 ^ length l2 + bv-to-nat l2
 proof (induct l1,simp-all)
  fix x xs
  assume ind: ∀ l2. bv-to-nat (xs @ l2) = bv-to-nat xs * 2 ^ length l2 + bv-to-nat l2
  show ∀ l2. bitval x * 2 ^ (length xs + length l2) + bv-to-nat xs * 2 ^ length l2 = (bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
  proof
   fix l2
   show bitval x * 2 ^ (length xs + length l2) + bv-to-nat xs * 2 ^ length l2 = (bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
   proof -
    have (2::nat) ^ (length xs + length l2) = 2 ^ length xs * 2 ^ length l2
     by (induct length xs,simp-all)
    hence bitval x * 2 ^ (length xs + length l2) + bv-to-nat xs * 2 ^ length l2 =
     bitval x * 2 ^ length xs * 2 ^ length l2 + bv-to-nat xs * 2 ^ length l2
     by simp
    also have ... = (bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
     by (simp add: ring-distribs)
```

```
      finally show ?thesis .
     qed
    qed
  qed
  thus ?thesis ..
qed

lemma bv-nat-bv [simp]: bv-to-nat (nat-to-bv n) = n
proof (induct n rule: less-induct)
  fix n
  assume ind: !!j. j < n ⟹ bv-to-nat (nat-to-bv j) = j
  show bv-to-nat (nat-to-bv n) = n
  proof (rule n-div-2-cases [of n])
    assume n = 0 then show ?thesis by simp
  next
    assume nn: n div 2 < n
    assume n0: 0 < n
    from ind and nn
    have ind': bv-to-nat (nat-to-bv (n div 2)) = n div 2 by blast
    from n0 have n0': n ≠ 0 by simp
    show ?thesis
      apply (subst nat-to-bv-def)
      apply (simp only: nat-to-bv-helper.simps [of n])
      apply (simp only: n0' if-False)
      apply (subst unfold-nat-to-bv-helper)
      apply (subst bv-to-nat-dist-append)
      apply (fold nat-to-bv-def)
      apply (simp add: ind' split del: split-if)
      apply (cases n mod 2 = 0)
      proof (simp-all)
        assume n mod 2 = 0
        with mod-div-equality [of n 2]
        show n div 2 * 2 = n by simp
      next
        assume n mod 2 = Suc 0
        with mod-div-equality [of n 2]
        show Suc (n div 2 * 2) = n by arith
      qed
  qed
qed

lemma bv-to-nat-type [simp]: bv-to-nat (norm-unsigned w) = bv-to-nat w
  by (rule bit-list-induct) simp-all

lemma length-norm-unsigned-le [simp]: length (norm-unsigned w) ≤ length w
  by (rule bit-list-induct) simp-all

lemma bv-to-nat-rew-msb: bv-msb w = 1 ==> bv-to-nat w = 2 ^ (length w - 1)
+ bv-to-nat (tl w)
```

by (*rule bit-list-cases* [*of w*]) *simp-all*

lemma *norm-unsigned-result*: *norm-unsigned xs* = [] ∨ *bv-msb* (*norm-unsigned xs*) = **1**
proof (*rule length-induct* [*of - xs*])
fix *xs* :: *bit list*
assume *ind*: ∀ *ys*. *length ys* < *length xs* −−> *norm-unsigned ys* = [] ∨ *bv-msb* (*norm-unsigned ys*) = **1**
show *norm-unsigned xs* = [] ∨ *bv-msb* (*norm-unsigned xs*) = **1**
proof (*rule bit-list-cases* [*of xs*],*simp-all*)
fix *bs*
assume [*simp*]: *xs* = **0**#*bs*
from *ind*
have *length bs* < *length xs* −−> *norm-unsigned bs* = [] ∨ *bv-msb* (*norm-unsigned bs*) = **1** **..**
thus *norm-unsigned bs* = [] ∨ *bv-msb* (*norm-unsigned bs*) = **1** **by** *simp*
qed
qed

lemma *norm-empty-bv-to-nat-zero*:
assumes *nw*: *norm-unsigned w* = []
shows *bv-to-nat w* = *0*
proof −
have *bv-to-nat w* = *bv-to-nat* (*norm-unsigned w*) **by** *simp*
also have ... = *bv-to-nat* [] **by** (*subst nw*) (*rule refl*)
also have ... = *0* **by** *simp*
finally show *?thesis* **.**
qed

lemma *bv-to-nat-lower-limit*:
assumes *w0*: *0* < *bv-to-nat w*
shows *2* ˆ (*length* (*norm-unsigned w*) − *1*) ≤ *bv-to-nat w*
proof −
from *w0* **and** *norm-unsigned-result* [*of w*]
have *msbw*: *bv-msb* (*norm-unsigned w*) = **1**
by (*auto simp add*: *norm-empty-bv-to-nat-zero*)
have *2* ˆ (*length* (*norm-unsigned w*) − *1*) ≤ *bv-to-nat* (*norm-unsigned w*)
by (*subst bv-to-nat-rew-msb* [*OF msbw*],*simp*)
thus *?thesis* **by** *simp*
qed

lemmas [*simp del*] = *nat-to-bv-non0*

lemma *norm-unsigned-length* [*intro*!]: *length* (*norm-unsigned w*) ≤ *length w*
by (*subst norm-unsigned-def*,*rule rem-initial-length*)

lemma *norm-unsigned-equal*:
length (*norm-unsigned w*) = *length w* ==> *norm-unsigned w* = *w*
by (*simp add*: *norm-unsigned-def*,*rule rem-initial-equal*)

lemma *bv-extend-norm-unsigned*: *bv-extend* (*length w*) **0** (*norm-unsigned w*) = *w*
by (*simp add*: *norm-unsigned-def*,*rule bv-extend-rem-initial*)

lemma *norm-unsigned-append1* [*simp*]:
  norm-unsigned xs ≠ [] ==> *norm-unsigned* (*xs* @ *ys*) = *norm-unsigned xs* @ *ys*
by (*simp add*: *norm-unsigned-def*,*rule rem-initial-append1*)

lemma *norm-unsigned-append2* [*simp*]:
  norm-unsigned xs = [] ==> *norm-unsigned* (*xs* @ *ys*) = *norm-unsigned ys*
by (*simp add*: *norm-unsigned-def*,*rule rem-initial-append2*)

lemma *bv-to-nat-zero-imp-empty*:
  bv-to-nat w = *0* ⟹ *norm-unsigned w* = []
by (*atomize* (*full*), *induct w rule*: *bit-list-induct*) *simp-all*

lemma *bv-to-nat-nzero-imp-nempty*:
  bv-to-nat w ≠ *0* ⟹ *norm-unsigned w* ≠ []
by (*induct w rule*: *bit-list-induct*) *simp-all*

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 (*cases x*)
  assume [*simp*]: *x* = **1**
  show *?thesis*
    apply (*simp add*: *nat-to-bv-non0*)
    apply *safe*
  proof −
    fix *q*
    assume *Suc* (*2* ∗ *bv-to-nat w*) = *2* ∗ *q*
    hence *orig*: (*2* ∗ *bv-to-nat w* + *1*) *mod 2* = *2* ∗ *q mod 2* (**is** *?lhs* = *?rhs*)
      by *simp*
    have *?lhs* = (*1* + *2* ∗ *bv-to-nat w*) *mod 2*
      by (*simp add*: *add-commute*)
    also have ... = *1*
      by (*subst mod-add1-eq*) *simp*
    finally have *eq1*: *?lhs* = *1* **.**
    have *?rhs* = *0* **by** *simp*
    with *orig* **and** *eq1*
    show *nat-to-bv* (*Suc* (*2* ∗ *bv-to-nat w*) *div 2*) @ [**0**] = *norm-unsigned* (*w* @ [**1**])
      by *simp*
  next
    have *nat-to-bv* (*Suc* (*2* ∗ *bv-to-nat w*) *div 2*) @ [**1**] =
        nat-to-bv ((*1* + *2* ∗ *bv-to-nat w*) *div 2*) @ [**1**]
      by (*simp add*: *add-commute*)
    also have ... = *nat-to-bv* (*bv-to-nat w*) @ [**1**]
      by (*subst div-add1-eq*) *simp*
    also have ... = *norm-unsigned w* @ [**1**]

```
    by (subst ass) (rule refl)
  also have ... = norm-unsigned (w @ [1])
    by (cases norm-unsigned w) simp-all
  finally show nat-to-bv (Suc (2 * bv-to-nat w) div 2) @ [1] = norm-unsigned
(w @ [1]) .
 qed
next
 assume [simp]: x = 0
 show ?thesis
 proof (cases bv-to-nat w = 0)
  assume bv-to-nat w = 0
  thus ?thesis
    by (simp add: bv-to-nat-zero-imp-empty)
 next
  assume bv-to-nat w ≠ 0
  thus ?thesis
    apply simp
    apply (subst nat-to-bv-non0)
    apply simp
    apply auto
    apply (subst ass)
    apply (cases norm-unsigned w)
    apply (simp-all add: norm-empty-bv-to-nat-zero)
    done
 qed
qed

lemma nat-helper2: nat-to-bv (2 ^ length xs + bv-to-nat xs) = 1 # xs
proof −
 have ∀xs. nat-to-bv (2 ^ length (rev xs) + bv-to-nat (rev xs)) = 1 # (rev xs)
(is ∀xs. ?P xs)
 proof
  fix xs
  show ?P xs
  proof (rule length-induct [of - xs])
    fix xs :: bit list
    assume ind: ∀ys. length ys < length xs −−> ?P ys
    show ?P xs
    proof (cases xs)
     assume xs = []
     then show ?thesis by (simp add: nat-to-bv-non0)
    next
     fix y ys
     assume [simp]: xs = y # ys
     show ?thesis
      apply simp
      apply (subst bv-to-nat-dist-append)
      apply simp
     proof −
```

```
        have nat-to-bv (2 * 2 ^ length ys + (bv-to-nat (rev ys) * 2 + bitval y)) =
          nat-to-bv (2 * (2 ^ length ys + bv-to-nat (rev ys)) + bitval y)
          by (simp add: add-ac mult-ac)
        also have ... = nat-to-bv (2 * (bv-to-nat (1#rev ys)) + bitval y)
          by simp
        also have ... = norm-unsigned (1#rev ys) @ [y]
        proof -
          from ind
          have nat-to-bv (2 ^ length (rev ys) + bv-to-nat (rev ys)) = 1 # rev ys
            by auto
          hence [simp]: nat-to-bv (2 ^ length ys + bv-to-nat (rev ys)) = 1 # rev ys
            by simp
          show ?thesis
            apply (subst nat-helper1)
            apply simp-all
            done
        qed
        also have ... = (1#rev ys) @ [y]
          by simp
        also have ... = 1 # rev ys @ [y]
          by simp
        finally show nat-to-bv (2 * 2 ^ length ys + (bv-to-nat (rev ys) * 2 +
bitval y)) =
          1 # rev ys @ [y] .
      qed
    qed
  qed
qed
hence nat-to-bv (2 ^ length (rev (rev xs)) + bv-to-nat (rev (rev xs))) =
    1 # rev (rev xs) ..
thus ?thesis by simp
qed

lemma nat-bv-nat [simp]: nat-to-bv (bv-to-nat w) = norm-unsigned w
proof (rule bit-list-induct [of - w],simp-all)
  fix xs
  assume nat-to-bv (bv-to-nat xs) = norm-unsigned xs
  have bv-to-nat xs = bv-to-nat (norm-unsigned xs) by simp
  have bv-to-nat xs < 2 ^ length xs
    by (rule bv-to-nat-upper-range)
  show nat-to-bv (2 ^ length xs + bv-to-nat xs) = 1 # xs
    by (rule nat-helper2)
qed

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 -
```

```
  from one
  have nat-to-bv (bv-to-nat xs) = nat-to-bv (bv-to-nat ys)
    by simp
  hence xsys: norm-unsigned xs = norm-unsigned ys
    by simp
  have xs = bv-extend (length xs) 0 (norm-unsigned xs)
    by (simp add: bv-extend-norm-unsigned)
  also have ... = bv-extend (length ys) 0 (norm-unsigned ys)
    by (simp add: xsys len)
  also have ... = ys
    by (simp add: bv-extend-norm-unsigned)
  finally show ?thesis .
qed

lemma norm-unsigned-nat-to-bv [simp]:
  norm-unsigned (nat-to-bv n) = nat-to-bv n
proof -
 have norm-unsigned (nat-to-bv n) = nat-to-bv (bv-to-nat (norm-unsigned (nat-to-bv
n)))
    by (subst nat-bv-nat) simp
  also have ... = nat-to-bv n by simp
  finally show ?thesis .
qed

lemma length-nat-to-bv-upper-limit:
  assumes nk: n ≤ 2 ^ k - 1
  shows      length (nat-to-bv n) ≤ k
proof (cases n = 0)
  case True
  thus ?thesis
    by (simp add: nat-to-bv-def nat-to-bv-helper.simps)
next
  case False
  hence n0: 0 < n by simp
  show ?thesis
  proof (rule ccontr)
    assume ~ length (nat-to-bv n) ≤ k
    hence k < length (nat-to-bv n) by simp
    hence k ≤ length (nat-to-bv n) - 1 by arith
    hence (2::nat) ^ k ≤ 2 ^ (length (nat-to-bv n) - 1) by simp
    also have ... = 2 ^ (length (norm-unsigned (nat-to-bv n)) - 1) by simp
    also have ... ≤ bv-to-nat (nat-to-bv n)
      by (rule bv-to-nat-lower-limit) (simp add: n0)
    also have ... = n by simp
    finally have 2 ^ k ≤ n .
    with n0 have 2 ^ k - 1 < n by arith
    with nk show False by simp
  qed
qed
```

```
lemma length-nat-to-bv-lower-limit:
  assumes nk: 2 ^ k ≤ n
  shows      k < length (nat-to-bv n)
proof (rule ccontr)
  assume ~ k < length (nat-to-bv n)
  hence lnk: length (nat-to-bv n) ≤ k by simp
  have n = bv-to-nat (nat-to-bv n) by simp
  also have ... < 2 ^ length (nat-to-bv n)
    by (rule bv-to-nat-upper-range)
  also from lnk have ... ≤ 2 ^ k by simp
  finally have n < 2 ^ k .
  with nk show False by simp
qed
```

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
  by (simp add: bv-add-def)

lemma bv-add-type2 [simp]: bv-add w1 (norm-unsigned w2) = bv-add w1 w2
  by (simp add: bv-add-def)

lemma bv-add-returntype [simp]: norm-unsigned (bv-add w1 w2) = bv-add w1 w2
  by (simp add: bv-add-def)

lemma bv-add-length: length (bv-add w1 w2) ≤ Suc (max (length w1) (length w2))
proof (unfold bv-add-def,rule length-nat-to-bv-upper-limit)
  from bv-to-nat-upper-range [of w1] and bv-to-nat-upper-range [of w2]
  have bv-to-nat w1 + bv-to-nat w2 ≤ (2 ^ length w1 − 1) + (2 ^ length w2 − 1)
    by arith
  also have ... ≤
      max (2 ^ length w1 − 1) (2 ^ length w2 − 1) + max (2 ^ length w1 − 1) (2
^ length w2 − 1)
    by (rule add-mono,safe intro!: le-maxI1 le-maxI2)
  also have ... = 2 ∗ max (2 ^ length w1 − 1) (2 ^ length w2 − 1) by simp
  also have ... ≤ 2 ^ Suc (max (length w1) (length w2)) − 2
  proof (cases length w1 ≤ length w2)
    assume w1w2: length w1 ≤ length w2
    hence (2::nat) ^ length w1 ≤ 2 ^ length w2 by simp
    hence (2::nat) ^ length w1 − 1 ≤ 2 ^ length w2 − 1 by arith
    with w1w2 show ?thesis
      by (simp add: diff-mult-distrib2 split: split-max)
  next
    assume [simp]: ~ (length w1 ≤ length w2)
```

```
    have ~ ((2::nat) ^ length w1 - 1 ≤ 2 ^ length w2 - 1)
    proof
      assume (2::nat) ^ length w1 - 1 ≤ 2 ^ length w2 - 1
      hence ((2::nat) ^ length w1 - 1) + 1 ≤ (2 ^ length w2 - 1) + 1
        by (rule add-right-mono)
      hence (2::nat) ^ length w1 ≤ 2 ^ length w2 by simp
      hence length w1 ≤ length w2 by simp
      thus False by simp
    qed
    thus ?thesis
      by (simp add: diff-mult-distrib2 split: split-max)
  qed
  finally show bv-to-nat w1 + bv-to-nat w2 ≤ 2 ^ Suc (max (length w1) (length
w2)) - 1
    by arith
qed

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
  by (simp add: bv-mult-def)

lemma bv-mult-type2 [simp]: bv-mult w1 (norm-unsigned w2) = bv-mult w1 w2
  by (simp add: bv-mult-def)

lemma bv-mult-returntype [simp]: norm-unsigned (bv-mult w1 w2) = bv-mult w1
w2
  by (simp add: bv-mult-def)

lemma bv-mult-length: length (bv-mult w1 w2) ≤ length w1 + length w2
proof (unfold bv-mult-def,rule length-nat-to-bv-upper-limit)
  from bv-to-nat-upper-range [of w1] and bv-to-nat-upper-range [of w2]
  have h: bv-to-nat w1 ≤ 2 ^ length w1 - 1 ∧ bv-to-nat w2 ≤ 2 ^ length w2 - 1
    by arith
  have bv-to-nat w1 * bv-to-nat w2 ≤ (2 ^ length w1 - 1) * (2 ^ length w2 - 1)
    apply (cut-tac h)
    apply (rule mult-mono)
    apply auto
    done
  also have ... < 2 ^ length w1 * 2 ^ length w2
    by (rule mult-strict-mono,auto)
  also have ... = 2 ^ (length w1 + length w2)
    by (simp add: power-add)
  finally show bv-to-nat w1 * bv-to-nat w2 ≤ 2 ^ (length w1 + length w2) - 1
    by arith
qed
```

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**] = []
  by *simp*

lemma *norm-signed1* [*simp*]: *norm-signed* [**1**] = [**1**]
  by *simp*

lemma *norm-signed01* [*simp*]: *norm-signed* (**0**#**1**#*xs*) = **0**#**1**#*xs*
  by *simp*

lemma *norm-signed00* [*simp*]: *norm-signed* (**0**#**0**#*xs*) = *norm-signed* (**0**#*xs*)
  by *simp*

lemma *norm-signed10* [*simp*]: *norm-signed* (**1**#**0**#*xs*) = **1**#**0**#*xs*
  by *simp*

lemma *norm-signed11* [*simp*]: *norm-signed* (**1**#**1**#*xs*) = *norm-signed* (**1**#*xs*)
  by *simp*

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*))
  by (*simp add*: *int-to-bv-def*)

lemma *int-to-bv-lt0* [*simp*]:
   $n < 0$ ==> *int-to-bv n* = *norm-signed* (*bv-not* (**0**#*nat-to-bv* (*nat* ($-n-$ *1*))))
  by (*simp add*: *int-to-bv-def*)

lemma *norm-signed-idem* [*simp*]: *norm-signed* (*norm-signed w*) = *norm-signed w*
proof (*rule bit-list-induct* [*of - w*], *simp-all*)
  fix *xs*
  assume *eq*: *norm-signed* (*norm-signed xs*) = *norm-signed xs*
  show *norm-signed* (*norm-signed* (**0**#*xs*)) = *norm-signed* (**0**#*xs*)

```
  proof (rule bit-list-cases [of xs],simp-all)
    fix ys
    assume xs = 0#ys
    from this [symmetric] and eq
    show norm-signed (norm-signed (0#ys)) = norm-signed (0#ys)
      by simp
  qed
next
  fix xs
  assume eq: norm-signed (norm-signed xs) = norm-signed xs
  show norm-signed (norm-signed (1#xs)) = norm-signed (1#xs)
  proof (rule bit-list-cases [of xs],simp-all)
    fix ys
    assume xs = 1#ys
    from this [symmetric] and eq
    show norm-signed (norm-signed (1#ys)) = norm-signed (1#ys)
      by simp
  qed
qed

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
  by (simp add: bv-to-int-def)

lemma bv-to-int-Cons0 [simp]: bv-to-int (0#bs) = int (bv-to-nat bs)
  by (simp add: bv-to-int-def)

lemma bv-to-int-Cons1 [simp]: bv-to-int (1#bs) = − int (bv-to-nat (bv-not bs) +
1)
  by (simp add: bv-to-int-def)

lemma bv-to-int-type [simp]: bv-to-int (norm-signed w) = bv-to-int w
proof (rule bit-list-induct [of - w], simp-all)
  fix xs
  assume ind: bv-to-int (norm-signed xs) = bv-to-int xs
  show bv-to-int (norm-signed (0#xs)) = int (bv-to-nat xs)
  proof (rule bit-list-cases [of xs], simp-all)
    fix ys
    assume [simp]: xs = 0#ys
    from ind
    show bv-to-int (norm-signed (0#ys)) = int (bv-to-nat ys)
      by simp
  qed
next
```

```
  fix xs
  assume ind: bv-to-int (norm-signed xs) = bv-to-int xs
  show bv-to-int (norm-signed (1#xs)) = −1 − int (bv-to-nat (bv-not xs))
  proof (rule bit-list-cases [of xs], simp-all)
    fix ys
    assume [simp]: xs = 1#ys
    from ind
    show bv-to-int (norm-signed (1#ys)) = −1 − int (bv-to-nat (bv-not ys))
      by simp
  qed
qed

lemma bv-to-int-upper-range: bv-to-int w < 2 ^ (length w − 1)
proof (rule bit-list-cases [of w],simp-all)
  fix bs
  from bv-to-nat-upper-range
  show int (bv-to-nat bs) < 2 ^ length bs
    by (simp add: int-nat-two-exp)
next
  fix bs
  have −1 − int (bv-to-nat (bv-not bs)) ≤ 0 by simp
  also have ... < 2 ^ length bs by (induct bs) simp-all
  finally show −1 − int (bv-to-nat (bv-not bs)) < 2 ^ length bs .
qed

lemma bv-to-int-lower-range: − (2 ^ (length w − 1)) ≤ bv-to-int w
proof (rule bit-list-cases [of w],simp-all)
  fix bs :: bit list
  have − (2 ^ length bs) ≤ (0::int) by (induct bs) simp-all
  also have ... ≤ int (bv-to-nat bs) by simp
  finally show − (2 ^ length bs) ≤ int (bv-to-nat bs) .
next
  fix bs
  from bv-to-nat-upper-range [of bv-not bs]
  show − (2 ^ length bs) ≤ −1 − int (bv-to-nat (bv-not bs))
    by (simp add: int-nat-two-exp)
qed

lemma int-bv-int [simp]: int-to-bv (bv-to-int w) = norm-signed w
proof (rule bit-list-cases [of w],simp)
  fix xs
  assume [simp]: w = 0#xs
  show ?thesis
    apply simp
    apply (subst norm-signed-Cons [of 0 xs])
    apply simp
    using norm-unsigned-result [of xs]
    apply safe
    apply (rule bit-list-cases [of norm-unsigned xs])
```

```
    apply simp-all
    done
next
  fix xs
  assume [simp]: w = 1#xs
  show ?thesis
    apply (simp del: int-to-bv-lt0)
    apply (rule bit-list-induct [of - xs])
    apply simp
    apply (subst int-to-bv-lt0)
    apply (subgoal-tac − int (bv-to-nat (bv-not (0 # bs))) + −1 < 0 + 0)
    apply simp
    apply (rule add-le-less-mono)
    apply simp
    apply simp
    apply (simp del: bv-to-nat1 bv-to-nat-helper)
    apply simp
    done
qed

lemma bv-int-bv [simp]: bv-to-int (int-to-bv i) = i
  by (cases 0 ≤ i) simp-all

lemma bv-msb-norm [simp]: bv-msb (norm-signed w) = bv-msb w
  by (rule bit-list-cases [of w]) (simp-all add: norm-signed-Cons)

lemma norm-signed-length: length (norm-signed w) ≤ length w
  apply (cases w, simp-all)
  apply (subst norm-signed-Cons)
  apply (case-tac a, simp-all)
  apply (rule rem-initial-length)
  done

lemma norm-signed-equal: length (norm-signed w) = length w ==> norm-signed
w = w
proof (rule bit-list-cases [of w], simp-all)
  fix xs
  assume length (norm-signed (0#xs)) = Suc (length xs)
  thus norm-signed (0#xs) = 0#xs
    apply (simp add: norm-signed-Cons)
    apply safe
    apply simp-all
    apply (rule norm-unsigned-equal)
    apply assumption
    done
next
  fix xs
  assume length (norm-signed (1#xs)) = Suc (length xs)
  thus norm-signed (1#xs) = 1#xs
```

```
    apply (simp add: norm-signed-Cons)
    apply (rule rem-initial-equal)
    apply assumption
    done
qed

lemma bv-extend-norm-signed: bv-msb w = b ==> bv-extend (length w) b (norm-signed
w) = w
proof (rule bit-list-cases [of w],simp-all)
  fix xs
  show bv-extend (Suc (length xs)) 0 (norm-signed (0#xs)) = 0#xs
  proof (simp add: norm-signed-list-def,auto)
    assume norm-unsigned xs = []
    hence xx: rem-initial 0 xs = []
      by (simp add: norm-unsigned-def)
    have bv-extend (Suc (length xs)) 0 (0#rem-initial 0 xs) = 0#xs
      apply (simp add: bv-extend-def replicate-app-Cons-same)
      apply (fold bv-extend-def)
      apply (rule bv-extend-rem-initial)
      done
    thus bv-extend (Suc (length xs)) 0 [0] = 0#xs
      by (simp add: xx)
  next
    show bv-extend (Suc (length xs)) 0 (0#norm-unsigned xs) = 0#xs
      apply (simp add: norm-unsigned-def)
      apply (simp add: bv-extend-def replicate-app-Cons-same)
      apply (fold bv-extend-def)
      apply (rule bv-extend-rem-initial)
      done
  qed
next
  fix xs
  show bv-extend (Suc (length xs)) 1 (norm-signed (1#xs)) = 1#xs
    apply (simp add: norm-signed-Cons)
    apply (simp add: bv-extend-def replicate-app-Cons-same)
    apply (fold bv-extend-def)
    apply (rule bv-extend-rem-initial)
    done
qed

lemma bv-to-int-qinj:
  assumes one: bv-to-int xs = bv-to-int ys
  and     len: length xs = length ys
  shows        xs = ys
proof -
  from one
  have int-to-bv (bv-to-int xs) = int-to-bv (bv-to-int ys) by simp
  hence xsys: norm-signed xs = norm-signed ys by simp
  hence xsys': bv-msb xs = bv-msb ys
```

```
  proof -
    have bv-msb xs = bv-msb (norm-signed xs) by simp
    also have ... = bv-msb (norm-signed ys) by (simp add: xsys)
    also have ... = bv-msb ys by simp
    finally show ?thesis .
  qed
  have xs = bv-extend (length xs) (bv-msb xs) (norm-signed xs)
    by (simp add: bv-extend-norm-signed)
  also have ... = bv-extend (length ys) (bv-msb ys) (norm-signed ys)
    by (simp add: xsys xsys' len)
  also have ... = ys
    by (simp add: bv-extend-norm-signed)
  finally show ?thesis .
qed

lemma int-to-bv-returntype [simp]: norm-signed (int-to-bv w) = int-to-bv w
  by (simp add: int-to-bv-def)

lemma bv-to-int-msb0: 0 ≤ bv-to-int w1 ==> bv-msb w1 = 0
  by (rule bit-list-cases,simp-all)

lemma bv-to-int-msb1: bv-to-int w1 < 0 ==> bv-msb w1 = 1
  by (rule bit-list-cases,simp-all)

lemma bv-to-int-lower-limit-gt0:
  assumes w0: 0 < bv-to-int w
  shows      2 ^ (length (norm-signed w) - 2) ≤ bv-to-int w
proof -
  from w0
  have 0 ≤ bv-to-int w by simp
  hence [simp]: bv-msb w = 0 by (rule bv-to-int-msb0)
  have 2 ^ (length (norm-signed w) - 2) ≤ bv-to-int (norm-signed w)
  proof (rule bit-list-cases [of w])
    assume w = []
    with w0 show ?thesis by simp
  next
    fix w'
    assume weq: w = 0 # w'
    thus ?thesis
    proof (simp add: norm-signed-Cons,safe)
      assume norm-unsigned w' = []
      with weq and w0 show False
        by (simp add: norm-empty-bv-to-nat-zero)
    next
      assume w'0: norm-unsigned w' ≠ []
      have 0 < bv-to-nat w'
      proof (rule ccontr)
        assume ~ (0 < bv-to-nat w')
        hence bv-to-nat w' = 0
```

```
          by arith
        hence norm-unsigned w' = []
          by (simp add: bv-to-nat-zero-imp-empty)
        with w'0
        show False by simp
      qed
      with bv-to-nat-lower-limit [of w']
      show 2 ^ (length (norm-unsigned w') − Suc 0) ≤ int (bv-to-nat w')
        by (simp add: int-nat-two-exp)
    qed
  next
    fix w'
    assume w = 1 # w'
    from w0 have bv-msb w = 0 by simp
    with prems show ?thesis by simp
  qed
  also have ... = bv-to-int w by simp
  finally show ?thesis .
qed

lemma norm-signed-result: norm-signed w = [] ∨ norm-signed w = [1] ∨ bv-msb
(norm-signed w) ≠ bv-msb (tl (norm-signed w))
  apply (rule bit-list-cases [of w],simp-all)
  apply (case-tac bs,simp-all)
  apply (case-tac a,simp-all)
  apply (simp add: norm-signed-Cons)
  apply safe
  apply simp
proof −
  fix l
  assume msb: 0 = bv-msb (norm-unsigned l)
  assume norm-unsigned l ≠ []
  with norm-unsigned-result [of l]
  have bv-msb (norm-unsigned l) = 1 by simp
  with msb show False by simp
next
  fix xs
  assume p: 1 = bv-msb (tl (norm-signed (1 # xs)))
  have 1 ≠ bv-msb (tl (norm-signed (1 # xs)))
    by (rule bit-list-induct [of - xs],simp-all)
  with p show False by simp
qed

lemma bv-to-int-upper-limit-lem1:
  assumes w0: bv-to-int w < −1
  shows       bv-to-int w < − (2 ^ (length (norm-signed w) − 2))
proof −
  from w0
  have bv-to-int w < 0 by simp
```

```
hence msbw [simp]: bv-msb w = 1
  by (rule bv-to-int-msb1)
have bv-to-int w = bv-to-int (norm-signed w) by simp
also from norm-signed-result [of w]
have ... < − (2 ^ (length (norm-signed w) − 2))
proof safe
  assume norm-signed w = []
  hence bv-to-int (norm-signed w) = 0 by simp
  with w0 show ?thesis by simp
next
  assume norm-signed w = [1]
  hence bv-to-int (norm-signed w) = −1 by simp
  with w0 show ?thesis by simp
next
  assume bv-msb (norm-signed w) ≠ bv-msb (tl (norm-signed w))
  hence msb-tl: 1 ≠ bv-msb (tl (norm-signed w)) by simp
  show bv-to-int (norm-signed w) < − (2 ^ (length (norm-signed w) − 2))
  proof (rule bit-list-cases [of norm-signed w])
    assume norm-signed w = []
    hence bv-to-int (norm-signed w) = 0 by simp
    with w0 show ?thesis by simp
  next
    fix w′
    assume nw: norm-signed w = 0 # w′
    from msbw have bv-msb (norm-signed w) = 1 by simp
    with nw show ?thesis by simp
  next
    fix w′
    assume weq: norm-signed w = 1 # w′
    show ?thesis
    proof (rule bit-list-cases [of w′])
      assume w′eq: w′ = []
      from w0 have bv-to-int (norm-signed w) < −1 by simp
      with w′eq and weq show ?thesis by simp
    next
      fix w″
      assume w′eq: w′ = 0 # w″
      show ?thesis
        apply (simp add: weq w′eq)
        apply (subgoal-tac − int (bv-to-nat (bv-not w″)) + −1 < 0 + 0)
        apply (simp add: int-nat-two-exp)
        apply (rule add-le-less-mono)
        apply simp-all
        done
    next
      fix w″
      assume w′eq: w′ = 1 # w″
      with weq and msb-tl show ?thesis by simp
    qed
```

```
    qed
  qed
  finally show ?thesis .
qed

lemma length-int-to-bv-upper-limit-gt0:
  assumes w0: 0 < i
  and     wk: i ≤ 2 ^ (k − 1) − 1
  shows       length (int-to-bv i) ≤ k
proof (rule ccontr)
  from w0 wk
  have k1: 1 < k
    by (cases k − 1,simp-all)
  assume ~ length (int-to-bv i) ≤ k
  hence k < length (int-to-bv i) by simp
  hence k ≤ length (int-to-bv i) − 1 by arith
  hence a: k − 1 ≤ length (int-to-bv i) − 2 by arith
  hence (2::int) ^ (k − 1) ≤ 2 ^ (length (int-to-bv i) − 2) by simp
  also have ... ≤ i
  proof −
    have 2 ^ (length (norm-signed (int-to-bv i)) − 2) ≤ bv-to-int (int-to-bv i)
    proof (rule bv-to-int-lower-limit-gt0)
      from w0 show 0 < bv-to-int (int-to-bv i) by simp
    qed
    thus ?thesis by simp
  qed
  finally have 2 ^ (k − 1) ≤ i .
  with wk show False by simp
qed

lemma pos-length-pos:
  assumes i0: 0 < bv-to-int w
  shows       0 < length w
proof −
  from norm-signed-result [of w]
  have 0 < length (norm-signed w)
  proof (auto)
    assume ii: norm-signed w = []
    have bv-to-int (norm-signed w) = 0 by (subst ii) simp
    hence bv-to-int w = 0 by simp
    with i0 show False by simp
  next
    assume ii: norm-signed w = []
    assume jj: bv-msb w ≠ 0
    have 0 = bv-msb (norm-signed w)
      by (subst ii) simp
    also have ... ≠ 0
      by (simp add: jj)
    finally show False by simp
```

```
  qed
  also have ... ≤ length w
    by (rule norm-signed-length)
  finally show ?thesis .
qed

lemma neg-length-pos:
  assumes i0: bv-to-int w < −1
  shows      0 < length w
proof −
  from norm-signed-result [of w]
  have 0 < length (norm-signed w)
  proof (auto)
    assume ii: norm-signed w = []
    have bv-to-int (norm-signed w) = 0
      by (subst ii) simp
    hence bv-to-int w = 0 by simp
    with i0 show False by simp
  next
    assume ii: norm-signed w = []
    assume jj: bv-msb w ≠ 0
    have 0 = bv-msb (norm-signed w) by (subst ii) simp
    also have ... ≠ 0 by (simp add: jj)
    finally show False by simp
  qed
  also have ... ≤ length w
    by (rule norm-signed-length)
  finally show ?thesis .
qed

lemma length-int-to-bv-lower-limit-gt0:
  assumes wk: 2 ^ (k − 1) ≤ i
  shows      k < length (int-to-bv i)
proof (rule ccontr)
  have 0 < (2::int) ^ (k − 1)
    by (rule zero-less-power) simp
  also have ... ≤ i by (rule wk)
  finally have i0: 0 < i .
  have lii0: 0 < length (int-to-bv i)
    apply (rule pos-length-pos)
    apply (simp,rule i0)
    done
  assume ~ k < length (int-to-bv i)
  hence length (int-to-bv i) ≤ k by simp
  with lii0
  have a: length (int-to-bv i) − 1 ≤ k − 1
    by arith
  have i < 2 ^ (length (int-to-bv i) − 1)
  proof −
```

```
    have i = bv-to-int (int-to-bv i)
      by simp
    also have ... < 2 ^ (length (int-to-bv i) - 1)
      by (rule bv-to-int-upper-range)
    finally show ?thesis .
  qed
  also have (2::int) ^ (length (int-to-bv i) - 1) ≤ 2 ^ (k - 1) using a
    by simp
  finally have i < 2 ^ (k - 1) .
  with wk show False by simp
qed

lemma length-int-to-bv-upper-limit-lem1:
  assumes w1: i < -1
  and     wk: - (2 ^ (k - 1)) ≤ i
  shows       length (int-to-bv i) ≤ k
proof (rule ccontr)
  from w1 wk
  have k1: 1 < k by (cases k - 1) simp-all
  assume ~ length (int-to-bv i) ≤ k
  hence k < length (int-to-bv i) by simp
  hence k ≤ length (int-to-bv i) - 1 by arith
  hence a: k - 1 ≤ length (int-to-bv i) - 2 by arith
  have i < - (2 ^ (length (int-to-bv i) - 2))
  proof -
    have i = bv-to-int (int-to-bv i)
      by simp
    also have ... < - (2 ^ (length (norm-signed (int-to-bv i)) - 2))
      by (rule bv-to-int-upper-limit-lem1,simp,rule w1)
    finally show ?thesis by simp
  qed
  also have ... ≤ -(2 ^ (k - 1))
  proof -
    have (2::int) ^ (k - 1) ≤ 2 ^ (length (int-to-bv i) - 2) using a by simp
    thus ?thesis by simp
  qed
  finally have i < -(2 ^ (k - 1)) .
  with wk show False by simp
qed

lemma length-int-to-bv-lower-limit-lem1:
  assumes wk: i < -(2 ^ (k - 1))
  shows       k < length (int-to-bv i)
proof (rule ccontr)
  from wk have i ≤ -(2 ^ (k - 1)) - 1 by simp
  also have ... < -1
  proof -
    have 0 < (2::int) ^ (k - 1)
      by (rule zero-less-power) simp
```

```
    hence −((2::int) ^ (k − 1)) < 0 by simp
    thus ?thesis by simp
  qed
  finally have i1: i < −1 .
  have lii0: 0 < length (int-to-bv i)
    apply (rule neg-length-pos)
    apply (simp, rule i1)
    done
  assume ~ k < length (int-to-bv i)
  hence length (int-to-bv i) ≤ k
    by simp
  with lii0 have a: length (int-to-bv i) − 1 ≤ k − 1 by arith
  hence (2::int) ^ (length (int-to-bv i) − 1) ≤ 2 ^ (k − 1) by simp
  hence −((2::int) ^ (k − 1)) ≤ − (2 ^ (length (int-to-bv i) − 1)) by simp
  also have ... ≤ i
  proof −
    have − (2 ^ (length (int-to-bv i) − 1)) ≤ bv-to-int (int-to-bv i)
      by (rule bv-to-int-lower-range)
    also have ... = i
      by simp
    finally show ?thesis .
  qed
  finally have −(2 ^ (k − 1)) ≤ i .
  with wk show False by simp
qed
```

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
  by (simp add: utos-def norm-signed-Cons)

lemma utos-returntype [simp]: norm-signed (utos w) = utos w
  by (simp add: utos-def)

lemma utos-length: length (utos w) ≤ Suc (length w)
  by (simp add: utos-def norm-signed-Cons)

lemma bv-to-int-utos: bv-to-int (utos w) = int (bv-to-nat w)
proof (simp add: utos-def norm-signed-Cons, safe)
  assume norm-unsigned w = []
  hence bv-to-nat (norm-unsigned w) = 0 by simp
  thus bv-to-nat w = 0 by simp
qed
```

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*
by (*simp add*: *bv-uminus-def*)

lemma *bv-uminus-returntype* [*simp*]: *norm-signed* (*bv-uminus w*) = *bv-uminus w*
by (*simp add*: *bv-uminus-def*)

lemma *bv-uminus-length*: *length* (*bv-uminus w*) ≤ *Suc* (*length w*)
proof −
have *1* < −*bv-to-int w* ∨ −*bv-to-int w* = *1* ∨ −*bv-to-int w* = *0* ∨ −*bv-to-int w* = −*1* ∨ −*bv-to-int w* < −*1*
by *arith*
thus *?thesis*
proof *safe*
assume *p*: *1* < − *bv-to-int w*
have *lw*: *0* < *length w*
apply (*rule neg-length-pos*)
using *p*
apply *simp*
done
show *?thesis*
proof (*simp add*: *bv-uminus-def*,*rule length-int-to-bv-upper-limit-gt0*,*simp-all*)
from *prems* **show** *bv-to-int w* < *0* **by** *simp*
next
have −(*2*ˆ(*length w* − *1*)) ≤ *bv-to-int w*
by (*rule bv-to-int-lower-range*)
hence − *bv-to-int w* ≤ *2*ˆ(*length w* − *1*) **by** *simp*
also from *lw* **have** ... < *2* ˆ *length w* **by** *simp*
finally show − *bv-to-int w* < *2* ˆ *length w* **by** *simp*
qed
next
assume *p*: − *bv-to-int w* = *1*
hence *lw*: *0* < *length w* **by** (*cases w*) *simp-all*
from *p*
show *?thesis*
apply (*simp add*: *bv-uminus-def*)
using *lw*
apply (*simp* (*no-asm*) *add*: *nat-to-bv-non0*)
done
next
assume − *bv-to-int w* = *0*
thus *?thesis* **by** (*simp add*: *bv-uminus-def*)
next
assume *p*: − *bv-to-int w* = −*1*
thus *?thesis* **by** (*simp add*: *bv-uminus-def*)

```
  next
    assume p: − bv-to-int w < −1
    show ?thesis
      apply (simp add: bv-uminus-def)
      apply (rule length-int-to-bv-upper-limit-lem1)
      apply (rule p)
      apply simp
    proof −
      have bv-to-int w < 2 ^ (length w − 1)
        by (rule bv-to-int-upper-range)
      also have ... ≤ 2 ^ length w by simp
      finally show bv-to-int w ≤ 2 ^ length w by simp
    qed
  qed
qed

lemma bv-uminus-length-utos: length (bv-uminus (utos w)) ≤ Suc (length w)
proof −
  have −bv-to-int (utos w) = 0 ∨ −bv-to-int (utos w) = −1 ∨ −bv-to-int (utos
w) < −1
    by (simp add: bv-to-int-utos, arith)
  thus ?thesis
  proof safe
    assume −bv-to-int (utos w) = 0
    thus ?thesis by (simp add: bv-uminus-def)
  next
    assume −bv-to-int (utos w) = −1
    thus ?thesis by (simp add: bv-uminus-def)
  next
    assume p: −bv-to-int (utos w) < −1
    show ?thesis
      apply (simp add: bv-uminus-def)
      apply (rule length-int-to-bv-upper-limit-lem1)
      apply (rule p)
      apply (simp add: bv-to-int-utos)
      using bv-to-nat-upper-range [of w]
      apply (simp add: int-nat-two-exp)
      done
  qed
qed

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
  by (simp add: bv-sadd-def)

lemma bv-sadd-type2 [simp]: bv-sadd w1 (norm-signed w2) = bv-sadd w1 w2
```

```
  by (simp add: bv-sadd-def)

lemma bv-sadd-returntype [simp]: norm-signed (bv-sadd w1 w2) = bv-sadd w1 w2
  by (simp add: bv-sadd-def)

lemma adder-helper:
  assumes lw: 0 < max (length w1) (length w2)
  shows  ((2::int) ^ (length w1 - 1)) + (2 ^ (length w2 - 1)) ≤ 2 ^ max (length
w1) (length w2)
proof -
  have ((2::int) ^ (length w1 - 1)) + (2 ^ (length w2 - 1)) ≤
      2 ^ (max (length w1) (length w2) - 1) + 2 ^ (max (length w1) (length w2)
- 1)
    apply (cases length w1 ≤ length w2)
    apply (auto simp add: max-def)
    done
  also have ... = 2 ^ max (length w1) (length w2)
  proof -
    from lw
    show ?thesis
      apply simp
      apply (subst power-Suc [symmetric])
      apply (simp del: power.simps)
      done
  qed
  finally show ?thesis .
qed

lemma bv-sadd-length: length (bv-sadd w1 w2) ≤ Suc (max (length w1) (length
w2))
proof -
  let ?Q = bv-to-int w1 + bv-to-int w2

  have helper: ?Q ≠ 0 ==> 0 < max (length w1) (length w2)
  proof -
    assume p: ?Q ≠ 0
    show 0 < max (length w1) (length w2)
    proof (simp add: less-max-iff-disj,rule)
      assume [simp]: w1 = []
      show w2 ≠ []
      proof (rule ccontr,simp)
        assume [simp]: w2 = []
        from p show False by simp
      qed
    qed
  qed

  have 0 < ?Q ∨ ?Q = 0 ∨ ?Q = -1 ∨ ?Q < -1 by arith
  thus ?thesis
```

```
  proof safe
    assume ?Q = 0
    thus ?thesis
      by (simp add: bv-sadd-def)
  next
    assume ?Q = −1
    thus ?thesis
      by (simp add: bv-sadd-def)
  next
    assume p: 0 < ?Q
    show ?thesis
      apply (simp add: bv-sadd-def)
      apply (rule length-int-to-bv-upper-limit-gt0)
      apply (rule p)
    proof simp
      from bv-to-int-upper-range [of w2]
      have bv-to-int w2 ≤ 2 ^ (length w2 − 1)
        by simp
      with bv-to-int-upper-range [of w1]
      have bv-to-int w1 + bv-to-int w2 < (2 ^ (length w1 − 1)) + (2 ^ (length w2
− 1))
        by (rule zadd-zless-mono)
      also have ... ≤ 2 ^ max (length w1) (length w2)
        apply (rule adder-helper)
        apply (rule helper)
        using p
        apply simp
        done
      finally show ?Q < 2 ^ max (length w1) (length w2) .
    qed
  next
    assume p: ?Q < −1
    show ?thesis
      apply (simp add: bv-sadd-def)
      apply (rule length-int-to-bv-upper-limit-lem1,simp-all)
      apply (rule p)
    proof −
      have (2 ^ (length w1 − 1)) + 2 ^ (length w2 − 1) ≤ (2::int) ^ max (length
w1) (length w2)
        apply (rule adder-helper)
        apply (rule helper)
        using p
        apply simp
        done
      hence −((2::int) ^ max (length w1) (length w2)) ≤ − (2 ^ (length w1 − 1))
+ −(2 ^ (length w2 − 1))
        by simp
      also have − (2 ^ (length w1 − 1)) + −(2 ^ (length w2 − 1)) ≤ ?Q
        apply (rule add-mono)
```

```
      apply (rule bv-to-int-lower-range [of w1])
      apply (rule bv-to-int-lower-range [of w2])
      done
    finally show − (2^max (length w1) (length w2)) ≤ ?Q .
  qed
 qed
qed

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
 by (simp add: bv-sub-def)

lemma bv-sub-type2 [simp]: bv-sub w1 (norm-signed w2) = bv-sub w1 w2
 by (simp add: bv-sub-def)

lemma bv-sub-returntype [simp]: norm-signed (bv-sub w1 w2) = bv-sub w1 w2
 by (simp add: bv-sub-def)

lemma bv-sub-length: length (bv-sub w1 w2) ≤ Suc (max (length w1) (length w2))
proof (cases bv-to-int w2 = 0)
 assume p: bv-to-int w2 = 0
 show ?thesis
 proof (simp add: bv-sub-def bv-sadd-def bv-uminus-def p)
  have length (norm-signed w1) ≤ length w1
   by (rule norm-signed-length)
  also have ... ≤ max (length w1) (length w2)
   by (rule le-maxI1)
  also have ... ≤ Suc (max (length w1) (length w2))
   by arith
  finally show length (norm-signed w1) ≤ Suc (max (length w1) (length w2)) .
 qed
next
 assume bv-to-int w2 ≠ 0
 hence 0 < length w2 by (cases w2,simp-all)
 hence lmw: 0 < max (length w1) (length w2) by arith

 let ?Q = bv-to-int w1 − bv-to-int w2

 have 0 < ?Q ∨ ?Q = 0 ∨ ?Q = −1 ∨ ?Q < −1 by arith
 thus ?thesis
 proof safe
  assume ?Q = 0
  thus ?thesis
   by (simp add: bv-sub-def bv-sadd-def bv-uminus-def)
 next
  assume ?Q = −1
```

thus *?thesis*
by (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
next
assume *p*: *0* < *?Q*
show *?thesis*
apply (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
apply (*rule length-int-to-bv-upper-limit-gt0*)
apply (*rule p*)
proof *simp*
from *bv-to-int-lower-range* [*of w2*]
have *v2*: − *bv-to-int w2* ≤ *2* ˆ (*length w2* − *1*) **by** *simp*
have *bv-to-int w1* + − *bv-to-int w2* < (*2* ˆ (*length w1* − *1*)) + (*2* ˆ (*length w2* − *1*))
apply (*rule zadd-zless-mono*)
apply (*rule bv-to-int-upper-range* [*of w1*])
apply (*rule v2*)
done
also have ... ≤ *2* ˆ *max* (*length w1*) (*length w2*)
apply (*rule adder-helper*)
apply (*rule lmw*)
done
finally show *?Q* < *2* ˆ *max* (*length w1*) (*length w2*) **by** *simp*
qed
next
assume *p*: *?Q* < −*1*
show *?thesis*
apply (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
apply (*rule length-int-to-bv-upper-limit-lem1*)
apply (*rule p*)
proof *simp*
have (*2* ˆ (*length w1* − *1*)) + *2* ˆ (*length w2* − *1*) ≤ (*2*::*int*) ˆ *max* (*length w1*) (*length w2*)
apply (*rule adder-helper*)
apply (*rule lmw*)
done
hence −((*2*::*int*) ˆ *max* (*length w1*) (*length w2*)) ≤ − (*2* ˆ (*length w1* − *1*)) + −(*2* ˆ (*length w2* − *1*))
by *simp*
also have − (*2* ˆ (*length w1* − *1*)) + −(*2* ˆ (*length w2* − *1*)) ≤ *bv-to-int w1* + −*bv-to-int w2*
apply (*rule add-mono*)
apply (*rule bv-to-int-lower-range* [*of w1*])
using *bv-to-int-upper-range* [*of w2*]
apply *simp*
done
finally show − (*2*ˆ*max* (*length w1*) (*length w2*)) ≤ *?Q* **by** *simp*
qed
qed
qed

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*
by (*simp add*: *bv-smult-def*)

lemma *bv-smult-type2* [*simp*]: *bv-smult w1* (*norm-signed w2*) = *bv-smult w1 w2*
by (*simp add*: *bv-smult-def*)

lemma *bv-smult-returntype* [*simp*]: *norm-signed* (*bv-smult w1 w2*) = *bv-smult w1 w2*
by (*simp add*: *bv-smult-def*)

lemma *bv-smult-length*: *length* (*bv-smult w1 w2*) ≤ *length w1* + *length w2*
proof −
let *?Q* = *bv-to-int w1* ∗ *bv-to-int w2*

have *lmw*: *?Q* ≠ *0* ==> *0* < *length w1* ∧ *0* < *length w2* **by** *auto*

have *0* < *?Q* ∨ *?Q* = *0* ∨ *?Q* = −*1* ∨ *?Q* < −*1* **by** *arith*
thus *?thesis*
proof (*safe dest*!: *iffD1* [*OF mult-eq-0-iff*])
assume *bv-to-int w1* = *0*
thus *?thesis* **by** (*simp add*: *bv-smult-def*)
next
assume *bv-to-int w2* = *0*
thus *?thesis* **by** (*simp add*: *bv-smult-def*)
next
assume *p*: *?Q* = −*1*
show *?thesis*
apply (*simp add*: *bv-smult-def p*)
apply (*cut-tac lmw*)
apply *arith*
using *p*
apply *simp*
done
next
assume *p*: *0* < *?Q*
thus *?thesis*
proof (*simp add*: *zero-less-mult-iff*,*safe*)
assume *bi1*: *0* < *bv-to-int w1*
assume *bi2*: *0* < *bv-to-int w2*
show *?thesis*
apply (*simp add*: *bv-smult-def*)
apply (*rule length-int-to-bv-upper-limit-gt0*)
apply (*rule p*)
proof *simp*

```
      have ?Q < 2 ^ (length w1 - 1) * 2 ^ (length w2 - 1)
        apply (rule mult-strict-mono)
        apply (rule bv-to-int-upper-range)
        apply (rule bv-to-int-upper-range)
        apply (rule zero-less-power)
        apply simp
        using bi2
        apply simp
        done
      also have ... ≤ 2 ^ (length w1 + length w2 - Suc 0)
        apply simp
        apply (subst zpower-zadd-distrib [symmetric])
        apply simp
        done
      finally show ?Q < 2 ^ (length w1 + length w2 - Suc 0) .
    qed
  next
    assume bi1: bv-to-int w1 < 0
    assume bi2: bv-to-int w2 < 0
    show ?thesis
      apply (simp add: bv-smult-def)
      apply (rule length-int-to-bv-upper-limit-gt0)
      apply (rule p)
    proof simp
      have -bv-to-int w1 * -bv-to-int w2 ≤ 2 ^ (length w1 - 1) * 2 ^(length w2
- 1)
        apply (rule mult-mono)
        using bv-to-int-lower-range [of w1]
        apply simp
        using bv-to-int-lower-range [of w2]
        apply simp
        apply (rule zero-le-power,simp)
        using bi2
        apply simp
        done
      hence ?Q ≤ 2 ^ (length w1 - 1) * 2 ^(length w2 - 1)
        by simp
      also have ... < 2 ^ (length w1 + length w2 - Suc 0)
        apply simp
        apply (subst zpower-zadd-distrib [symmetric])
        apply simp
        apply (cut-tac lmw)
        apply arith
        apply (cut-tac p)
        apply arith
        done
      finally show ?Q < 2 ^ (length w1 + length w2 - Suc 0) .
    qed
  qed
```

next
assume *p*: *?Q* < -1
show *?thesis*
apply (*subst bv-smult-def*)
apply (*rule length-int-to-bv-upper-limit-lem1*)
apply (*rule p*)
proof *simp*
have $(2::int)$ ^ $(length\ w1 - 1) * 2$ ^$(length\ w2 - 1) \leq 2$ ^ $(length\ w1 + length\ w2 - Suc\ 0)$
apply *simp*
apply (*subst zpower-zadd-distrib* [*symmetric*])
apply *simp*
done
hence $-((2::int)$ ^ $(length\ w1 + length\ w2 - Suc\ 0)) \leq -(2$^$(length\ w1 - 1) * 2$ ^ $(length\ w2 - 1))$
by *simp*
also have ... $\leq$ *?Q*
proof $-$
from *p*
have *q*: $bv\text{-}to\text{-}int\ w1 * bv\text{-}to\text{-}int\ w2 < 0$
by *simp*
thus *?thesis*
proof (*simp add*: *mult-less-0-iff*,*safe*)
assume *bi1*: $0 < bv\text{-}to\text{-}int\ w1$
assume *bi2*: $bv\text{-}to\text{-}int\ w2 < 0$
have $-bv\text{-}to\text{-}int\ w2 * bv\text{-}to\text{-}int\ w1 \leq ((2::int)$^$(length\ w2 - 1)) * (2$ ^ $(length\ w1 - 1))$
apply (*rule mult-mono*)
using *bv-to-int-lower-range* [*of w2*]
apply *simp*
using *bv-to-int-upper-range* [*of w1*]
apply *simp*
apply (*rule zero-le-power*,*simp*)
using *bi1*
apply *simp*
done
hence $-?Q \leq ((2::int)$^$(length\ w1 - 1)) * (2$ ^ $(length\ w2 - 1))$
by (*simp add*: *zmult-ac*)
thus $-(((2::int)$^$(length\ w1 - Suc\ 0)) * (2$ ^ $(length\ w2 - Suc\ 0))) \leq$ *?Q*
by *simp*
next
assume *bi1*: $bv\text{-}to\text{-}int\ w1 < 0$
assume *bi2*: $0 < bv\text{-}to\text{-}int\ w2$
have $-bv\text{-}to\text{-}int\ w1 * bv\text{-}to\text{-}int\ w2 \leq ((2::int)$^$(length\ w1 - 1)) * (2$ ^ $(length\ w2 - 1))$
apply (*rule mult-mono*)
using *bv-to-int-lower-range* [*of w1*]
apply *simp*

```
          using bv-to-int-upper-range [of w2]
          apply simp
          apply (rule zero-le-power,simp)
          using bi2
          apply simp
          done
        hence −?Q ≤ ((2::int)ˆ(length w1 − 1)) ∗ (2 ˆ (length w2 − 1))
          by (simp add: zmult-ac)
        thus −(((2::int)ˆ(length w1 − Suc 0)) ∗ (2 ˆ (length w2 − Suc 0))) ≤
?Q
          by simp
      qed
    qed
    finally show −(2 ˆ (length w1 + length w2 − Suc 0)) ≤ ?Q .
  qed
 qed
qed

lemma bv-msb-one: bv-msb w = 1 ==> bv-to-nat w ≠ 0
by (cases w) simp-all

lemma bv-smult-length-utos: length (bv-smult (utos w1) w2) ≤ length w1 + length
w2
proof −
 let ?Q = bv-to-int (utos w1) ∗ bv-to-int w2

 have lmw: ?Q ≠ 0 ==> 0 < length (utos w1) ∧ 0 < length w2 by auto

 have 0 < ?Q ∨ ?Q = 0 ∨ ?Q = −1 ∨ ?Q < −1 by arith
 thus ?thesis
 proof (safe dest!: iffD1 [OF mult-eq-0-iff])
  assume bv-to-int (utos w1) = 0
  thus ?thesis by (simp add: bv-smult-def)
 next
  assume bv-to-int w2 = 0
  thus ?thesis by (simp add: bv-smult-def)
 next
  assume p: 0 < ?Q
  thus ?thesis
  proof (simp add: zero-less-mult-iff,safe)
   assume biw2: 0 < bv-to-int w2
   show ?thesis
    apply (simp add: bv-smult-def)
    apply (rule length-int-to-bv-upper-limit-gt0)
    apply (rule p)
   proof simp
    have ?Q < 2 ˆ length w1 ∗ 2 ˆ (length w2 − 1)
     apply (rule mult-strict-mono)
     apply (simp add: bv-to-int-utos int-nat-two-exp)
```

```
        apply (rule bv-to-nat-upper-range)
        apply (rule bv-to-int-upper-range)
        apply (rule zero-less-power,simp)
        using biw2
        apply simp
        done
      also have ... ≤ 2 ^ (length w1 + length w2 − Suc 0)
        apply simp
        apply (subst zpower-zadd-distrib [symmetric])
        apply simp
        apply (cut-tac lmw)
        apply arith
        using p
        apply auto
        done
      finally show ?Q < 2 ^ (length w1 + length w2 − Suc 0) .
    qed
  next
    assume bv-to-int (utos w1) < 0
    thus ?thesis by (simp add: bv-to-int-utos)
  qed
next
  assume p: ?Q = −1
  thus ?thesis
    apply (simp add: bv-smult-def)
    apply (cut-tac lmw)
    apply arith
    apply simp
    done
next
  assume p: ?Q < −1
  show ?thesis
    apply (subst bv-smult-def)
    apply (rule length-int-to-bv-upper-limit-lem1)
    apply (rule p)
  proof simp
    have (2::int) ^ length w1 ∗ 2 ^(length w2 − 1) ≤ 2 ^ (length w1 + length
w2 − Suc 0)
      apply simp
      apply (subst zpower-zadd-distrib [symmetric])
      apply simp
      apply (cut-tac lmw)
      apply arith
      apply (cut-tac p)
      apply arith
      done
    hence −((2::int) ^ (length w1 + length w2 − Suc 0)) ≤ −(2^ length w1 ∗ 2
^ (length w2 − 1))
      by simp
```

```
    also have ... ≤ ?Q
    proof −
      from p
      have q: bv-to-int (utos w1) ∗ bv-to-int w2 < 0
        by simp
      thus ?thesis
      proof (simp add: mult-less-0-iff,safe)
        assume bi1: 0 < bv-to-int (utos w1)
        assume bi2: bv-to-int w2 < 0
        have −bv-to-int w2 ∗ bv-to-int (utos w1) ≤ ((2::int)ˆ(length w2 − 1)) ∗
(2 ˆ length w1)
          apply (rule mult-mono)
          using bv-to-int-lower-range [of w2]
          apply simp
          apply (simp add: bv-to-int-utos)
          using bv-to-nat-upper-range [of w1]
          apply (simp add: int-nat-two-exp)
          apply (rule zero-le-power,simp)
          using bi1
          apply simp
          done
        hence −?Q ≤ ((2::int)ˆlength w1) ∗ (2 ˆ (length w2 − 1))
          by (simp add: zmult-ac)
        thus −(((2::int)ˆlength w1) ∗ (2 ˆ (length w2 − Suc 0))) ≤ ?Q
          by simp
      next
        assume bi1: bv-to-int (utos w1) < 0
        thus −(((2::int)ˆlength w1) ∗ (2 ˆ (length w2 − Suc 0))) ≤ ?Q
          by (simp add: bv-to-int-utos)
      qed
    qed
    finally show −(2 ˆ (length w1 + length w2 − Suc 0)) ≤ ?Q .
  qed
 qed
qed

lemma bv-smult-sym: bv-smult w1 w2 = bv-smult w2 w1
  by (simp add: bv-smult-def zmult-ac)
```

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 −
  have ∀ n. n < length w −−> bv-select w n = rev w ! n
  proof (rule length-induct [of - w],auto simp add: bv-select-def)
    fix xs :: bit list
    fix n
   assume ind: ∀ ys::bit list. length ys < length xs −−> (∀ n. n < length ys −−>
ys ! (length ys − Suc n) = rev ys ! n)
    assume notx: n < length xs
    show xs ! (length xs − Suc n) = rev xs ! n
    proof (cases xs)
      assume xs = []
      with notx show ?thesis by simp
    next
      fix y ys
      assume [simp]: xs = y # ys
      show ?thesis
      proof (auto simp add: nth-append)
        assume noty: n < length ys
        from spec [OF ind,of ys]
        have ∀ n. n < length ys −−> ys ! (length ys − Suc n) = rev ys ! n
          by simp
        hence n < length ys −−> ys ! (length ys − Suc n) = rev ys ! n ..
        from this and noty
        have ys ! (length ys − Suc n) = rev ys ! n ..
        thus (y # ys) ! (length ys − n) = rev ys ! n
          by (simp add: nth-Cons' noty linorder-not-less [symmetric])
      next
        assume ~ n < length ys
        hence x: length ys ≤ n by simp
        from notx have n < Suc (length ys) by simp
        hence n ≤ length ys by simp
        with x have length ys = n by simp
        thus y = [y] ! (n − length ys) by simp
      qed
    qed
  qed
  then have n < length w −−> bv-select w n = rev w ! n ..
  from this and notnull show ?thesis ..
qed

lemma bv-chop-append: bv-chop (w1 @ w2) (length w2) = (w1,w2)
  by (simp add: bv-chop-def Let-def)
```

```
lemma append-bv-chop-id: fst (bv-chop w l) @ snd (bv-chop w l) = w
  by (simp add: bv-chop-def Let-def)

lemma bv-chop-length-fst [simp]: length (fst (bv-chop w i)) = length w − i
  by (simp add: bv-chop-def Let-def)

lemma bv-chop-length-snd [simp]: length (snd (bv-chop w i)) = min i (length w)
  by (simp add: bv-chop-def Let-def)

lemma bv-slice-length [simp]: [| j ≤ i; i < length w |] ==> length (bv-slice w
(i,j)) = i − j + 1
  by (auto simp add: bv-slice-def)

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
  apply (simp add: length-nat-def)
  apply (rule Least-equality [symmetric])
  prefer 2
  apply (rule length-nat-to-bv-upper-limit)
  apply arith
  apply (rule ccontr)
proof −
  assume ~ n < 2 ^ length (nat-to-bv n)
  hence 2 ^ length (nat-to-bv n) ≤ n by simp
  hence length (nat-to-bv n) < length (nat-to-bv n)
    by (rule length-nat-to-bv-lower-limit)
  thus False by simp
qed

lemma length-nat-0 [simp]: length-nat 0 = 0
  by (simp add: length-nat-def Least-equality)

lemma length-nat-non0:
  assumes n0: n ≠ 0
  shows      length-nat n = Suc (length-nat (n div 2))
  apply (simp add: length-nat [symmetric])
  apply (subst nat-to-bv-non0 [of n])
  apply (simp-all add: n0)
  done

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 (cases 0 < i)
  assume i0: 0 < i
  hence length (int-to-bv i) =
      length (norm-signed (0 # norm-unsigned (nat-to-bv (nat i)))) by simp
  also from norm-unsigned-result [of nat-to-bv (nat i)]
  have ... = Suc (length-nat (nat i))
    apply safe
    apply (simp del: norm-unsigned-nat-to-bv)
    apply (drule norm-empty-bv-to-nat-zero)
    using prems
    apply simp
    apply (cases norm-unsigned (nat-to-bv (nat i)))
    apply (drule norm-empty-bv-to-nat-zero [of nat-to-bv (nat i)])
    using prems
    apply simp
    apply simp
    using prems
    apply (simp add: length-nat [symmetric])
    done
  finally show ?thesis
    using i0
    by (simp add: length-int-def)
next
  assume ~ 0 < i
  hence i0: i ≤ 0 by simp
  show ?thesis
  proof (cases i = 0)
    assume i = 0
    thus ?thesis by (simp add: length-int-def)
  next
    assume i ≠ 0
    with i0 have i0: i < 0 by simp
    hence length (int-to-bv i) =
        length (norm-signed (1 # bv-not (norm-unsigned (nat-to-bv (nat (- i) -
1)))))
      by (simp add: int-to-bv-def nat-diff-distrib)
    also from norm-unsigned-result [of nat-to-bv (nat (- i) - 1)]
    have ... = Suc (length-nat (nat (- i) - 1))
      apply safe
      apply (simp del: norm-unsigned-nat-to-bv)
      apply (drule norm-empty-bv-to-nat-zero [of nat-to-bv (nat (-i) - Suc 0)])
      using prems
      apply simp
      apply (cases - i - 1 = 0)
      apply simp
      apply (simp add: length-nat [symmetric])
```

```
    apply (cases norm-unsigned (nat-to-bv (nat (- i) - 1)))
    apply simp
    apply simp
    done
  finally
  show ?thesis
    using i0 by (simp add: length-int-def nat-diff-distrib del: int-to-bv-lt0)
 qed
qed

lemma length-int-0 [simp]: length-int 0 = 0
 by (simp add: length-int-def)

lemma length-int-gt0: 0 < i ==> length-int i = Suc (length-nat (nat i))
 by (simp add: length-int-def)

lemma length-int-lt0: i < 0 ==> length-int i = Suc (length-nat (nat (- i) - 1))
 by (simp add: length-int-def nat-diff-distrib)

lemma bv-chopI: [| w = w1 @ w2 ; i = length w2 |] ==> bv-chop w i = (w1,w2)
 by (simp add: bv-chop-def Let-def)

lemma bv-sliceI: [| j ≤ i ; i < length w ; w = w1 @ w2 @ w3 ; Suc i = length
w2 + j ; j = length w3  |] ==> bv-slice w (i,j) = w2
 apply (simp add: bv-slice-def)
 apply (subst bv-chopI [of w1 @ w2 @ w3 w1 w2 @ w3])
 apply simp
 apply simp
 apply simp
 apply (subst bv-chopI [of w2 @ w3 w2 w3],simp-all)
 done

lemma bv-slice-bv-slice:
 assumes ki: k ≤ i
 and     ij: i ≤ j
 and     jl: j ≤ l
 and     lw: l < length w
 shows       bv-slice w (j,i) = bv-slice (bv-slice w (l,k)) (j-k,i-k)
proof -
 def w1  == fst (bv-chop w (Suc l))
 and w2  == fst (bv-chop (snd (bv-chop w (Suc l))) (Suc j))
 and w3  == fst (bv-chop (snd (bv-chop (snd (bv-chop w (Suc l))) (Suc j))) i)
 and w4  == fst (bv-chop (snd (bv-chop (snd (bv-chop (snd (bv-chop w (Suc l)))
(Suc j))) i)) k)
  and w5  == snd (bv-chop (snd (bv-chop (snd (bv-chop (snd (bv-chop w (Suc
l))) (Suc j))) i)) k)
 note w-defs = this

 have w-def: w = w1 @ w2 @ w3 @ w4 @ w5
```

```
    by (simp add: w-defs append-bv-chop-id)

  from ki ij jl lw
  show ?thesis
    apply (subst bv-sliceI [where ?j = i and ?i = j and ?w = w and ?w1.0 =
w1 @ w2 and ?w2.0 = w3 and ?w3.0 = w4 @ w5])
    apply simp-all
    apply (rule w-def)
    apply (simp add: w-defs min-def)
    apply (simp add: w-defs min-def)
    apply (subst bv-sliceI [where ?j = k and ?i = l and ?w = w and ?w1.0 =
w1 and ?w2.0 = w2 @ w3 @ w4 and ?w3.0 = w5])
    apply simp-all
    apply (rule w-def)
    apply (simp add: w-defs min-def)
    apply (simp add: w-defs min-def)
    apply (subst bv-sliceI [where ?j = i−k and ?i = j−k and ?w = w2 @ w3
@ w4 and ?w1.0 = w2 and ?w2.0 = w3 and ?w3.0 = w4])
    apply simp-all
    apply (simp-all add: w-defs min-def)
    done
qed

lemma bv-to-nat-extend [simp]: bv-to-nat (bv-extend n 0 w) = bv-to-nat w
  apply (simp add: bv-extend-def)
  apply (subst bv-to-nat-dist-append)
  apply simp
  apply (induct n − length w)
   apply simp-all
  done

lemma bv-msb-extend-same [simp]: bv-msb w = b ==> bv-msb (bv-extend n b w)
= b
  apply (simp add: bv-extend-def)
  apply (induct n − length w)
   apply simp-all
  done

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 (cases bv-msb w)
  assume [simp]: bv-msb w = 0
  with a have [simp]: b = 0 by simp
  show ?thesis by (simp add: bv-to-int-def)
next
  assume [simp]: bv-msb w = 1
  with a have [simp]: b = 1 by simp
  show ?thesis
```

```
  apply (simp add: bv-to-int-def)
  apply (simp add: bv-extend-def)
  apply (induct n − length w,simp-all)
  done
qed

lemma length-nat-mono [simp]: x ≤ y ==> length-nat x ≤ length-nat y
proof (rule ccontr)
 assume xy: x ≤ y
 assume ~ length-nat x ≤ length-nat y
 hence lxly: length-nat y < length-nat x
  by simp
 hence length-nat y < (LEAST n. x < 2 ^ n)
  by (simp add: length-nat-def)
 hence ~ x < 2 ^ length-nat y
  by (rule not-less-Least)
 hence xx: 2 ^ length-nat y ≤ x
  by simp
 have yy: y < 2 ^ length-nat y
  apply (simp add: length-nat-def)
  apply (rule LeastI)
  apply (subgoal-tac y < 2 ^ y,assumption)
  apply (cases 0 ≤ y)
  apply (induct y,simp-all)
  done
 with xx have y < x by simp
 with xy show False by simp
qed

lemma length-nat-mono-int [simp]: x ≤ y ==> length-nat x ≤ length-nat y
 by (rule length-nat-mono) arith

lemma length-nat-pos [simp,intro!]: 0 < x ==> 0 < length-nat x
 by (simp add: length-nat-non0)

lemma length-int-mono-gt0: [| 0 ≤ x ; x ≤ y |] ==> length-int x ≤ length-int y
 by (cases x = 0) (simp-all add: length-int-gt0 nat-le-eq-zle)

lemma length-int-mono-lt0: [| x ≤ y ; y ≤ 0 |] ==> length-int y ≤ length-int x
 by (cases y = 0) (simp-all add: length-int-lt0)

lemmas [simp] = length-nat-non0

lemma nat-to-bv (number-of Numeral.Pls) = []
 by simp

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*)
by *simp*

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

lemma *fast-bv-to-nat-def*:
bv-to-nat bs == *number-of* (*fast-bv-to-nat-helper bs Numeral.Pls*)
proof (*simp add*: *bv-to-nat-def*)
have ∀ *bin*. ¬ (*neg* (*number-of bin* :: *int*)) ⟶ (*foldl* (%*bn b*. *2* ∗ *bn* + *bitval b*) (*number-of bin*) *bs*) = *number-of* (*fast-bv-to-nat-helper bs bin*)
apply (*induct bs*,*simp*)
apply (*case-tac a*,*simp-all*)
done
thus *foldl* (λ*bn b*. *2* ∗ *bn* + *bitval b*) *0 bs* ≡ *number-of* (*fast-bv-to-nat-helper bs Numeral.Pls*)
by (*simp del*: *nat-numeral-0-eq-0 add*: *nat-numeral-0-eq-0* [*symmetric*])
qed

declare *fast-bv-to-nat-Cons* [*simp del*]
declare *fast-bv-to-nat-Cons0* [*simp*]
declare *fast-bv-to-nat-Cons1* [*simp*]

setup ⟪
(∗*comments containing lcp are the removal of fast-bv-of-nat*∗)
let
fun is-const-bool (*Const*(*True*,-)) = *true*
| *is-const-bool* (*Const*(*False*,-)) = *true*
| *is-const-bool* - = *false*
fun is-const-bit (*Const*(*Word.bit.Zero*,-)) = *true*
| *is-const-bit* (*Const*(*Word.bit.One*,-)) = *true*
| *is-const-bit* - = *false*
fun num-is-usable (*Const*(*Numeral.Pls*,-)) = *true*
| *num-is-usable* (*Const*(*Numeral.Min*,-)) = *false*
| *num-is-usable* (*Const*(*Numeral.Bit*,-) $ *w* $ *b*) =
num-is-usable w andalso is-const-bool b
| *num-is-usable* - = *false*
fun vec-is-usable (*Const*(*List.list.Nil*,-)) = *true*
| *vec-is-usable* (*Const*(*List.list.Cons*,-) $ *b* $ *bs*) =
vec-is-usable bs andalso is-const-bit b
| *vec-is-usable* - = *false*
(∗*lcp*∗∗ *val fast1-th* = *PureThy.get-thm thy Word.fast-nat-to-bv-def*∗)
val fast2-th = @{*thm Word.fast-bv-to-nat-def*};

(*lcp** fun f sg thms (Const(Word.nat-to-bv,-) $ (Const(@{const-name Numeral.number-of},-) $ t)) =
  if num-is-usable t
  then SOME (Drule.cterm-instantiate [(cterm-of sg (Var((w,0),Type(IntDef.int,[]))),cterm-of sg t)] fast1-th)
   else NONE
  | f - - - = NONE *)
 fun g sg thms (Const(Word.bv-to-nat,-) $ (t as (Const(List.list.Cons,-) $ - $ -))) =
    if vec-is-usable t then
    SOME (Drule.cterm-instantiate [(cterm-of sg (Var((bs,0),Type(List.list,[Type(Word.bit,[])]))),cterm-of sg t)] fast2-th)
    else NONE
  | g - - - = NONE
 (*lcp** val simproc1 = Simplifier.simproc thy nat-to-bv [Word.nat-to-bv (number-of w)] f *)
  val simproc2 = Simplifier.simproc @{theory} bv-to-nat [Word.bv-to-nat (x # xs)] g
in
 (fn thy => (Simplifier.change-simpset-of thy (fn ss => ss addsimprocs [(*lcp*simproc1,*)simproc2]);
  thy))
end⟩⟩

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*)
 by (*simp add*: *bv-mapzip-def Let-def split*: *split-max*)

lemma *bv-mapzip-Nil* [*simp*]: *bv-mapzip f* [] [] = []
 by (*simp add*: *bv-mapzip-def Let-def*)

lemma *bv-mapzip-Cons* [*simp*]: *length w1* = *length w2* ==>
  bv-mapzip f (*x#w1*) (*y#w2*) = *f x y* # *bv-mapzip f w1 w2*
 by (*simp add*: *bv-mapzip-def Let-def*)

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* | *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-UnionI*: *Y* $\in$ *Pow(TFin S)* ==> *Union(Y)* $\in$ *TFin S*
  monos *Pow-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)*
  by *blast*

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

lemma *Abrial-axiom1*: $x \subseteq$ *succ S x*
  apply (*unfold succ-def*)
  apply (*rule split-if* [*THEN iffD2*])
  apply (*auto simp add*: *super-def maxchain-def psubset-def*)
  apply (*rule contrapos-np*, *assumption*)
  apply (*rule someI2*, *blast+*)
  done

lemmas *TFin-UnionI* = *TFin.Pow-UnionI* [*OF PowI*]

lemma *TFin-induct*:
        [| *n* $\in$ *TFin S*;
          !!*x*. [| *x* $\in$ *TFin S*; *P(x)* |] ==> *P(succ S x)*;

```
      !!Y. [| Y ⊆ TFin S; Ball Y P |] ==> P(Union Y) |]
    ==> P(n)
  apply (induct set: TFin)
   apply blast+
  done

lemma succ-trans: x ⊆ y ==> x ⊆ succ S y
  apply (erule subset-trans)
  apply (rule Abrial-axiom1)
  done
```

Lemma 1 of section 3.1

```
lemma TFin-linear-lemma1:
    [| n ∈ TFin S;  m ∈ TFin S;
       ∀x ∈ TFin S. x ⊆ m --> x = m | succ S x ⊆ m
     |] ==> n ⊆ m | succ S m ⊆ n
  apply (erule TFin-induct)
   apply (erule-tac [2] Union-lemma0)
  apply (blast del: subsetI intro: succ-trans)
  done
```

Lemma 2 of section 3.2

```
lemma TFin-linear-lemma2:
    m ∈ TFin S ==> ∀n ∈ TFin S. n ⊆ m --> n=m | succ S n ⊆ m
  apply (erule TFin-induct)
   apply (rule impI [THEN ballI])
```

case split using *TFin-linear-lemma1*

```
   apply (rule-tac n1 = n and m1 = x in TFin-linear-lemma1 [THEN disjE],
     assumption+)
    apply (drule-tac x = n in bspec, assumption)
    apply (blast del: subsetI intro: succ-trans, blast)
```

second induction step

```
  apply (rule impI [THEN ballI])
  apply (rule Union-lemma0 [THEN disjE])
    apply (rule-tac [3] disjI2)
    prefer 2 apply blast
   apply (rule ballI)
   apply (rule-tac n1 = n and m1 = x in TFin-linear-lemma1 [THEN disjE],
     assumption+, auto)
  apply (blast intro!: Abrial-axiom1 [THEN subsetD])
  done
```

Re-ordering the premises of Lemma 2

```
lemma TFin-subsetD:
    [| n ⊆ m;  m ∈ TFin S;  n ∈ TFin S |] ==> n=m | succ S n ⊆ m
  by (rule TFin-linear-lemma2 [rule-format])
```

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

lemma *TFin-subset-linear*: $[\mid m \in TFin\ S;\ \ n \in TFin\ S \mid] ==> n \subseteq m \mid m \subseteq n$
  apply (*rule disjE*)
   apply (*rule TFin-linear-lemma1* [*OF - -TFin-linear-lemma2*])
    apply (*assumption+*, *erule disjI2*)
  apply (*blast del*: *subsetI*
   intro: *subsetI Abrial-axiom1* [*THEN subset-trans*])
  done

Lemma 3 of section 3.3

lemma *eq-succ-upper*: $[\mid n \in TFin\ S;\ \ m \in TFin\ S;\ \ m = succ\ S\ m \mid] ==> n \subseteq m$
  apply (*erule TFin-induct*)
   apply (*drule TFin-subsetD*)
    apply (*assumption+*, *force*, *blast*)
  done

Property 3.3 of section 3.3

lemma *equal-succ-Union*: $m \in TFin\ S ==> (m = succ\ S\ m) = (m = Union(TFin\ S))$
  apply (*rule iffI*)
   apply (*rule Union-upper* [*THEN equalityI*])
    apply *assumption*
   apply (*rule eq-succ-upper* [*THEN Union-least*], *assumption+*)
  apply (*erule ssubst*)
  apply (*rule Abrial-axiom1* [*THEN equalityI*])
  apply (*blast del*: *subsetI intro*: *subsetI TFin-UnionI TFin.succI*)
  done

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*: $(\{\} :: 'a\ set\ set) \in chain\ S$
  by (*unfold chain-def*) *auto*

lemma *super-subset-chain*: $super\ S\ c \subseteq chain\ S$
  by (*unfold super-def*) *blast*

lemma *maxchain-subset-chain*: $maxchain\ S \subseteq chain\ S$
  by (*unfold maxchain-def*) *blast*

lemma *mem-super-Ex*: $c \in chain\ S - maxchain\ S ==>\ ?\ d.\ d \in super\ S\ c$
  by (*unfold super-def maxchain-def*) *auto*

lemma *select-super*:
   $c \in chain\ S - maxchain\ S ==> (\epsilon\, c'.\ c'\!: super\ S\ c): super\ S\ c$
  apply (*erule mem-super-Ex* [*THEN exE*])
  apply (*rule someI2*, *auto*)
  done

lemma *select-not-equals*:
$c \in chain\ S - maxchain\ S ==> (\epsilon\, c'.\ c'\!: super\ S\ c) \neq c$
apply (*rule notI*)
apply (*drule select-super*)
apply (*simp add*: *super-def psubset-def*)
done

lemma *succI3*: $c \in chain\ S - maxchain\ S ==> succ\ S\ c = (\epsilon\, c'.\ c'\!: super\ S\ c)$
by (*unfold succ-def*) (*blast intro*!: *if-not-P*)

lemma *succ-not-equals*: $c \in chain\ S - maxchain\ S ==> succ\ S\ c \neq c$
apply (*frule succI3*)
apply (*simp* (*no-asm-simp*))
apply (*rule select-not-equals*, *assumption*)
done

lemma *TFin-chain-lemma4*: $c \in TFin\ S ==> (c :: 'a\ set\ set)\!: chain\ S$
apply (*erule TFin-induct*)
apply (*simp add*: *succ-def select-super* [*THEN super-subset-chain*[*THEN subsetD*]])
apply (*unfold chain-def*)
apply (*rule CollectI*, *safe*)
apply (*drule bspec*, *assumption*)
apply (*rule-tac* [*2*] *m1* = *Xa* **and** *n1* = *X* **in** *TFin-subset-linear* [*THEN disjE*], *blast*+)
done

theorem *Hausdorff*: $\exists\, c.\ (c :: 'a\ set\ set)\!: maxchain\ S$
apply (*rule-tac x* = *Union* (*TFin S*) **in** *exI*)
apply (*rule classical*)
apply (*subgoal-tac succ S* (*Union* (*TFin S*)) = *Union* (*TFin S*))
prefer *2*
apply (*blast intro*!: *TFin-UnionI equal-succ-Union* [*THEN iffD2*, *symmetric*])
apply (*cut-tac subset-refl* [*THEN TFin-UnionI*, *THEN TFin-chain-lemma4*])
apply (*drule DiffI* [*THEN succ-not-equals*], *blast*+)
done

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

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

lemma *chain-Union-upper*: $[|\ c \in chain\ S;\ x \in c\ |] ==> x \subseteq Union(c)$
by (*unfold chain-def*) *auto*

```
lemma chain-ball-Union-upper: c ∈ chain S ==> ∀ x ∈ c. x ⊆ Union(c)
  by (unfold chain-def) auto

lemma maxchain-Zorn:
     [| c ∈ maxchain S; u ∈ S; Union(c) ⊆ u |] ==> Union(c) = u
  apply (rule ccontr)
  apply (simp add: maxchain-def)
  apply (erule conjE)
  apply (subgoal-tac ({u} Un c) ∈ super S c)
   apply simp
  apply (unfold super-def psubset-def)
  apply (blast intro: chain-extend dest: chain-Union-upper)
  done

theorem Zorn-Lemma:
    ∀ c ∈ chain S. Union(c): S ==> ∃ y ∈ S. ∀ z ∈ S. y ⊆ z --> y = z
  apply (cut-tac Hausdorff maxchain-subset-chain)
  apply (erule exE)
  apply (drule subsetD, assumption)
  apply (drule bspec, assumption)
  apply (rule-tac x = Union(c) in bexI)
   apply (rule ballI, rule impI)
   apply (blast dest!: maxchain-Zorn, assumption)
  done
```

38.4 Alternative version of Zorn's Lemma

```
lemma Zorn-Lemma2:
  ∀ c ∈ chain S. ∃ y ∈ S. ∀ x ∈ c. x ⊆ y
    ==> ∃ y ∈ S. ∀ x ∈ S. (y :: 'a set) ⊆ x --> y = x
  apply (cut-tac Hausdorff maxchain-subset-chain)
  apply (erule exE)
  apply (drule subsetD, assumption)
  apply (drule bspec, assumption, erule bexE)
  apply (rule-tac x = y in bexI)
   prefer 2 apply assumption
  apply clarify
  apply (rule ccontr)
  apply (frule-tac z = x in chain-extend)
    apply (assumption, blast)
  apply (unfold maxchain-def super-def psubset-def)
  apply (blast elim!: equalityCE)
  done
```

Various other lemmas

```
lemma chainD: [| c ∈ chain S; x ∈ c; y ∈ c |] ==> x ⊆ y | y ⊆ x
  by (unfold chain-def) blast

lemma chainD2: !!(c :: 'a set set). c ∈ chain S ==> c ⊆ S
  by (unfold chain-def) blast
```

end

39 List-Prefix: List prefixes and postfixes

theory *List-Prefix*
imports *Main*
begin

39.1 Prefix order on lists

instance *list* :: (*type*) *ord* **..**

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*
  by *intro-classes* (*auto simp add*: *prefix-def strict-prefix-def*)

lemma *prefixI* [*intro?*]: $ys = xs\ @\ zs ==> xs \le ys$
  unfolding *prefix-def* **by** *blast*

lemma *prefixE* [*elim?*]:
  assumes $xs \le ys$
  obtains *zs* **where** $ys = xs\ @\ zs$
  using *assms* **unfolding** *prefix-def* **by** *blast*

lemma *strict-prefixI'* [*intro?*]: $ys = xs\ @\ z\ \#\ zs ==> xs < ys$
  unfolding *strict-prefix-def prefix-def* **by** *blast*

lemma *strict-prefixE'* [*elim?*]:
  assumes $xs < ys$
  obtains *z zs* **where** $ys = xs\ @\ z\ \#\ zs$
proof −
  from ‹$xs < ys$› **obtain** *us* **where** $ys = xs\ @\ us$ **and** $xs \ne ys$
    unfolding *strict-prefix-def prefix-def* **by** *blast*
  with *that* **show** *?thesis* **by** (*auto simp add*: *neq-Nil-conv*)
qed

lemma *strict-prefixI* [*intro?*]: $xs \le ys ==> xs \ne ys ==> xs < (ys{::}'a\ list)$
  unfolding *strict-prefix-def* **by** *blast*

lemma *strict-prefixE* [*elim?*]:
  fixes *xs ys* :: *'a list*
  assumes $xs < ys$
  obtains $xs \le ys$ **and** $xs \ne ys$
  using *assms* **unfolding** *strict-prefix-def* **by** *blast*

39.2 Basic properties of prefixes

```
theorem Nil-prefix [iff]: [] ≤ xs
  by (simp add: prefix-def)

theorem prefix-Nil [simp]: (xs ≤ []) = (xs = [])
  by (induct xs) (simp-all add: prefix-def)

lemma prefix-snoc [simp]: (xs ≤ ys @ [y]) = (xs = ys @ [y] ∨ xs ≤ ys)
proof
  assume xs ≤ ys @ [y]
  then obtain zs where zs: ys @ [y] = xs @ zs ..
  show xs = ys @ [y] ∨ xs ≤ ys
  proof (cases zs rule: rev-cases)
    assume zs = []
    with zs have xs = ys @ [y] by simp
    then show ?thesis ..
  next
    fix z zs' assume zs = zs' @ [z]
    with zs have ys = xs @ zs' by simp
    then have xs ≤ ys ..
    then show ?thesis ..
  qed
next
  assume xs = ys @ [y] ∨ xs ≤ ys
  then show xs ≤ ys @ [y]
  proof
    assume xs = ys @ [y]
    then show ?thesis by simp
  next
    assume xs ≤ ys
    then obtain zs where ys = xs @ zs ..
    then have ys @ [y] = xs @ (zs @ [y]) by simp
    then show ?thesis ..
  qed
qed

lemma Cons-prefix-Cons [simp]: (x # xs ≤ y # ys) = (x = y ∧ xs ≤ ys)
  by (auto simp add: prefix-def)

lemma same-prefix-prefix [simp]: (xs @ ys ≤ xs @ zs) = (ys ≤ zs)
  by (induct xs) simp-all

lemma same-prefix-nil [iff]: (xs @ ys ≤ xs) = (ys = [])
proof −
  have (xs @ ys ≤ xs @ []) = (ys ≤ []) by (rule same-prefix-prefix)
  then show ?thesis by simp
qed

lemma prefix-prefix [simp]: xs ≤ ys ==> xs ≤ ys @ zs
```

```
proof -
  assume xs ≤ ys
  then obtain us where ys = xs @ us ..
  then have ys @ zs = xs @ (us @ zs) by simp
  then show ?thesis ..
qed

lemma append-prefixD: xs @ ys ≤ zs ⟹ xs ≤ zs
  by (auto simp add: prefix-def)

theorem prefix-Cons: (xs ≤ y # ys) = (xs = [] ∨ (∃ zs. xs = y # zs ∧ zs ≤ ys))
  by (cases xs) (auto simp add: prefix-def)

theorem prefix-append:
    (xs ≤ ys @ zs) = (xs ≤ ys ∨ (∃ us. xs = ys @ us ∧ us ≤ zs))
  apply (induct zs rule: rev-induct)
   apply force
  apply (simp del: append-assoc add: append-assoc [symmetric])
  apply simp
  apply blast
  done

lemma append-one-prefix:
    xs ≤ ys ==> length xs < length ys ==> xs @ [ys ! length xs] ≤ ys
  apply (unfold prefix-def)
  apply (auto simp add: nth-append)
  apply (case-tac zs)
   apply auto
  done

theorem prefix-length-le: xs ≤ ys ==> length xs ≤ length ys
  by (auto simp add: prefix-def)

lemma prefix-same-cases:
    (xs1::'a list) ≤ ys ⟹ xs2 ≤ ys ⟹ xs1 ≤ xs2 ∨ xs2 ≤ xs1
  apply (simp add: prefix-def)
  apply (erule exE)+
  apply (simp add: append-eq-append-conv-if split: if-splits)
   apply (rule disjI2)
   apply (rule-tac x = drop (size xs2) xs1 in exI)
   apply clarify
   apply (drule sym)
   apply (insert append-take-drop-id [of length xs2 xs1])
   apply simp
  apply (rule disjI1)
  apply (rule-tac x = drop (size xs1) xs2 in exI)
  apply clarify
  apply (insert append-take-drop-id [of length xs1 xs2])
  apply simp
```

done

lemma *set-mono-prefix*:
$xs \le ys \Longrightarrow set\ xs \subseteq set\ ys$
by (*auto simp add*: *prefix-def*)

lemma *take-is-prefix*:
$take\ n\ xs \le xs$
apply (*simp add*: *prefix-def*)
apply (*rule-tac x=drop n xs* **in** *exI*)
apply *simp*
done

lemma *map-prefixI*:
$xs \le ys \Longrightarrow map\ f\ xs \le map\ f\ ys$
by (*clarsimp simp*: *prefix-def*)

lemma *prefix-length-less*:
$xs < ys \Longrightarrow length\ xs < length\ ys$
apply (*clarsimp simp*: *strict-prefix-def*)
apply (*frule prefix-length-le*)
apply (*rule ccontr*, *simp*)
apply (*clarsimp simp*: *prefix-def*)
done

lemma *strict-prefix-simps* [*simp*]:
$xs < [] = False$
$[] < (x \# xs) = True$
$(x \# xs) < (y \# ys) = (x = y \wedge xs < ys)$
by (*simp-all add*: *strict-prefix-def cong*: *conj-cong*)

lemma *take-strict-prefix*:
$xs < ys \Longrightarrow take\ n\ xs < ys$
apply (*induct n arbitrary*: *xs ys*)
apply (*case-tac ys*, *simp-all*)[*1*]
apply (*case-tac xs*, *simp*)
apply (*case-tac ys*, *simp-all*)
done

lemma *not-prefix-cases*:
assumes *pfx*: $\neg\ ps \le ls$
obtains
($c1$) $ps \ne []$ **and** $ls = []$
| ($c2$) *a as x xs* **where** $ps = a\#as$ **and** $ls = x\#xs$ **and** $x = a$ **and** $\neg\ as \le xs$
| ($c3$) *a as x xs* **where** $ps = a\#as$ **and** $ls = x\#xs$ **and** $x \ne a$
proof (*cases ps*)
case *Nil*
then show *?thesis* **using** *pfx* **by** *simp*
next

```
  case (Cons a as)
  then have c: ps = a#as .

  show ?thesis
  proof (cases ls)
    case Nil
    have ps ≠ [] by (simp add: Nil Cons)
    from this and Nil show ?thesis by (rule c1)
  next
    case (Cons x xs)
    show ?thesis
    proof (cases x = a)
      case True
      have ¬ as ≤ xs using pfx c Cons True by simp
      with c Cons True show ?thesis by (rule c2)
    next
      case False
      with c Cons show ?thesis by (rule c3)
    qed
  qed
qed

lemma not-prefix-induct [consumes 1, case-names Nil Neq Eq]:
  assumes np: ¬ ps ≤ ls
    and base: ⋀x xs. P (x#xs) []
    and r1: ⋀x xs y ys. x ≠ y ⟹ P (x#xs) (y#ys)
    and r2: ⋀x xs y ys. ⟦ x = y; ¬ xs ≤ ys; P xs ys ⟧ ⟹ P (x#xs) (y#ys)
  shows P ps ls using np
proof (induct ls arbitrary: ps)
  case Nil then show ?case
    by (auto simp: neq-Nil-conv elim!: not-prefix-cases intro!: base)
next
  case (Cons y ys)
  then have npfx: ¬ ps ≤ (y # ys) by simp
  then obtain x xs where pv: ps = x # xs
    by (rule not-prefix-cases) auto

  from Cons
  have ih: ⋀ps. ¬ps ≤ ys ⟹ P ps ys by simp

  show ?case using npfx
    by (simp only: pv) (erule not-prefix-cases, auto intro: r1 r2 ih)
qed
```

39.3 Parallel lists

```
definition
  parallel :: 'a list => 'a list => bool  (infixl ∥ 50) where
  (xs ∥ ys) = (¬ xs ≤ ys ∧ ¬ ys ≤ xs)
```

lemma *parallelI* [*intro*]: $\neg\ xs \leq ys$ ==> $\neg\ ys \leq xs$ ==> $xs \parallel ys$
  unfolding *parallel-def* **by** *blast*

lemma *parallelE* [*elim*]:
  assumes $xs \parallel ys$
  obtains $\neg\ xs \leq ys \wedge \neg\ ys \leq xs$
  using *assms* **unfolding** *parallel-def* **by** *blast*

theorem *prefix-cases*:
  obtains $xs \leq ys \mid ys < xs \mid xs \parallel ys$
  unfolding *parallel-def strict-prefix-def* **by** *blast*

theorem *parallel-decomp*:
  $xs \parallel ys$ ==> $\exists\, as\ b\ bs\ c\ cs.\ b \neq c \wedge xs = as\ @\ b\ \#\ bs \wedge ys = as\ @\ c\ \#\ cs$
proof (*induct xs rule*: *rev-induct*)
  case *Nil*
  then have *False* **by** *auto*
  then show *?case* **..**
next
  case (*snoc x xs*)
  show *?case*
  proof (*rule prefix-cases*)
    assume *le*: $xs \leq ys$
    then obtain ys' **where** *ys*: $ys = xs\ @\ ys'$ **..**
    show *?thesis*
    proof (*cases ys'*)
      assume $ys' = []$ **with** *ys* **have** $xs = ys$ **by** *simp*
      with *snoc* **have** $[x] \parallel []$ **by** *auto*
      then have *False* **by** *blast*
      then show *?thesis* **..**
    next
      fix *c cs* **assume** *ys'*: $ys' = c\ \#\ cs$
      with *snoc ys* **have** $xs\ @\ [x] \parallel xs\ @\ c\ \#\ cs$ **by** (*simp only*:)
      then have $x \neq c$ **by** *auto*
      moreover have $xs\ @\ [x] = xs\ @\ x\ \#\ []$ **by** *simp*
      moreover from *ys ys'* **have** $ys = xs\ @\ c\ \#\ cs$ **by** (*simp only*:)
      ultimately show *?thesis* **by** *blast*
    qed
  next
    assume $ys < xs$ **then have** $ys \leq xs\ @\ [x]$ **by** (*simp add*: *strict-prefix-def*)
    with *snoc* **have** *False* **by** *blast*
    then show *?thesis* **..**
  next
    assume $xs \parallel ys$
    with *snoc* **obtain** *as b bs c cs* **where** *neq*: $(b::'a) \neq c$
      and *xs*: $xs = as\ @\ b\ \#\ bs$ **and** *ys*: $ys = as\ @\ c\ \#\ cs$
      by *blast*
    from *xs* **have** $xs\ @\ [x] = as\ @\ b\ \#\ (bs\ @\ [x])$ **by** *simp*

with *neq ys* **show** *?thesis* **by** *blast*
qed
qed

lemma *parallel-append*:
$a \parallel b \Longrightarrow a\ @\ c \parallel b\ @\ d$
by (*rule parallelI*)
(*erule parallelE*, *erule conjE*,
induct rule: *not-prefix-induct*, *simp*+)+

lemma *parallel-appendI*:
$[\![\ xs \parallel ys;\ x = xs\ @\ xs'\ ;\ y = ys\ @\ ys'\ ]\!] \Longrightarrow x \parallel y$
by *simp* (*rule parallel-append*)

lemma *parallel-commute*: $(a \parallel b) = (b \parallel a)$
unfolding *parallel-def* **by** *auto*

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*
unfolding *postfix-def* **by** *blast*

lemma *postfixE* [*elim?*]:
assumes *xs* >>= *ys*
obtains *zs* **where** *xs* = *zs* @ *ys*
using *assms* **unfolding** *postfix-def* **by** *blast*

lemma *postfix-refl* [*iff*]: *xs* >>= *xs*
by (*auto simp add*: *postfix-def*)
lemma *postfix-trans*: $[\![$*xs* >>= *ys*; *ys* >>= *zs*$]\!] \Longrightarrow$ *xs* >>= *zs*
by (*auto simp add*: *postfix-def*)
lemma *postfix-antisym*: $[\![$*xs* >>= *ys*; *ys* >>= *xs*$]\!] \Longrightarrow$ *xs* = *ys*
by (*auto simp add*: *postfix-def*)

lemma *Nil-postfix* [*iff*]: *xs* >>= []
by (*simp add*: *postfix-def*)
lemma *postfix-Nil* [*simp*]: ([] >>= *xs*) = (*xs* = [])
by (*auto simp add*: *postfix-def*)

lemma *postfix-ConsI*: *xs* >>= *ys* $\Longrightarrow$ *x#xs* >>= *ys*
by (*auto simp add*: *postfix-def*)
lemma *postfix-ConsD*: *xs* >>= *y#ys* $\Longrightarrow$ *xs* >>= *ys*
by (*auto simp add*: *postfix-def*)

lemma *postfix-appendI*: *xs* >>= *ys* $\Longrightarrow$ *zs* @ *xs* >>= *ys*

```
  by (auto simp add: postfix-def)
lemma postfix-appendD: xs >>= zs @ ys ⟹ xs >>= ys
  by (auto simp add: postfix-def)

lemma postfix-is-subset: xs >>= ys ==> set ys ⊆ set xs
proof –
  assume xs >>= ys
  then obtain zs where xs = zs @ ys ..
  then show ?thesis by (induct zs) auto
qed

lemma postfix-ConsD2: x#xs >>= y#ys ==> xs >>= ys
proof –
  assume x#xs >>= y#ys
  then obtain zs where x#xs = zs @ y#ys ..
  then show ?thesis
    by (induct zs) (auto intro!: postfix-appendI postfix-ConsI)
qed

lemma postfix-to-prefix: xs >>= ys ⟷ rev ys ≤ rev xs
proof
  assume xs >>= ys
  then obtain zs where xs = zs @ ys ..
  then have rev xs = rev ys @ rev zs by simp
  then show rev ys <= rev xs ..
next
  assume rev ys <= rev xs
  then obtain zs where rev xs = rev ys @ zs ..
  then have rev (rev xs) = rev zs @ rev (rev ys) by simp
  then have xs = rev zs @ ys by simp
  then show xs >>= ys ..
qed

lemma distinct-postfix:
  assumes distinct xs
    and xs >>= ys
  shows distinct ys
  using assms by (clarsimp elim!: postfixE)

lemma postfix-map:
  assumes xs >>= ys
  shows map f xs >>= map f ys
  using assms by (auto elim!: postfixE intro: postfixI)

lemma postfix-drop: as >>= drop n as
  unfolding postfix-def
  by (rule exI [where x = take n as]) simp

lemma postfix-take:
```

```
    xs >>= ys ⟹ xs = take (length xs − length ys) xs @ ys
  by (clarsimp elim!: postfixE)

lemma parallelD1: x ∥ y ⟹ ¬ x ≤ y
  by blast

lemma parallelD2: x ∥ y ⟹ ¬ y ≤ x
  by blast

lemma parallel-Nil1 [simp]: ¬ x ∥ []
  unfolding parallel-def by simp

lemma parallel-Nil2 [simp]: ¬ [] ∥ x
  unfolding parallel-def by simp

lemma Cons-parallelI1:
  a ≠ b ⟹ a # as ∥ b # bs by auto

lemma Cons-parallelI2:
  ⟦ a = b; as ∥ bs ⟧ ⟹ a # as ∥ b # bs
  apply simp
  apply (rule parallelI)
   apply simp
   apply (erule parallelD1)
  apply simp
  apply (erule parallelD2)
 done

lemma not-equal-is-parallel:
  assumes neq: xs ≠ ys
    and len: length xs = length ys
  shows xs ∥ ys
  using len neq
proof (induct rule: list-induct2)
  case 1
  then show ?case by simp
next
  case (2 a as b bs)
  have ih: as ≠ bs ⟹ as ∥ bs by fact

  show ?case
  proof (cases a = b)
    case True
    then have as ≠ bs using 2 by simp
    then show ?thesis by (rule Cons-parallelI2 [OF True ih])
  next
    case False
    then show ?thesis by (rule Cons-parallelI1)
  qed
```

qed

39.5 Executable code

lemma *less-eq-code* [*code func*]:
  $([]::'a::\{eq, ord\}\ list) \leq xs \longleftrightarrow True$
  $(x::'a::\{eq, ord\})\ \#\ xs \leq [] \longleftrightarrow False$
  $(x::'a::\{eq, ord\})\ \#\ xs \leq y\ \#\ ys \longleftrightarrow x = y \land xs \leq ys$
 by *simp-all*

lemma *less-code* [*code func*]:
  $xs < ([]::'a::\{eq, ord\}\ list) \longleftrightarrow False$
  $[] < (x::'a::\{eq, ord\})\#\ xs \longleftrightarrow True$
  $(x::'a::\{eq, ord\})\ \#\ xs < y\ \#\ ys \longleftrightarrow x = y \land xs < ys$
 unfolding *strict-prefix-def* **by** *auto*

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: $(xs::('a::ord)\ list) \leq ys \equiv (xs < ys \lor xs = ys)$
 list-less-def: $(xs::('a::ord)\ list) < ys \equiv (xs, ys) \in lexord\ \{(u,v).\ u < v\}$ **..**

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

instance *list* :: (*order*) *order*
 apply (*intro-classes*, *unfold list-ord-defs*)
 apply *safe*
 apply (*rule-tac r1* = $\{(a::'a,b).\ a < b\}$ **in** *lexord-irreflexive* [*THEN notE*])
 apply *simp*
 apply *assumption*
 apply (*blast intro*: *lexord-trans transI order-less-trans*)
 apply (*rule-tac r1* = $\{(a::'a,b).\ a < b\}$ **in** *lexord-irreflexive* [*THEN notE*])
 apply *simp*
 apply (*blast intro*: *lexord-trans transI order-less-trans*)
 done

instance *list* :: (*linorder*) *linorder*
 apply (*intro-classes*, *unfold list-le-def list-less-def*, *safe*)
 apply (*cut-tac* $x = x$ **and** $y = y$ **and** $r = \{(a,b).\ a < b\}$ **in** *lexord-linear*)
  apply *force*

apply *simp*
done

instance *list* :: (*linorder*) *distrib-lattice*
inf ≡ *min*
sup ≡ *max*
by *intro-classes*
(*auto simp add*: *inf-list-def sup-list-def min-max.sup-inf-distrib1*)

lemmas [*code func del*] = *inf-list-def sup-list-def*

lemma *not-less-Nil* [*simp*]: $\neg$ (x < [])
by (*unfold list-less-def*) *simp*

lemma *Nil-less-Cons* [*simp*]: [] < a # x
by (*unfold list-less-def*) *simp*

lemma *Cons-less-Cons* [*simp*]: a # x < b # y $\longleftrightarrow$ $a < b \vee a = b \wedge x < y$
by (*unfold list-less-def*) *simp*

lemma *le-Nil* [*simp*]: $x \leq$ [] $\longleftrightarrow$ x = []
by (*unfold list-ord-defs*, *cases* x) *auto*

lemma *Nil-le-Cons* [*simp*]: [] $\leq x$
by (*unfold list-ord-defs*, *cases* x) *auto*

lemma *Cons-le-Cons* [*simp*]: a # $x \leq b$ # y $\longleftrightarrow$ $a < b \vee a = b \wedge x \leq y$
by (*unfold list-ord-defs*) *auto*

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* $\longleftrightarrow$ $x < y \vee x = y \wedge$ *xs* < *ys*
by *simp-all*

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* $\longleftrightarrow$ $x < y \vee x = y \wedge$ *xs* $\leq$ *ys*
by *simp-all*

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.