The Supplemental Isabelle/HOL Library

June 21, 2010

Contents

1 Abstract-Rat: Abstract rational numbers

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

types $Num = int \times int$

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

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

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

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

declare *gcd-dvd1-int*[*presburger*]
declare *gcd-dvd2-int*[*presburger*]
lemma *normNum-isnormNum* [*simp*]: $isnormNum\ (normNum\ x)$
proof −
  have $\exists\ a\ b.\ x = (a,b)$ **by** *auto*
  then obtain $a\ b$ **where** x[*simp*]: $x = (a,b)$ **by** *blast*
 {assume $a=0 \vee b = 0$ **hence** *?thesis* **by** (*simp add*: *normNum-def isnormNum-def*)**}**

  moreover
  {assume *anz*: $a \neq 0$ **and** *bnz*: $b \neq 0$
    let $?g = gcd\ a\ b$
    let $?a' = a\ div\ ?g$
    let $?b' = b\ div\ ?g$
    let $?g' = gcd\ ?a'\ ?b'$
    from *anz bnz* **have** $?g \neq 0$ **by** *simp* **with** *gcd-ge-0-int*[*of a b*]
    have *gpos*: $?g > 0$ **by** *arith*
    have *gdvd*: $?g\ dvd\ a\ ?g\ dvd\ b$ **by** *arith*+
    from *zdvd-mult-div-cancel*[*OF gdvd*(*1*)] *zdvd-mult-div-cancel*[*OF gdvd*(*2*)]
    anz bnz
    have *nz'*:$?a' \neq 0\ ?b' \neq 0$
      by − (*rule notI*, *simp*)+

from *anz bnz* **have** *stupid*: $a \neq 0 \vee b \neq 0$ **by** *arith*
from *div-gcd-coprime-int*[*OF stupid*] **have** *gp1*: $?g' = 1$ **.**
from *bnz* **have** $b < 0 \vee b > 0$ **by** *arith*
moreover
{assume *b*: $b > 0$
from *b* **have** $?b' \geq 0$
by (*presburger add*: *pos-imp-zdiv-nonneg-iff*[*OF gpos*])
with *nz'* **have** *b'*: $?b' > 0$ **by** *arith*
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'*: $?b' \geq 0$
from *gpos* **have** *th*: $?g \geq 0$ **by** *arith*
from *mult-nonneg-nonneg*[*OF th b'*] *zdvd-mult-div-cancel*[*OF gdvd*(*2*)]
have *False* **using** *b* **by** *arith* **}**
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*)**}**
ultimately have *?thesis* **by** *blast*
}
ultimately show *?thesis* **by** *blast*
qed

Arithmetic over Num

definition
$Nadd :: Num \Rightarrow Num \Rightarrow Num$ (**infixl** $+_N$ *60*)
where
$Nadd = (\lambda(a,b)\ (a',b').$ *if* $a = 0 \vee b = 0$ *then* $normNum(a',b')$
else if $a'=0 \vee b' = 0$ *then* $normNum(a,b)$
else $normNum(a*b' + b*a',\ b*b'))$

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

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

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

definition

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

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

lemma *Nneg-normN*[*simp*]: *isnormNum x* ⟹ *isnormNum* (~$_N$ *x*)
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 ∃ *a b*. *x* = (*a*,*b*) **and** ∃ *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*: *isnormNum-def Let-def Nmul-def split-def*)**}**
moreover
{assume *a*′ = *0*
hence *?thesis* **using** *yn ab ab*′
by (*simp add*: *isnormNum-def Let-def Nmul-def split-def*)**}**
moreover
{assume *a*: *a* ≠*0* **and** *a*′: *a*′≠*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* ⟹ *isnormNum* (*Ninv x*)
by (*simp add*: *Ninv-def isnormNum-def split-def*)
(*cases fst x* = *0*, *auto simp add*: *gcd-commute-int*)

lemma *isnormNum-int*[*simp*]:
isnormNum 0$_N$ *isnormNum* (*1*::*int*)$_N$ *i* ≠ *0* ⟹ *isnormNum i*$_N$
by (*simp-all add*: *isnormNum-def*)

Relations over Num

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

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

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

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

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

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

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

lemma *INum-int* [*simp*]: *INum i*$_N$ = ((*of-int i*) ::'*a*::*field*) *INum 0*$_N$ = (*0*::'*a*::*field*)
by (*simp-all add*: *INum-def*)

lemma *isnormNum-unique*[*simp*]:
assumes *na*: *isnormNum x* **and** *nb*: *isnormNum y*
shows ((*INum x* ::'*a*::{*field-char-0*, *field-inverse-zero*}) = *INum y*) = (*x* = *y*)
(**is** *?lhs* = *?rhs*)
proof
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*
by (*simp add*: *INum-def split-def isnormNum-def split*: *split-if-asm*)**}**
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: gcd a b = 1 gcd b a = 1 gcd a' b' = 1 gcd b' a' = 1

      by (simp-all add: isnormNum-def add: gcd-commute-int)
    from eq have raw-dvd: a dvd a'*b b dvd b'*a a' dvd a*b' b' dvd b*a'
      apply -
      apply algebra
      apply algebra
      apply simp
      apply algebra
      done
    from zdvd-antisym-abs[OF coprime-dvd-mult-int[OF gcd1(2) raw-dvd(2)]
      coprime-dvd-mult-int[OF gcd1(4) raw-dvd(4)]]
      have eq1: b = b' using pos by arith
      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::{field-char-0,
field-inverse-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-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 = λx. 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 algebra-simps prems)
qed

lemma of-int-div: (d::int) ~= 0 ==> d dvd n ==>
    (of-int(n div d)::'a::field-char-0) = of-int n / of-int d
  apply (frule of-int-div-aux [of d n, where ?'a = 'a])
  apply simp
  apply (simp add: dvd-eq-mod-eq-0)
done
```

lemma *normNum*[*simp*]: *INum* (*normNum x*) = (*INum x* :: *'a*::{*field-char-0*, *field-inverse-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* = *gcd 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*: (*INum x* ::*'a*::{*field-char-0*, *field-inverse-zero*}) = *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* :: {*field-char-0*, *field-inverse-zero*})
proof−
let *?z* = *0*:: *'a*
have ∃ *a b*. *x* = (*a*,*b*) ∃ *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* ∨ *a'*= *0* ∨ *b* =*0* ∨ *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* ≠ *0 a'*≠ *0* **and** *bb'*: *b* ≠ *0 b'* ≠ *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' ≠ 0
     let ?g = gcd (a * b' + b * a') (b*b')
     have gz: ?g ≠ 0 using z by simp
     have ?thesis using aa' bb' z gz
       of-int-div[where ?'a = 'a, OF gz gcd-dvd1-int[where x=a * b' + b * a'
and y=b*b']]  of-int-div[where ?'a = 'a,
       OF gz gcd-dvd2-int[where x=a * b' + b * a' and y=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 :: {field-char-0,
field-inverse-zero})
proof−
  let ?z = 0::'a
  have ∃ a b. x = (a,b) ∃ 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 ∨ a'= 0 ∨ b = 0 ∨ 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 ≠ 0 a' ≠ 0 b ≠ 0 b' ≠ 0
    let ?g=gcd (a*a') (b*b')
    have gz: ?g ≠ 0 using z by simp
     from z of-int-div[where ?'a = 'a, OF gz gcd-dvd1-int[where x=a*a' and
y=b*b']]
     of-int-div[where ?'a = 'a , OF gz gcd-dvd2-int[where x=a*a' and y=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 (~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 :: {field-char-0,
field-inverse-zero})
by (simp add: Nsub-def split-def)

lemma Ninv[simp]: INum (Ninv x) = (1::'a :: field-inverse-zero) / (INum x)
  by (simp add: Ninv-def INum-def split-def)

lemma Ndiv[simp]: INum (x ÷N y) = INum x / (INum y ::'a :: {field-char-0,
field-inverse-zero}) by (simp add: Ndiv-def)
```

lemma *Nlt0-iff*[*simp*]: **assumes** *nx*: *isnormNum x*
shows ((*INum x* :: *'a* :: {*field-char-0*, *linordered-field-inverse-zero*})< *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*: *Nlt0-def INum-def*) **}**
moreover
{assume *a*: *a*≠*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* :: {*field-char-0*, *linordered-field-inverse-zero*}) ≤ *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*: *Nle0-def INum-def*) **}**
moreover
{assume *a*: *a*≠*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* :: {*field-char-0*, *linordered-field-inverse-zero*})> *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* :: {*field-char-0*, *linordered-field-inverse-zero*}) ≥ *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*: *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 :: {*field-char-0*, *linordered-field-inverse-zero*}) < *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 :: {*field-char-0*, *linordered-field-inverse-zero*})≤ *INum y*)
= (*x* ≤*N* *y*)
proof−
have ((*INum x* ::'*a*) ≤ *INum y*) = (*INum* (*x* −*N* *y*) ≤ (*0*::'*a*)) **using** *nx ny* **by** *simp*
also have ... = (*0*≥*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*:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows *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*) = *INum* (*y* +*N* *x*) **by** *simp*
with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

lemma [*simp*]:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows (*0*, *b*) +*N* *y* = *normNum y*
and (*a*, *0*) +*N* *y* = *normNum y*
and *x* +*N* (*0*, *b*) = *normNum x*
and *x* +*N* (*a*, *0*) = *normNum x*
apply (*simp add*: *Nadd-def split-def*)
apply (*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 *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
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*) **by** *simp*
with *isnormNum-unique*[*OF n nx*]
show *?thesis* **by** *simp*
qed

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

lemma *Nadd-normNum1*[*simp*]:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows *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*) **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*]:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows *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*) **by** *simp*
also have ... = *INum* (*x* +$_N$ *y*) **by** *simp*
finally show *?thesis* **using** *isnormNum-unique*[*OF n*] **by** *simp*
qed

lemma *Nadd-assoc*:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
shows *x* +$_N$ *y* +$_N$ *z* = *x* +$_N$ (*y* +$_N$ *z*)
proof−
have *n*: *isnormNum* (*x* +$_N$ *y* +$_N$ *z*) *isnormNum* (*x* +$_N$ (*y* +$_N$ *z*)) **by** *simp-all*

have *INum* $(x +_N y +_N z) = (INum\ (x +_N (y +_N z)) :: 'a)$ **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 gcd-commute-int mult-commute*)

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

by *simp-all*
have *INum* $(x +_N y +_N z) = (INum\ (x +_N (y +_N z)) :: 'a)$ **by** *simp*
with *isnormNum-unique*[*OF n*] **show** *?thesis* **by** *simp*
qed

lemma *Nsub0*:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
assumes *x*: *isnormNum x* **and** *y*:*isnormNum y* **shows** $(x -_N y = 0_N) = (x = y)$
proof−
{ fix *h* :: '*a*
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*]: $c *_N 0_N = 0_N \quad 0_N *_N c = 0_N$
by (*simp-all add*: *Nmul-def Let-def split-def*)

lemma *Nmul-eq0*[*simp*]:
assumes *SORT-CONSTRAINT*('*a*::{*field-char-0*, *field-inverse-zero*})
assumes *nx*:*isnormNum x* **and** *ny*: *isnormNum y*
shows $(x *_N y = 0_N) = (x = 0_N \vee y = 0_N)$
proof−
{ fix *h* :: '*a*
have $\exists\ a\ b\ a'\ b'.\ x = (a,b) \wedge y = (a',b')$ **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*])
by (*simp add*: *INum-def split-def isnormNum-def fst-conv snd-conv split*:

split-if-asm)
}
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*)
apply (*cases fst c* = *0*, *simp-all*, *cases c*, *simp-all*)+
done

end

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

theory *Mapping*
imports *Main*
begin

2.1 Type definition and primitive operations

datatype ('*a*, '*b*) *mapping* = *Mapping* '*a* $\rightharpoonup$ '*b*

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

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

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

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

2.2 Derived operations

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

definition *ordered-keys* :: ('*a*::*linorder*, '*b*) *mapping* ⇒ '*a list* **where**
ordered-keys m = (*if finite* (*keys m*) *then sorted-list-of-set* (*keys m*) *else* [])

definition *is-empty* :: ('*a*, '*b*) *mapping* ⇒ *bool* **where**
is-empty m $\longleftrightarrow$ *keys m* = {}

definition *size* :: ('*a*, '*b*) *mapping* ⇒ *nat* **where**
size m = (*if finite* (*keys m*) *then card* (*keys m*) *else 0*)

definition *replace* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping* **where**
replace k v m = (*if k* ∈ *keys m then update k v m else m*)

definition *default* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping* **where**
default k v m = (*if k* ∈ *keys m then m else update k v m*)

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

definition *map-default* :: '*a* ⇒ '*b* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping* **where**
map-default k v f m = *map-entry k f* (*default k v m*)

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

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

2.3 Properties

lemma *lookup-inject* [*simp*]:
lookup m = *lookup n* ⟷ *m* = *n*
by (*cases m*, *cases n*) *simp*

lemma *mapping-eqI*:
assumes *lookup m* = *lookup n*
shows *m* = *n*
using *assms* **by** *simp*

lemma *keys-is-none-lookup* [*code-inline*]:
k ∈ *keys m* ⟷ ¬ (*Option.is-none* (*lookup m k*))
by (*auto simp add*: *keys-def is-none-def*)

lemma *lookup-empty* [*simp*]:
lookup empty = *Map.empty*
by (*simp add*: *empty-def*)

lemma *lookup-update* [*simp*]:
lookup (*update k v m*) = (*lookup m*) (*k* ↦ *v*)
by (*cases m*) *simp*

lemma *lookup-delete* [*simp*]:

lookup (*delete k m*) = (*lookup m*) (*k* := *None*)
by (*cases m*) *simp*

lemma *lookup-map-entry* [*simp*]:
lookup (*map-entry k f m*) = (*lookup m*) (*k* := *Option.map f* (*lookup m k*))
by (*cases lookup m k*) (*simp-all add*: *map-entry-def expand-fun-eq*)

lemma *lookup-tabulate* [*simp*]:
lookup (*tabulate ks f*) = (*Some o f*) |‘ *set ks*
by (*induct ks*) (*auto simp add*: *tabulate-def restrict-map-def expand-fun-eq*)

lemma *lookup-bulkload* [*simp*]:
lookup (*bulkload xs*) = (λ*k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None*)
by (*simp add*: *bulkload-def*)

lemma *update-update*:
update k v (*update k w m*) = *update k v m*
k ≠ *l* ⟹ *update k v* (*update l w m*) = *update l w* (*update k v m*)
by (*rule mapping-eqI*, *simp add*: *fun-upd-twist*)+

lemma *update-delete* [*simp*]:
update k v (*delete k m*) = *update k v m*
by (*rule mapping-eqI*) *simp*

lemma *delete-update*:
delete k (*update k v m*) = *delete k m*
k ≠ *l* ⟹ *delete k* (*update l v m*) = *update l v* (*delete k m*)
by (*rule mapping-eqI*, *simp add*: *fun-upd-twist*)+

lemma *delete-empty* [*simp*]:
delete k empty = *empty*
by (*rule mapping-eqI*) *simp*

lemma *replace-update*:
k ∉ *keys m* ⟹ *replace k v m* = *m*
k ∈ *keys m* ⟹ *replace k v m* = *update k v m*
by (*rule mapping-eqI*) (*auto simp add*: *replace-def fun-upd-twist*)+

lemma *size-empty* [*simp*]:
size empty = *0*
by (*simp add*: *size-def keys-def*)

lemma *size-update*:
finite (*keys m*) ⟹ *size* (*update k v m*) =
(*if k* ∈ *keys m then size m else Suc* (*size m*))
by (*auto simp add*: *size-def insert-dom keys-def*)

lemma *size-delete*:
size (*delete k m*) = (*if k* ∈ *keys m then size m* − *1 else size m*)

by (*simp add*: *size-def keys-def*)

lemma *size-tabulate* [*simp*]:
size (*tabulate ks f*) = *length* (*remdups ks*)
by (*simp add*: *size-def distinct-card* [*of remdups ks*, *symmetric*] *comp-def keys-def*)

lemma *bulkload-tabulate*:
bulkload xs = *tabulate* [*0*..<*length xs*] (*nth xs*)
by (*rule mapping-eqI*) (*simp add*: *expand-fun-eq*)

lemma *is-empty-empty*:
is-empty m ⟷ *m* = *Mapping Map.empty*
by (*cases m*) (*simp add*: *is-empty-def keys-def*)

lemma *is-empty-empty'* [*simp*]:
is-empty empty
by (*simp add*: *is-empty-empty empty-def*)

lemma *is-empty-update* [*simp*]:
¬ *is-empty* (*update k v m*)
by (*cases m*) (*simp add*: *is-empty-empty*)

lemma *is-empty-delete*:
is-empty (*delete k m*) ⟷ *is-empty m* ∨ *keys m* = {*k*}
by (*cases m*) (*auto simp add*: *is-empty-empty keys-def dom-eq-empty-conv* [*symmetric*]
simp del: *dom-eq-empty-conv*)

lemma *is-empty-replace* [*simp*]:
is-empty (*replace k v m*) ⟷ *is-empty m*
by (*auto simp add*: *replace-def*) (*simp add*: *is-empty-def*)

lemma *is-empty-default* [*simp*]:
¬ *is-empty* (*default k v m*)
by (*auto simp add*: *default-def*) (*simp add*: *is-empty-def*)

lemma *is-empty-map-entry* [*simp*]:
is-empty (*map-entry k f m*) ⟷ *is-empty m*
by (*cases lookup m k*)
(*auto simp add*: *map-entry-def*, *simp add*: *is-empty-empty*)

lemma *is-empty-map-default* [*simp*]:
¬ *is-empty* (*map-default k v f m*)
by (*simp add*: *map-default-def*)

lemma *keys-empty* [*simp*]:
keys empty = {}
by (*simp add*: *keys-def*)

lemma *keys-update* [*simp*]:

keys (*update k v m*) = *insert k* (*keys m*)
by (*simp add*: *keys-def*)

lemma *keys-delete* [*simp*]:
keys (*delete k m*) = *keys m* − {*k*}
by (*simp add*: *keys-def*)

lemma *keys-replace* [*simp*]:
keys (*replace k v m*) = *keys m*
by (*auto simp add*: *keys-def replace-def*)

lemma *keys-default* [*simp*]:
keys (*default k v m*) = *insert k* (*keys m*)
by (*auto simp add*: *keys-def default-def*)

lemma *keys-map-entry* [*simp*]:
keys (*map-entry k f m*) = *keys m*
by (*auto simp add*: *keys-def*)

lemma *keys-map-default* [*simp*]:
keys (*map-default k v f m*) = *insert k* (*keys m*)
by (*simp add*: *map-default-def*)

lemma *keys-tabulate* [*simp*]:
keys (*tabulate ks f*) = *set ks*
by (*simp add*: *tabulate-def keys-def map-of-map-restrict o-def*)

lemma *keys-bulkload* [*simp*]:
keys (*bulkload xs*) = {*0*..<*length xs*}
by (*simp add*: *keys-tabulate bulkload-tabulate*)

lemma *distinct-ordered-keys* [*simp*]:
distinct (*ordered-keys m*)
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-infinite* [*simp*]:
¬ *finite* (*keys m*) ⟹ *ordered-keys m* = []
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-empty* [*simp*]:
ordered-keys empty = []
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-update* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*update k v m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*update k v m*) = *insort k* (*ordered-keys m*)
by (*simp-all add*: *ordered-keys-def*) (*auto simp only*: *sorted-list-of-set-insert* [*symmetric*] *insert-absorb*)

lemma *ordered-keys-delete* [*simp*]:
ordered-keys (*delete k m*) = *remove1 k* (*ordered-keys m*)
proof (*cases finite* (*keys m*))
case *False* **then show** *?thesis* **by** *simp*
next
case *True* **note** *fin* = *True*
show *?thesis*
proof (*cases k* ∈ *keys m*)
case *False* **with** *fin* **have** *k* ∉ *set* (*sorted-list-of-set* (*keys m*)) **by** *simp*
with *False* **show** *?thesis* **by** (*simp add*: *ordered-keys-def remove1-idem*)
next
case *True* **with** *fin* **show** *?thesis* **by** (*simp add*: *ordered-keys-def sorted-list-of-set-remove*)
qed
qed

lemma *ordered-keys-replace* [*simp*]:
ordered-keys (*replace k v m*) = *ordered-keys m*
by (*simp add*: *replace-def*)

lemma *ordered-keys-default* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*default k v m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*default k v m*) = *insort k* (*ordered-keys m*)
by (*simp-all add*: *default-def*)

lemma *ordered-keys-map-entry* [*simp*]:
ordered-keys (*map-entry k f m*) = *ordered-keys m*
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-map-default* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*map-default k v f m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*map-default k v f m*) = *insort k* (*ordered-keys m*)
by (*simp-all add*: *map-default-def*)

lemma *ordered-keys-tabulate* [*simp*]:
ordered-keys (*tabulate ks f*) = *sort* (*remdups ks*)
by (*simp add*: *ordered-keys-def sorted-list-of-set-sort-remdups*)

lemma *ordered-keys-bulkload* [*simp*]:
ordered-keys (*bulkload ks*) = [*0*..<*length ks*]
by (*simp add*: *ordered-keys-def*)

2.4 Some technical code lemmas

lemma [*code*]:
mapping-case f m = *f* (*Mapping.lookup m*)
by (*cases m*) *simp*

lemma [*code*]:
mapping-rec f m = f (Mapping.lookup m)
by (*cases m*) *simp*

lemma [*code*]:
Nat.size (m :: (-, -) mapping) = 0
by (*cases m*) *simp*

lemma [*code*]:
mapping-size f g m = 0
by (*cases m*) *simp*

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

end

3 AssocList: Map operations implemented on association lists

theory *AssocList*
imports *Main Mapping*
begin

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

3.1 *update* and *updates*

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

lemma *update-conv'*: *map-of* (*update k v al*) = (*map-of al*)(*k*↦*v*)
by (*induct al*) (*auto simp add*: *expand-fun-eq*)

corollary *update-conv*: *map-of* (*update k v al*) *k'* = ((*map-of al*)(*k*↦*v*)) *k'*
by (*simp add*: *update-conv'*)

lemma *dom-update*: *fst* ‘ *set* (*update k v al*) = {*k*} ∪ *fst* ‘ *set al*
by (*induct al*) *auto*

lemma *update-keys*:
map fst (*update k v al*) =
(*if k* ∈ *set* (*map fst al*) *then map fst al else map fst al* @ [*k*])

by (*induct al*) *simp-all*

lemma *distinct-update*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*update k v al*))
using *assms* **by** (*simp add*: *update-keys*)

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 *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 (*simp add*: *update-conv' expand-fun-eq*)

lemma *update-Some-unfold*:
map-of (*update k v al*) *x* = *Some y* $\longleftrightarrow$
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*)

definition *updates* :: *'key list* $\Rightarrow$ *'val list* $\Rightarrow$ (*'key* $\times$ *'val*) *list* $\Rightarrow$ (*'key* $\times$ *'val*) *list*
where
updates ks vs al = *foldl* (λ*al* (*k*, *v*). *update k v al*) *al* (*zip ks vs*)

lemma *updates-simps* [*simp*]:
updates [] *vs ps* = *ps*
updates ks [] *ps* = *ps*
updates (*k*#*ks*) (*v*#*vs*) *ps* = *updates ks vs* (*update k v ps*)
by (*simp-all add*: *updates-def*)

lemma *updates-key-simp* [*simp*]:
updates (*k* # *ks*) *vs ps* =
(*case vs of* [] ⇒ *ps* | *v* # *vs* ⇒ *updates ks vs* (*update k v ps*))
by (*cases vs*) *simp-all*

lemma *updates-conv'*: *map-of* (*updates ks vs al*) = (*map-of al*)(*ks*[↦]*vs*)
proof −
have *foldl* (λ*f* (*k*, *v*). *f*(*k* ↦ *v*)) (*map-of al*) (*zip ks vs*) =
map-of (*foldl* (λ*al* (*k*, *v*). *update k v al*) *al* (*zip ks vs*))
by (*rule foldl-apply*) (*auto simp add*: *expand-fun-eq update-conv'*)
then show *?thesis*
by (*simp add*: *updates-def map-upds-fold-map-upd*)
qed

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

lemma *distinct-updates*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*updates ks vs al*))
proof −
from *assms* **have** *distinct* (*foldl*
(λ*al* (*k*, *v*). *if k* ∈ *set al then al else al* @ [*k*])
(*map fst al*) (*zip ks vs*))
by (*rule foldl-invariant*) *auto*
moreover have *foldl* (λ*al* (*k*, *v*). *if k* ∈ *set al then al else al* @ [*k*])
(*map fst al*) (*zip ks vs*)
= *map fst* (*foldl* (λ*al* (*k*, *v*). *update k v al*) *al* (*zip ks vs*))
by (*rule foldl-apply*) (*simp add*: *update-keys split-def comp-def*)
ultimately show *?thesis* **by** (*simp add*: *updates-def*)
qed

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* ⟹ *updates xs* (*ys*@*zs*) *al* = *updates xs ys al*
by (*induct xs arbitrary*: *ys al*) (*auto split*: *list.splits*)

3.2 *delete*

definition *delete* :: *'key* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* **where**
delete-eq: *delete k* = *filter* (λ(*k'*, -). *k* ≠ *k'*)

lemma *delete-simps* [*simp*]:
delete k [] = []
delete k (*p*#*ps*) = (*if fst p* = *k then delete k ps else p* # *delete k ps*)
by (*auto simp add*: *delete-eq*)

lemma *delete-conv'*: *map-of* (*delete k al*) = (*map-of al*)(*k* := *None*)
by (*induct al*) (*auto simp add*: *expand-fun-eq*)

corollary *delete-conv*: *map-of* (*delete k al*) *k'* = ((*map-of al*)(*k* := *None*)) *k'*
by (*simp add*: *delete-conv'*)

lemma *delete-keys*:
map fst (*delete k al*) = *removeAll k* (*map fst al*)
by (*simp add*: *delete-eq removeAll-filter-not-eq filter-map split-def comp-def*)

lemma *distinct-delete*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*delete k al*))
using *assms* **by** (*simp add*: *delete-keys distinct-removeAll*)

lemma *delete-id* [*simp*]: *k* ∉ *fst* ' *set al* ⟹ *delete k al* = *al*
by (*auto simp add*: *image-iff delete-eq filter-id-conv*)

lemma *delete-idem*: *delete k* (*delete k al*) = *delete k al*
by (*simp add*: *delete-eq*)

lemma *map-of-delete* [*simp*]:
k' ≠ *k* ⟹ *map-of* (*delete k al*) *k'* = *map-of al k'*
by (*simp add*: *delete-conv'*)

lemma *delete-notin-dom*: *k* ∉ *fst* ‘ *set* (*delete k al*)
by (*auto simp add*: *delete-eq*)

lemma *dom-delete-subset*: *fst* ‘ *set* (*delete k al*) ⊆ *fst* ‘ *set al*
by (*auto simp add*: *delete-eq*)

lemma *delete-update-same*:
delete k (*update k v al*) = *delete k al*
by (*induct al*) *simp-all*

lemma *delete-update*:
k ≠ *l* ⟹ *delete l* (*update k v al*) = *update k v* (*delete l al*)
by (*induct al*) *simp-all*

lemma *delete-twist*: *delete x* (*delete y al*) = *delete y* (*delete x al*)
by (*simp add*: *delete-eq conj-commute*)

lemma *length-delete-le*: *length* (*delete k al*) ≤ *length al*
by (*simp add*: *delete-eq*)

3.3 *restrict*

definition *restrict* :: *'key set* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* **where**
restrict-eq: *restrict A* = *filter* (λ(*k*, *v*). *k* ∈ *A*)

lemma *restr-simps* [*simp*]:
restrict A [] = []
restrict A (*p*#*ps*) = (*if fst p* ∈ *A then p* # *restrict A ps else restrict A ps*)
by (*auto simp add*: *restrict-eq*)

lemma *restr-conv'*: *map-of* (*restrict A al*) = ((*map-of al*)|‘ *A*)
proof
fix *k*
show *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k*
by (*induct al*) (*simp*, *cases k* ∈ *A*, *auto*)
qed

corollary *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k*
by (*simp add*: *restr-conv'*)

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

by (*induct al*) (*auto simp add*: *restrict-eq*)

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

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* ∩ *A*
by (*induct al*) (*auto simp add*: *restrict-eq*)

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

lemma *restr-restr* [*simp*]: *restrict A* (*restrict B al*) = *restrict* (*A*∩*B*) *al*
by (*induct al*) (*auto simp add*: *restrict-eq*)

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*)
apply (*simp add*: *delete-eq restrict-eq*)
apply (*auto simp add*: *split-def*)
proof −
have $\bigwedge y.\ y \neq x \longleftrightarrow x \neq y$ **by** *auto*
then show [*p* ← *al*. *fst p* ∈ *D* ∧ *x* ≠ *fst p*] = [*p* ← *al*. *fst p* ∈ *D* ∧ *fst p* ≠ *x*]
by *simp*
assume *x* ∉ *D*
then have $\bigwedge y.\ y \in D \longleftrightarrow y \in D \wedge x \neq y$ **by** *auto*
then show [*p* ← *al* . *fst p* ∈ *D* ∧ *x* ≠ *fst p*] = [*p* ← *al* . *fst p* ∈ *D*]
by *simp*
qed

lemma *update-restr*:
map-of (*update x y* (*restrict D al*)) = *map-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 ∈ *D* ⟹

map-of (*update x y* (*restrict D al*)) = *map-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* ⊆ *D* ⟧
⟹ *map-of* (*restrict D* (*updates xs ys al*)) =
map-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*

3.4 *clearjunk*

function *clearjunk* :: ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list* **where**
clearjunk [] = []
| *clearjunk* (*p*#*ps*) = *p* # *clearjunk* (*delete* (*fst p*) *ps*)
by *pat-completeness auto*
termination by (*relation measure length*)
(*simp-all add*: *less-Suc-eq-le length-delete-le*)

lemma *map-of-clearjunk*:
map-of (*clearjunk al*) = *map-of al*
by (*induct al rule*: *clearjunk.induct*)
(*simp-all add*: *expand-fun-eq*)

lemma *clearjunk-keys-set*:
set (*map fst* (*clearjunk al*)) = *set* (*map fst al*)
by (*induct al rule*: *clearjunk.induct*)
(*simp-all add*: *delete-keys*)

lemma *dom-clearjunk*:
fst ' *set* (*clearjunk al*) = *fst* ' *set al*
using *clearjunk-keys-set* **by** *simp*

lemma *distinct-clearjunk* [*simp*]:
distinct (*map fst* (*clearjunk al*))
by (*induct al rule*: *clearjunk.induct*)
(*simp-all del*: *set-map add*: *clearjunk-keys-set delete-keys*)

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

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

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

lemma *clearjunk-updates*:
clearjunk (*updates ks vs al*) = *updates ks vs* (*clearjunk al*)
proof −
have *foldl* (λ*al* (*k*, *v*). *update k v al*) (*clearjunk al*) (*zip ks vs*) =
clearjunk (*foldl* (λ*al* (*k*, *v*). *update k v al*) *al* (*zip ks vs*))
by (*rule foldl-apply*) (*simp add*: *clearjunk-update expand-fun-eq split-def*)
then show *?thesis* **by** (*simp add*: *updates-def*)
qed

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

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

lemma *distinct-clearjunk-id* [*simp*]:
distinct (*map fst al*) ⟹ *clearjunk al* = *al*
by (*induct al rule*: *clearjunk.induct*) *auto*

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

lemma *length-clearjunk*:
length (*clearjunk al*) ≤ *length al*
proof (*induct al rule*: *clearjunk.induct* [*case-names Nil Cons*])
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons kv al*)
moreover have *length* (*delete* (*fst kv*) *al*) ≤ *length al* **by** (*fact length-delete-le*)
ultimately have *length* (*clearjunk* (*delete* (*fst kv*) *al*)) ≤ *length al* **by** (*rule order-trans*)
then show *?case* **by** *simp*
qed

lemma *delete-map*:
assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*

shows *delete k (map f ps) = map f (delete k ps)*
by (*simp add*: *delete-eq filter-map comp-def split-def assms*)

lemma *clearjunk-map*:
assumes $\bigwedge kv$. *fst (f kv) = fst kv*
shows *clearjunk (map f ps) = map f (clearjunk ps)*
by (*induct ps rule*: *clearjunk.induct* [*case-names Nil Cons*])
(*simp-all add*: *clearjunk-delete delete-map assms*)

3.5 *map-ran*

definition *map-ran* :: (*'key* ⇒ *'val* ⇒ *'val*) ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* **where**
map-ran f = map (λ(*k*, *v*). (*k*, *f k v*))

lemma *map-ran-simps* [*simp*]:
map-ran f [] = []
map-ran f ((*k*, *v*) # *ps*) = (*k*, *f k v*) # *map-ran f ps*
by (*simp-all add*: *map-ran-def*)

lemma *dom-map-ran*:
fst ' set (*map-ran f al*) = *fst ' set al*
by (*simp add*: *map-ran-def image-image split-def*)

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

lemma *distinct-map-ran*:
distinct (*map fst al*) ⟹ *distinct* (*map fst* (*map-ran f al*))
by (*simp add*: *map-ran-def split-def comp-def*)

lemma *map-ran-filter*:
map-ran f [*p*←*ps*. *fst p* ≠ *a*] = [*p*←*map-ran f ps*. *fst p* ≠ *a*]
by (*simp add*: *map-ran-def filter-map split-def comp-def*)

lemma *clearjunk-map-ran*:
clearjunk (*map-ran f al*) = *map-ran f* (*clearjunk al*)
by (*simp add*: *map-ran-def split-def clearjunk-map*)

3.6 *merge*

definition *merge* :: (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* **where**
merge qs ps = foldr (λ(*k*, *v*). *update k v*) *ps qs*

lemma *merge-simps* [*simp*]:
merge qs [] = *qs*
merge qs (*p*#*ps*) = *update* (*fst p*) (*snd p*) (*merge qs ps*)
by (*simp-all add*: *merge-def split-def*)

lemma *merge-updates*:
merge qs ps = *updates* (*rev* (*map fst ps*)) (*rev* (*map snd ps*)) *qs*
by (*simp add*: *merge-def updates-def split-def*
foldr-foldl zip-rev zip-map-fst-snd)

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** (*simp add*: *merge-updates distinct-updates*)

lemma *clearjunk-merge*:
clearjunk (*merge xs ys*) = *merge* (*clearjunk xs*) *ys*
by (*simp add*: *merge-updates clearjunk-updates*)

lemma *merge-conv′*:
map-of (*merge xs ys*) = *map-of xs* ++ *map-of ys*
proof −
have *foldl* (λ*m* (*k*, *v*). *m*(*k* ↦ *v*)) (*map-of xs*) (*rev ys*) =
map-of (*foldl* (λ*xs* (*k*, *v*). *update k v xs*) *xs* (*rev ys*))
by (*rule foldl-apply*) (*simp add*: *expand-fun-eq split-def update-conv′*)
then show *?thesis*
by (*simp add*: *merge-def map-add-map-of-foldr foldr-foldl split-def*)
qed

corollary *merge-conv*:
map-of (*merge xs ys*) *k* = (*map-of xs* ++ *map-of ys*) *k*
by (*simp add*: *merge-conv′*)

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

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

3.7 *compose*

function *compose* :: ('*key* × '*a*) *list* ⇒ ('*a* × '*b*) *list* ⇒ ('*key* × '*b*) *list* **where**
compose [] *ys* = []
| *compose* (*x*#*xs*) *ys* = (*case map-of ys* (*snd x*)
of None ⇒ *compose* (*delete* (*fst x*) *xs*) *ys*
| *Some v* ⇒ (*fst x*, *v*) # *compose xs ys*)
by *pat-completeness auto*
termination by (*relation measure* (*length* ∘ *fst*))
(*simp-all add*: *less-Suc-eq-le length-delete-le*)

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)
```

3.8 Implementation of mappings

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

code-datatype Mapping

lemma lookup-Mapping [simp, code]:
  Mapping.lookup (Mapping xs) = map-of xs
  by (simp add: Mapping-def)

lemma keys-Mapping [simp, code]:
  Mapping.keys (Mapping xs) = set (map fst xs)
```

by (*simp add*: *keys-def dom-map-of-conv-image-fst*)

lemma *empty-Mapping* [*code*]:
Mapping.empty = *Mapping* []
by (*rule mapping-eqI*) *simp*

lemma *is-empty-Mapping* [*code*]:
Mapping.is-empty (*Mapping xs*) ⟷ *null xs*
by (*cases xs*) (*simp-all add*: *is-empty-def*)

lemma *update-Mapping* [*code*]:
Mapping.update k v (*Mapping xs*) = *Mapping* (*update k v xs*)
by (*rule mapping-eqI*) (*simp add*: *update-conv'*)

lemma *delete-Mapping* [*code*]:
Mapping.delete k (*Mapping xs*) = *Mapping* (*delete k xs*)
by (*rule mapping-eqI*) (*simp add*: *delete-conv'*)

lemma *ordered-keys-Mapping* [*code*]:
Mapping.ordered-keys (*Mapping xs*) = *sort* (*remdups* (*map fst xs*))
by (*simp only*: *ordered-keys-def keys-Mapping sorted-list-of-set-sort-remdups*)
simp

lemma *size-Mapping* [*code*]:
Mapping.size (*Mapping xs*) = *length* (*remdups* (*map fst xs*))
by (*simp add*: *size-def length-remdups-card-conv dom-map-of-conv-image-fst*)

lemma *tabulate-Mapping* [*code*]:
Mapping.tabulate ks f = *Mapping* (*map* (λ*k*. (*k*, *f k*)) *ks*)
by (*rule mapping-eqI*) (*simp add*: *map-of-map-restrict*)

lemma *bulkload-Mapping* [*code*]:
Mapping.bulkload vs = *Mapping* (*map* (λ*n*. (*n*, *vs* ! *n*)) [*0*..<*length vs*])
by (*rule mapping-eqI*) (*simp add*: *map-of-map-restrict expand-fun-eq*)

lemma [*code*, *code del*]:
HOL.eq (*x* :: (-, -) *mapping*) *y* ⟷ *x* = *y* **by** (*fact eq-equals*)

end

4 SetsAndFunctions: Operations on sets and functions

theory *SetsAndFunctions*
imports *Main*
begin

This library lifts operations like addition and muliplication to sets and

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

4.1 Basic definitions

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

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

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

instance ..

end

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

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

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

instance ..

end

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

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

instance ..

end

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

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

instance ..

end

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

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

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

instance *fun* :: (*type*,*semigroup-add*)*semigroup-add*
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*: *fun-Compl-def func-plus func-zero*)
apply (*simp add*: *fun-Compl-def func-plus fun-diff-def 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 fun-Compl-def fun-diff-def*
func-one func-zero algebra-simps)
apply (*drule fun-cong*)
apply *simp*
done

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

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

interpretation *set-semigroup-mult*: *semigroup-mult op* ⊗ :: ('*a*::*semigroup-mult*) *set* => '*a set* => '*a set*
apply *default*
apply (*unfold set-times-def*)
apply (*force simp add*: *mult-assoc*)
done

interpretation *set-comm-monoid-add*: *comm-monoid-add op* ⊕ :: ('*a*::*comm-monoid-add*) *set* => '*a set* => '*a set* {*0*}
apply *default*
apply (*unfold set-plus-def*)
apply (*force simp add*: *add-ac*)
apply *force*
done

interpretation *set-comm-monoid-mult*: *comm-monoid-mult op* ⊗ :: ('*a*::*comm-monoid-mult*) *set* => '*a set* => '*a set* {*1*}
apply *default*
apply (*unfold set-times-def*)
apply (*force simp add*: *mult-ac*)
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-def*)

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-def 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*: $((a::'a::semigroup\text{-}add) +o B) \oplus C = a +o (B \oplus C)$
apply (*auto simp add*: *elt-set-plus-def set-plus-def*)
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*: $C \oplus ((a::'a::comm\text{-}monoid\text{-}add) +o D) = a +o (C \oplus D)$
apply (*auto intro*!: *subsetI simp add*: *elt-set-plus-def set-plus-def 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*]: $(C::('a::plus)\ set) <= D ==> E <= F ==> C \oplus E <= D \oplus F$
by (*auto simp add*: *set-plus-def*)

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

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

lemma *set-plus-mono5*: $a:C ==> B <= D ==> a +o B <= C \oplus D$
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*: $C <= D ==> E <= F ==> x : C \oplus E ==>$
$x : D \oplus F$
apply (*frule set-plus-mono2*)
prefer *2*
apply *force*
apply *assumption*
done

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

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

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

lemma *set-zero-plus2*: $(0::'a::comm\text{-}monoid\text{-}add) : A ==> B <= A \oplus B$
apply (*auto intro*!: *subsetI simp add*: *set-plus-def*)
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*]: $a : C ==> b : D ==> a * b : C \otimes D$
by (*auto simp add*: *set-times-def*)

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

lemma *set-times-rearrange*: $((a::'a::comm\text{-}monoid\text{-}mult) *o\ C) \otimes (b *o\ D) = (a * b) *o\ (C \otimes D)$
apply (*auto simp add*: *elt-set-times-def set-times-def*)
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*: $((a::'a::semigroup\text{-}mult) *o\ B) \otimes C = a *o\ (B \otimes C)$
apply (*auto simp add*: *elt-set-times-def set-times-def*)
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*: $C \otimes ((a::'a::comm\text{-}monoid\text{-}mult) *o\ D) = a *o\ (C \otimes D)$
apply (*auto intro*!: *subsetI simp add*: *elt-set-times-def set-times-def 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*]: $(C::('a::times)\ set) <= D ==> E <= F ==> C \otimes E <= D \otimes F$
by (*auto simp add*: *set-times-def*)

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

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

lemma *set-times-mono5*: $a{:}C \Longrightarrow B <= D \Longrightarrow a *o B <= C \otimes D$
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*: $C <= D \Longrightarrow x : a *o C \Longrightarrow x : a *o D$
apply (*frule set-times-mono*)
apply *auto*
done

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

lemma *set-times-mono3-b*: $a : C \Longrightarrow x : a *o D \Longrightarrow x : C \otimes D$
apply (*frule set-times-mono3*)
apply *auto*
done

lemma *set-times-mono4-b*: $(a::'a::comm\text{-}monoid\text{-}mult) : C \Longrightarrow x : a *o D \Longrightarrow x : D \otimes C$
apply (*frule set-times-mono4*)
apply *auto*
done

lemma *set-one-times* [*simp*]: $(1::'a::comm\text{-}monoid\text{-}mult) *o C = C$
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*: $(a::'a::semiring) *o (B \oplus C) = (a *o B) \oplus (a *o C)$
apply (*auto simp add*: *set-plus-def 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*: $((a::'a::semiring) +o C) \otimes D <=$

$a *o D \oplus C \otimes D$
apply (*auto intro*!: *subsetI simp add*:
elt-set-plus-def elt-set-times-def set-times-def
set-plus-def 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 *Complex-Main SetsAndFunctions*
begin

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

The main changes in this version are as follows:

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

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

Note also since the Big O library includes rules that demonstrate set inclusion, to use the automated reasoners effectively with the library one

should redeclare the theorem *subsetI* as an intro rule, rather than as an *intro*! rule, for example, using **declare** *subsetI* [*del*, *intro*].

5.1 Definitions

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

lemma *bigo-pos-const*: (*EX* (*c*::'*a*::*linordered-idom*).
ALL x. (*abs* (*h x*)) <= (*c* * (*abs* (*f x*))))
= (*EX c*. *0* < *c* & (*ALL x*. (*abs*(*h x*)) <= (*c* * (*abs* (*f x*))))))
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-def*)
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-def*)
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-def 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 fun-Compl-def 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 fun-diff-def*)
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 fun-diff-def)
    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-def 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*: *algebra-simps*)
done

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

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 fun-Compl-def 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*: $(c::'a::linordered\text{-}field) \sim= 0 ==> (\%x.\ 1) : O(\%x.\ c)$
apply (*simp add*: *bigo-def*)
apply (*rule-tac x* = *abs*(*inverse c*) **in** *exI*)
apply (*simp add*: *abs-mult* [*symmetric*])
done

lemma *bigo-const4*: $(c::'a::linordered\text{-}field) \sim= 0 ==> O(\%x.\ 1) <= O(\%x.\ c)$
by (*rule bigo-elt-subset*, *rule bigo-const3*, *assumption*)

lemma *bigo-const* [*simp*]: $(c::'a::linordered\text{-}field) \sim= 0 ==>$
$O(\%x.\ c) = O(\%x.\ 1)$
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*: $(c::'a::linordered\text{-}field) \sim= 0 ==> f : O(\%x.\ c * f\ x)$
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*: $(c::'a::linordered\text{-}field) \sim= 0 ==>$
$O(f) <= O(\%x.\ c * f\ x)$
by (*rule bigo-elt-subset*, *rule bigo-const-mult3*, *assumption*)

lemma *bigo-const-mult* [*simp*]: $(c::'a::linordered\text{-}field) \sim= 0 ==>$
$O(\%x.\ c * f\ x) = O(f)$
by (*rule equalityI*, *rule bigo-const-mult2*, *erule bigo-const-mult4*)

lemma *bigo-const-mult5* [*simp*]: $(c::'a::linordered\text{-}field) \sim= 0 ==>$
$(\%x.\ c) *o O(f) = O(f)$
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 fun-Compl-def*
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 fun-diff-def*)
apply (*subst setsum-subtractf* [*symmetric*])
apply (*subst right-diff-distrib* [*symmetric*])
apply (*rule bigo-setsum3*)
apply (*subst fun-diff-def* [*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 fun-diff-def*)
apply (*subst setsum-subtractf* [*symmetric*])
apply (*subst right-diff-distrib* [*symmetric*])
apply (*rule bigo-setsum5*)
apply (*subst fun-diff-def* [*symmetric*])
apply (*drule set-plus-imp-minus*)
apply *auto*

done

5.3 Misc useful stuff

lemma *bigo-useful-intro*: $A <= O(f) ==> B <= O(f) ==>$
$A \oplus B <= O(f)$
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*: $(c::'a::linordered\text{-}field) \sim= 0 ==>$
$(\%x.\ c) * f =o O(h) ==> f =o O(h)$
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 fun-diff-def*)
apply (*subst fun-diff-def* [*symmetric*])
apply (*rule set-plus-imp-minus*)
apply *simp*
apply (*simp add*: *fun-diff-def*)
done

5.4 Less than or equal to

definition
lesso :: ('*a* => '*b*::*linordered-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 fun-diff-def*)
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*: *algebra-simps*)
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*: *algebra-simps*)
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 fun-diff-def*)

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*: *algebra-simps*)
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*: *algebra-simps*)
done

lemma *bigo-lesso4*: $f <o g =o O(k::'a=>'b::linordered\text{-}field) ==>$
$g =o h +o O(k) ==> f <o h =o O(k)$
apply (*unfold lesso-def*)
apply (*drule set-plus-imp-minus*)
apply (*drule bigo-abs5*) **back**
apply (*simp add*: *fun-diff-def*)
apply (*drule bigo-useful-add*)
apply *assumption*
apply (*erule bigo-lesseq2*) **back**
apply (*rule allI*)
apply (*auto simp add*: *func-plus fun-diff-def algebra-simps*
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*: *algebra-simps*)
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

lemma *bigo-LIMSEQ1*: *f* =*o* *O*(*g*) ==> *g* ----> *0* ==> *f* ----> (*0*::*real*)
apply (*simp add*: *LIMSEQ-iff bigo-alt-def*)
apply *clarify*
apply (*drule-tac x* = *r* / *c* **in** *spec*)
apply (*drule mp*)
apply (*erule divide-pos-pos*)
apply *assumption*
apply *clarify*
apply (*rule-tac x* = *no* **in** *exI*)
apply (*rule allI*)
apply (*drule-tac x* = *n* **in** *spec*)+
apply (*rule impI*)
apply (*drule mp*)
apply *assumption*
apply (*rule order-le-less-trans*)
apply *assumption*
apply (*rule order-less-le-trans*)
apply (*subgoal-tac c* * *abs*(*g n*) < *c* * (*r* / *c*))
apply *assumption*
apply (*erule mult-strict-left-mono*)
apply *assumption*
apply *simp*
done

lemma *bigo-LIMSEQ2*: *f* =*o* *g* +*o* *O*(*h*) ==> *h* ----> *0* ==> *f* ----> *a*
==> *g* ----> (*a*::*real*)
apply (*drule set-plus-imp-minus*)
apply (*drule bigo-LIMSEQ1*)
apply *assumption*
apply (*simp only*: *fun-diff-def*)
apply (*erule LIMSEQ-diff-approach-zero2*)
apply *assumption*
done

end

6 Binomial: Binomial Coefficients

theory *Binomial*
imports *Complex-Main*
begin

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

primrec *binomial* :: *nat* ⇒ *nat* ⇒ *nat* (**infixl** *choose 65*) **where**

binomial-0: $(0 \text{ choose } k) = (\text{if } k = 0 \text{ then } 1 \text{ else } 0)$
| *binomial-Suc*: $(\text{Suc } n \text{ choose } k) =$
$(\text{if } k = 0 \text{ then } 1 \text{ else } (n \text{ choose } (k - 1)) + (n \text{ choose } k))$

lemma *binomial-n-0* [*simp*]: $(n \text{ choose } 0) = 1$
by (*cases n*) *simp-all*

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

lemma *binomial-Suc-Suc* [*simp*]:
$(\text{Suc } n \text{ choose } \text{Suc } k) = (n \text{ choose } k) + (n \text{ choose } \text{Suc } k)$
by *simp*

lemma *binomial-eq-0*: $!!k.\ n < k \Longrightarrow (n \text{ choose } k) = 0$
by (*induct n*) *auto*

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

lemma *binomial-n-n* [*simp*]: $(n \text{ choose } n) = 1$
by (*induct n*) (*simp-all add*: *binomial-eq-0*)

lemma *binomial-Suc-n* [*simp*]: $(\text{Suc } n \text{ choose } n) = \text{Suc } n$
by (*induct n*) *simp-all*

lemma *binomial-1* [*simp*]: $(n \text{ choose } \text{Suc } 0) = n$
by (*induct n*) *simp-all*

lemma *zero-less-binomial*: $k \leq n \Longrightarrow (n \text{ choose } k) > 0$
by (*induct n k rule*: *diff-induct*) *simp-all*

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

lemma *zero-less-binomial-iff*: $(n \text{ choose } k > 0) = (k \leq n)$
by(*simp add*: *linorder-not-less binomial-eq-0-iff neq0-conv*[*symmetric*]
del:*neq0-conv*)

lemma *Suc-times-binomial-eq*:
$!!k.\ k \leq n \Longrightarrow \text{Suc } n * (n \text{ choose } k) = (\text{Suc } n \text{ choose } \text{Suc } k) * \text{Suc } k$
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*:
$k \leq n$ ==> (*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*:
[|$k \leq n$; 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*. $B \subseteq A$ & *card B* = *0*} = *1*
by (*simp cong add*: *conj-cong add*: *finite-subset* [*THEN card-0-eq*])

lemma *choose-deconstruct*: *finite M* ==> $x \notin M$
==> {*s*. *s* <= *insert x M* & *card*(*s*) = *Suc k*}
= {*s*. *s* <= *M* & *card*(*s*) = *Suc k*} *Un*
{*s*. *EX t*. *t* <= *M* & *card*(*t*) = *k* & *s* = *insert x t*}
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

lemma *finite-bex-subset*[*simp*]:
finite B ⟹ (!!*A*. *A*<=*B* ⟹ *finite*{*x*. *P x A*}) ⟹ *finite*{*x*. *EX A*<=*B*. *P x A*}
apply(*subgoal-tac* {*x*. *EX A*<=*B*. *P x A*} = (*UN A*:*Pow B*. {*x*. *P x A*}))
apply *simp*
apply *blast*
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*; $x \notin A$|] ==>
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)
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 $\ldots = a\hat{}(n+1) + b\hat{}(n+1) +$
$(\sum k{=}1..n.\ (n\ choose\ (k-1)) * a\hat{}k * b\hat{}(n+1-k)) +$
$(\sum k{=}1..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n+1-k))$
by (*simp add*: *decomp2*)
also have
$\ldots = a\hat{}(n+1) + b\hat{}(n+1) + (\sum k{=}1..n.\ (n+1\ choose\ k) * a\hat{}k * b\hat{}(n+1-k))$
by (*simp add*: *nat-distrib setsum-addf binomial.simps*)
also have $\ldots = (\sum k{=}0..n+1.\ (n+1\ choose\ k) * a\hat{}k * b\hat{}(n+1-k))$
using *decomp* **by** *simp*
finally show *?case* **by** *simp*
qed

6.2 Pochhammer's symbol : generalized raising factorial

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

lemma *pochhammer-0*[*simp*]: *pochhammer a 0* = *1*
by (*simp add*: *pochhammer-def*)

lemma *pochhammer-1*[*simp*]: *pochhammer a 1* = *a* **by** (*simp add*: *pochhammer-def*)
lemma *pochhammer-Suc0*[*simp*]: *pochhammer a* (*Suc 0*) = *a*
by (*simp add*: *pochhammer-def*)

lemma *pochhammer-Suc-setprod*: *pochhammer a* (*Suc n*) = *setprod* ($\lambda n.\ a +$ *of-nat* n) $\{0\ ..\ n\}$
by (*simp add*: *pochhammer-def*)

lemma *setprod-nat-ivl-Suc*: *setprod f* $\{0\ ..$ *Suc n*$\}$ = *setprod f* $\{0..n\} *$ *f* (*Suc n*)
proof−
have *th*: *finite* $\{0..n\}$ *finite* $\{$*Suc n*$\}$ $\{0..n\} \cap \{$*Suc n*$\} = \{\}$ **by** *auto*
have *eq*: $\{0..$*Suc n*$\} = \{0..n\} \cup \{$*Suc n*$\}$ **by** *auto*
show *?thesis* **unfolding** *eq setprod-Un-disjoint*[*OF th*] **by** *simp*
qed

lemma *setprod-nat-ivl-1-Suc*: *setprod f* $\{0\ ..$ *Suc n*$\}$ = *f 0* $*$ *setprod f* $\{1..$ *Suc n*$\}$
proof−
have *th*: *finite* $\{0\}$ *finite* $\{1..$*Suc n*$\}$ $\{0\} \cap \{1..$ *Suc n*$\} = \{\}$ **by** *auto*
have *eq*: $\{0..$*Suc n*$\} = \{0\} \cup \{1\ ..$ *Suc n*$\}$ **by** *auto*
show *?thesis* **unfolding** *eq setprod-Un-disjoint*[*OF th*] **by** *simp*
qed

lemma *pochhammer-Suc*: *pochhammer a* (*Suc n*) = *pochhammer a n* $*$ (*a* + *of-nat* *n*)
proof−
{assume *n*=*0* **then have** *?thesis* **by** *simp***}**
moreover
{fix *m* **assume** *m*: *n* = *Suc m*

```
    have ?thesis  unfolding m pochhammer-Suc-setprod setprod-nat-ivl-Suc ..}
  ultimately show ?thesis by (cases n, auto)
qed

lemma pochhammer-rec: pochhammer a (Suc n) = a * pochhammer (a + 1) n
proof-
  {assume n=0 then have ?thesis by (simp add: pochhammer-Suc-setprod)}
  moreover
  {assume n0: n ≠ 0
    have th0: finite {1 .. n} 0 ∉ {1 .. n} by auto
    have eq: insert 0 {1 .. n} = {0..n} by auto
    have th1: (∏ n∈{1::nat..n}. a + of-nat n) =
      (∏ n∈{0::nat..n - 1}. a + 1 + of-nat n)
      apply (rule setprod-reindex-cong[where f = Suc])
      using n0 by (auto simp add: expand-fun-eq field-simps)
    have ?thesis apply (simp add: pochhammer-def)
    unfolding setprod-insert[OF th0, unfolded eq]
    using th1 by (simp add: field-simps)}
ultimately show ?thesis by blast
qed

lemma pochhammer-fact: of-nat (fact n) = pochhammer 1 n
  unfolding fact-altdef-nat

  apply (cases n, simp-all add: of-nat-setprod pochhammer-Suc-setprod)
  apply (rule setprod-reindex-cong[where f=Suc])
  by (auto simp add: expand-fun-eq)

lemma pochhammer-of-nat-eq-0-lemma: assumes kn: k > n
  shows pochhammer (- (of-nat n :: 'a:: idom)) k = 0
proof-
  from kn obtain h where h: k = Suc h by (cases k, auto)
  {assume n0: n=0 then have ?thesis using kn
     by (cases k, simp-all add: pochhammer-rec del: pochhammer-Suc)}
  moreover
  {assume n0: n ≠ 0
    then have ?thesis apply (simp add: h pochhammer-Suc-setprod)
  apply (rule-tac x=n in bexI)
  using h kn by auto}
ultimately show ?thesis by blast
qed

lemma pochhammer-of-nat-eq-0-lemma': assumes kn: k ≤ n
  shows pochhammer (- (of-nat n :: 'a:: {idom, ring-char-0})) k ≠ 0
proof-
  {assume k=0 then have ?thesis by simp}
  moreover
  {fix h assume h: k = Suc h
    then have ?thesis apply (simp add: pochhammer-Suc-setprod)
```

```
    using h kn by (auto simp add: algebra-simps)}
  ultimately show ?thesis by (cases k, auto)
qed

lemma pochhammer-of-nat-eq-0-iff:
  shows pochhammer (− (of-nat n :: 'a:: {idom, ring-char-0})) k = 0 ⟷ k > n
  (is ?l = ?r)
  using pochhammer-of-nat-eq-0-lemma[of n k, where ?'a='a]
    pochhammer-of-nat-eq-0-lemma'[of k n, where ?'a = 'a]
  by (auto simp add: not-le[symmetric])


lemma pochhammer-eq-0-iff:
  pochhammer a n = (0::'a::field-char-0) ⟷ (EX k < n . a = − of-nat k)
  apply (auto simp add: pochhammer-of-nat-eq-0-iff)
 apply (cases n, auto simp add: pochhammer-def algebra-simps group-add-class.eq-neg-iff-add-eq-0)
  apply (rule-tac x=x in exI)
  apply auto
  done


lemma pochhammer-eq-0-mono:
  pochhammer a n = (0::'a::field-char-0) ⟹ m ≥ n ⟹ pochhammer a m = 0
  unfolding pochhammer-eq-0-iff by auto

lemma pochhammer-neq-0-mono:
  pochhammer a m ≠ (0::'a::field-char-0) ⟹ m ≥ n ⟹ pochhammer a n ≠ 0
  unfolding pochhammer-eq-0-iff by auto

lemma pochhammer-minus:
  assumes kn: k ≤ n
  shows pochhammer (− b) k = ((− 1) ^ k :: 'a::comm-ring-1) * pochhammer (b
− of-nat k + 1) k
proof−
  {assume k0: k = 0 then have ?thesis by simp}
  moreover
  {fix h assume h: k = Suc h
    have eq: ((− 1) ^ Suc h :: 'a) = setprod (%i. − 1) {0 .. h}
      using setprod-constant[where A={0 .. h} and y=− 1 :: 'a]
      by auto
    have ?thesis
      unfolding h h pochhammer-Suc-setprod eq setprod-timesf[symmetric]
      apply (rule strong-setprod-reindex-cong[where f = %i. h − i])
      apply (auto simp add: inj-on-def image-def h )
      apply (rule-tac x=h − x in bexI)
      by (auto simp add: expand-fun-eq h of-nat-diff)}
  ultimately show ?thesis by (cases k, auto)
qed
```

lemma *pochhammer-minus'*:
assumes *kn*: $k \leq n$
shows *pochhammer* $(b - \textit{of-nat}\ k + 1)\ k = ((-1)\ \hat{}\ k :: 'a{::}\textit{comm-ring-1}) *$
pochhammer $(-b)\ k$
unfolding *pochhammer-minus*[*OF kn*, **where** *b*=*b*]
unfolding *mult-assoc*[*symmetric*]
unfolding *power-add*[*symmetric*]
apply *simp*
done

lemma *pochhammer-same*: *pochhammer* $(-\textit{of-nat}\ n)\ n = ((-1)\ \hat{}\ n :: 'a{::}\textit{comm-ring-1})$
$*$ *of-nat* $(\textit{fact}\ n)$
unfolding *pochhammer-minus*[*OF le-refl*[*of n*]]
by (*simp add*: *of-nat-diff pochhammer-fact*)

6.3 Generalized binomial coefficients

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

lemma *gbinomial-0*[*simp*]: *a gchoose 0* = *1 0 gchoose* (*Suc n*) = *0*
apply (*simp-all add*: *gbinomial-def*)
apply (*subgoal-tac* $(\prod i{::}nat{\in}\{0{::}nat..n\}.\ -\ \textit{of-nat}\ i) = (0::'b)$)
apply (*simp del*:*setprod-zero-iff*)
apply *simp*
done

lemma *gbinomial-pochhammer*: *a gchoose n* = $(-1)\ \hat{}\ n *$ *pochhammer* $(-a)\ n$
/ *of-nat* (*fact n*)
proof−
{assume *n*=*0* **then have** *?thesis* **by** *simp***}**
moreover
{assume *n0*: *n*≠*0*
from *n0 setprod-constant*[*of* $\{0\ ..\ n - 1\} - (1::'a)$]
have *eq*: $(-(1::'a))\ \hat{}\ n = \textit{setprod}\ (\lambda i.\ -1)\ \{0\ ..\ n - 1\}$
by *auto*
from *n0* **have** *?thesis*
by (*simp add*: *pochhammer-def gbinomial-def field-simps eq setprod-timesf*[*symmetric*])**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *binomial-fact-lemma*:
$k \leq n \Longrightarrow \textit{fact}\ k * \textit{fact}\ (n - k) * (n\ \textit{choose}\ k) = \textit{fact}\ n$
proof(*induct n arbitrary*: *k rule*: *nat-less-induct*)
fix *n k* **assume** *H*: $\forall m{<}n.\ \forall x{\leq}m.\ \textit{fact}\ x * \textit{fact}\ (m - x) * (m\ \textit{choose}\ x) =$
fact m **and** *kn*: $k \leq n$
let *?ths* = *fact k* $*$ *fact* $(n - k) *$ (*n choose k*) = *fact n*
{assume *n*=*0* **then have** *?ths* **using** *kn* **by** *simp***}**

```
  moreover
  {assume k=0 then have ?ths using kn by simp}
  moreover
  {assume nk: n=k then have ?ths by simp}
  moreover
  {fix m h assume n: n = Suc m and h: k = Suc h and hm: h < m
   from n have mn: m < n by arith
   from hm have hm': h ≤ m by arith
   from hm h n kn have km: k ≤ m by arith
   have m - h = Suc (m - Suc h) using  h km hm by arith
   with km h have th0: fact (m - h) = (m - h) * fact (m - k)
    by simp
   from n h th0
   have fact k * fact (n - k) * (n choose k) = k * (fact h * fact (m - h) * (m
choose h)) +  (m - h) * (fact k * fact (m - k) * (m choose k))
    by (simp add: field-simps)
   also have ... = (k + (m - h)) * fact m
    using H[rule-format, OF mn hm'] H[rule-format, OF mn km]
    by (simp add: field-simps)
   finally have ?ths using h n km by simp}
  moreover have n=0 ∨ k = 0 ∨ k = n ∨ (EX m h. n=Suc m ∧ k = Suc h ∧
h < m) using kn by presburger
  ultimately show ?ths by blast
qed

lemma binomial-fact:
  assumes kn: k ≤ n
  shows (of-nat (n choose k) :: 'a::field-char-0) = of-nat (fact n) / (of-nat (fact
k) * of-nat (fact (n - k)))
  using binomial-fact-lemma[OF kn]
  by (simp add: field-simps of-nat-mult [symmetric])

lemma binomial-gbinomial: of-nat (n choose k) = of-nat n gchoose k
proof-
  {assume kn: k > n
   from kn binomial-eq-0[OF kn] have ?thesis
    by (simp add: gbinomial-pochhammer field-simps
      pochhammer-of-nat-eq-0-iff)}
  moreover
  {assume k=0 then have ?thesis by simp}
  moreover
  {assume kn: k ≤ n and k0: k≠ 0
   from k0 obtain h where h: k = Suc h by (cases k, auto)
   from h
   have eq:(- 1 :: 'a) ^ k = setprod (λi. - 1) {0..h}
    by (subst setprod-constant, auto)
   have eq': (∏ i∈{0..h}. of-nat n + - (of-nat i :: 'a)) = (∏ i∈{n - h..n}. of-nat
i)
    apply (rule strong-setprod-reindex-cong[where f=op - n])
```

```
    using h kn
    apply (simp-all add: inj-on-def image-iff Bex-def expand-set-eq)
    apply clarsimp
    apply (presburger)
    apply presburger
   by (simp add: expand-fun-eq field-simps of-nat-add[symmetric] del: of-nat-add)
  have th0: finite {1..n - Suc h} finite {n - h .. n}
{1..n - Suc h} ∩ {n - h .. n} = {} and eq3: {1..n - Suc h} ∪ {n - h .. n} =
{1..n} using h kn by auto
  from eq[symmetric]
  have ?thesis using kn
    apply (simp add: binomial-fact[OF kn, where ?'a = 'a]
      gbinomial-pochhammer field-simps pochhammer-Suc-setprod)
     apply (simp add: pochhammer-Suc-setprod fact-altdef-nat h of-nat-setprod
setprod-timesf[symmetric] eq' del: One-nat-def power-Suc)
     unfolding setprod-Un-disjoint[OF th0, unfolded eq3, of of-nat:: nat ⇒ 'a]
eq[unfolded h]
    unfolding mult-assoc[symmetric]
    unfolding setprod-timesf[symmetric]
    apply simp
    apply (rule strong-setprod-reindex-cong[where f= op - n])
    apply (auto simp add: inj-on-def image-iff Bex-def)
    apply presburger
    apply (subgoal-tac (of-nat (n - x) :: 'a) = of-nat n - of-nat x)
    apply simp
    by (rule of-nat-diff, simp)
 }
 moreover
 have k > n ∨ k = 0 ∨ (k ≤ n ∧ k ≠ 0) by arith
 ultimately show ?thesis by blast
qed

lemma gbinomial-1[simp]: a gchoose 1 = a
 by (simp add: gbinomial-def)

lemma gbinomial-Suc0[simp]: a gchoose (Suc 0) = a
 by (simp add: gbinomial-def)

lemma gbinomial-mult-1: a * (a gchoose n) = of-nat n * (a gchoose n) + of-nat
(Suc n) * (a gchoose (Suc n)) (is ?l = ?r)
proof-
 have ?r = ((- 1) ^n * pochhammer (- a) n / of-nat (fact n)) * (of-nat n -
(- a + of-nat n))
   unfolding gbinomial-pochhammer
   pochhammer-Suc fact-Suc of-nat-mult right-diff-distrib power-Suc
   by (simp add: field-simps del: of-nat-Suc)
 also have ... = ?l unfolding gbinomial-pochhammer
   by (simp add: field-simps)
 finally show ?thesis ..
```

qed

lemma *gbinomial-mult-1'*: $(a\ gchoose\ n) * a = of\text{-}nat\ n * (a\ gchoose\ n) + of\text{-}nat\ (Suc\ n) * (a\ gchoose\ (Suc\ n))$
by (*simp add*: *mult-commute gbinomial-mult-1*)

lemma *gbinomial-Suc*: $a\ gchoose\ (Suc\ k) = (setprod\ (\lambda i.\ a - of\text{-}nat\ i)\ \{0\ ..\ k\})\ /\ of\text{-}nat\ (fact\ (Suc\ k))$
by (*simp add*: *gbinomial-def*)

lemma *gbinomial-mult-fact*:
$(of\text{-}nat\ (fact\ (Suc\ k)) :: 'a) * ((a::'a::field\text{-}char\text{-}0)\ gchoose\ (Suc\ k)) = (setprod\ (\lambda i.\ a - of\text{-}nat\ i)\ \{0\ ..\ k\})$
unfolding *gbinomial-Suc*
by (*simp-all add*: *field-simps del*: *fact-Suc*)

lemma *gbinomial-mult-fact'*:
$((a::'a::field\text{-}char\text{-}0)\ gchoose\ (Suc\ k)) * (of\text{-}nat\ (fact\ (Suc\ k)) :: 'a) = (setprod\ (\lambda i.\ a - of\text{-}nat\ i)\ \{0\ ..\ k\})$
using *gbinomial-mult-fact*[*of k a*]
apply (*subst mult-commute*) **.**

lemma *gbinomial-Suc-Suc*: $((a::'a::field\text{-}char\text{-}0) + 1)\ gchoose\ (Suc\ k) = a\ gchoose\ k + (a\ gchoose\ (Suc\ k))$
proof−
{assume $k = 0$ **then have** *?thesis* **by** *simp***}**
moreover
{fix h **assume** h: $k = Suc\ h$
have *eq0*: $(\prod i{\in}\{1..k\}.\ (a + 1) - of\text{-}nat\ i) = (\prod i{\in}\{0..h\}.\ a - of\text{-}nat\ i)$
apply (*rule strong-setprod-reindex-cong*[**where** $f = Suc$])
using h **by** *auto*

have $of\text{-}nat\ (fact\ (Suc\ k)) * (a\ gchoose\ k + (a\ gchoose\ (Suc\ k))) = ((a\ gchoose\ Suc\ h) * of\text{-}nat\ (fact\ (Suc\ h)) * of\text{-}nat\ (Suc\ k)) + (\prod i{\in}\{0::nat..Suc\ h\}.\ a - of\text{-}nat\ i)$
unfolding h
apply (*simp add*: *field-simps del*: *fact-Suc*)
unfolding *gbinomial-mult-fact'*
apply (*subst fact-Suc*)
unfolding *of-nat-mult*
apply (*subst mult-commute*)
unfolding *mult-assoc*
unfolding *gbinomial-mult-fact*
by (*simp add*: *field-simps*)
also have $\ldots = (\prod i{\in}\{0..h\}.\ a - of\text{-}nat\ i) * (a + 1)$
unfolding *gbinomial-mult-fact' setprod-nat-ivl-Suc*
by (*simp add*: *field-simps h*)
also have $\ldots = (\prod i{\in}\{0..k\}.\ (a + 1) - of\text{-}nat\ i)$
using *eq0*

unfolding *h setprod-nat-ivl-1-Suc*
by *simp*
also have ... = *of-nat* (*fact* (*Suc k*)) * ((*a* + *1*) *gchoose* (*Suc k*))
unfolding *gbinomial-mult-fact* **..**
finally have *?thesis* **by** (*simp del*: *fact-Suc*) **}**
ultimately show *?thesis* **by** (*cases k*, *auto*)
qed

lemma *binomial-symmetric*: **assumes** *kn*: $k \leq n$
shows *n choose k* = *n choose* ($n - k$)
proof−
from *kn* **have** *kn'*: $n - k \leq n$ **by** *arith*
from *binomial-fact-lemma*[*OF kn*] *binomial-fact-lemma*[*OF kn'*]
have *fact k* * *fact* ($n - k$) * (*n choose k*) = *fact* ($n - k$) * *fact* ($n - (n - k)$)
* (*n choose* ($n - k$)) **by** *simp*
then show *?thesis* **using** *kn* **by** *simp*
qed

end

7 Bit: The Field of Integers mod 2

theory *Bit*
imports *Main*
begin

7.1 Bits as a datatype

typedef (**open**) *bit* = *UNIV* :: *bool set* **..**

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

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

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

instance ..

end

rep-datatype (*bit*) *0*::*bit 1*::*bit*
proof −
fix *P* **and** *x* :: *bit*
assume *P* (*0*::*bit*) **and** *P* (*1*::*bit*)

then have $\forall b.\ P\ (Abs\text{-}bit\ b)$
unfolding *zero-bit-def one-bit-def*
by (*simp add*: *all-bool-eq*)
then show $P\ x$
by (*induct x*) *simp*
next
show $(0::bit) \neq (1::bit)$
unfolding *zero-bit-def one-bit-def*
by (*simp add*: *Abs-bit-inject*)
qed

lemma *bit-not-0-iff* [*iff*]: $(x::bit) \neq 0 \longleftrightarrow x = 1$
by (*induct x*) *simp-all*

lemma *bit-not-1-iff* [*iff*]: $(x::bit) \neq 1 \longleftrightarrow x = 0$
by (*induct x*) *simp-all*

7.2 Type *bit* forms a field

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

definition *plus-bit-def*:
$x + y = bit\text{-}case\ y\ (bit\text{-}case\ 1\ 0\ y)\ x$

definition *times-bit-def*:
$x * y = bit\text{-}case\ 0\ y\ x$

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

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

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

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

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

instance proof
qed (*unfold field-bit-defs*, *auto split*: *bit.split*)

end

lemma *bit-add-self*: $x + x = (0 :: bit)$
unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

lemma *bit-mult-eq-1-iff* [*simp*]: $x * y = (1 :: bit) \longleftrightarrow x = 1 \wedge y = 1$
unfolding *times-bit-def* **by** (*simp split*: *bit.split*)

Not sure whether the next two should be simp rules.

lemma *bit-add-eq-0-iff*: $x + y = (0 :: bit) \longleftrightarrow x = y$
unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

lemma *bit-add-eq-1-iff*: $x + y = (1 :: bit) \longleftrightarrow x \neq y$
unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

7.3 Numerals at type *bit*

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

definition *number-of-bit-def*:
(*number-of w* :: *bit*) = *of-int w*

instance proof
qed (*rule number-of-bit-def*)

end

All numerals reduce to either 0 or 1.

lemma *bit-minus1* [*simp*]: $-1 = (1 :: bit)$
by (*simp only*: *number-of-Min uminus-bit-def*)

lemma *bit-number-of-even* [*simp*]: *number-of* (*Int.Bit0 w*) = $(0 :: bit)$
by (*simp only*: *number-of-Bit0 add-0-left bit-add-self*)

lemma *bit-number-of-odd* [*simp*]: *number-of* (*Int.Bit1 w*) = $(1 :: bit)$
by (*simp only*: *number-of-Bit1 add-assoc bit-add-self*
monoid-add-class.add-0-right)

end

8 Boolean-Algebra: Boolean Algebras

theory *Boolean-Algebra*
imports *Main*
begin

locale *boolean* =
fixes *conj* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\sqcap$ *70*)
fixes *disj* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\sqcup$ *65*)

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

sublocale *boolean* < *conj*!: *abel-semigroup conj* **proof**
qed (*fact conj-assoc conj-commute*)+

sublocale *boolean* < *disj*!: *abel-semigroup disj* **proof**
qed (*fact disj-assoc disj-commute*)+

context *boolean*
begin

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

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

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

lemma *dual*: *boolean disj conj compl one zero*
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

8.1 Complement

lemma *complement-unique*:
assumes *1*: $a \sqcap x = \mathbf{0}$
assumes *2*: $a \sqcup x = \mathbf{1}$

assumes *3*: $a \sqcap y = \mathbf{0}$
assumes *4*: $a \sqcup y = \mathbf{1}$
shows $x = y$
proof −
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*)

8.2 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

8.3 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

8.4 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

8.5 Symmetric Difference

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

sublocale *boolean-xor* < *xor*!: *abel-semigroup xor* **proof**
  fix $x\ y\ z$ :: $'a$
  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
  show $x \oplus y = y \oplus x$
    by (*simp only*: *xor-def conj-commute disj-commute*)
qed

context *boolean-xor*
begin

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

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

lemma *xor-def2*:
$x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$
by (*simp only*: *xor-def conj-disj-distribs*
disj-ac conj-ac conj-cancel-right disj-zero-left)

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

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

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

9 Product-ord: Order on product types

theory *Product-ord*
imports *Main*
begin

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

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

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

instance ..

end

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

instance $*$ :: (*preorder*, *preorder*) *preorder* **proof**
qed (*auto simp*: *prod-le-def prod-less-def less-le-not-le intro*: *order-trans*)

instance $*$ :: (*order*, *order*) *order* **proof**
qed (*auto simp add*: *prod-le-def*)

instance $*$ :: (*linorder*, *linorder*) *linorder* **proof**
qed (*auto simp*: *prod-le-def*)

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

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

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

instance proof
qed (*auto simp add*: *inf-prod-def sup-prod-def min-max.sup-inf-distrib1*)

end

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

definition
bot-prod-def: *bot* = (*bot*, *bot*)

instance proof
qed (*auto simp add*: *bot-prod-def prod-le-def*)

end

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

definition
top-prod-def: *top* = (*top*, *top*)

instance proof
qed (*auto simp add*: *top-prod-def prod-le-def*)

end

end

10 Char-nat: Mapping between characters and natural numbers

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

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

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

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

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

unfolding *nibble-of-nat-def mod-mod-trivial* **..**

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

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

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

lemma *nibble-of-nat-of-nibble*: *nibble-of-nat* (*nat-of-nibble n*) = *n*
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 ∈ {*15*, *14*, *13*, *12*, *11*, *10*, *9*, *8*, *7*, *6*, *5*, *4*, *3*, *2*, *1*, *0*}
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* *0*))))))))))))))} **by** *simp*
from *this* [*simplified set-unfold atLeastAtMost-singleton*]
show *?thesis* **by** (*simp add*: *numeral-2-eq-2* [*symmetric*])
qed
then show *?thesis* **unfolding** *nibble-of-nat-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*]:
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* **..**

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

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

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

lemma *Char-char-of-nat*:
Char n m = *char-of-nat* (*nat-of-nibble n* ∗ *16* + *nat-of-nibble m*)
unfolding *char-of-nat-def Let-def nibble-pair-of-nat-def*
by (*auto simp add*: *div-add1-eq mod-add-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-self2-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-add-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 String

code-modulename *OCaml*
Char-nat String

code-modulename *Haskell*

Char-nat String

end

11 Char-ord: Order on characters

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

instantiation *nibble* :: *linorder*
begin

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

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

instance proof
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 $n < m \longleftrightarrow n \leq m \wedge \neg\ m \leq n$
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

end

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

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

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

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

end

instantiation *char* :: *linorder*
begin

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

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

instance
by *default* (*auto simp*: *char-less-eq-def char-less-def split*: *char.splits*)

end

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

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

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

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

end

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

unfolding *char-less-eq-def char-less-def* **by** *simp-all*

end

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

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

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

setup ⟪
fold String-Code.add-literal-char [*SML*, *OCaml*, *Haskell*, *Scala*]
#> *String-Code.add-literal-list-string Haskell*
⟫

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

code-reserved *SML*
char

code-reserved *OCaml*
char

code-reserved *Scala*
char

code-const *eq-class.eq* :: *char* ⇒ *char* ⇒ *bool*
(*SML* !((- : *char*) = -))
(*OCaml* !((- : *char*) = -))
(*Haskell* **infixl** *4* ==)
(*Scala* **infixl** *5* ==)

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

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

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

lemma [*code*]:
literal-case f s = *f* (*explode s*)
literal-rec f s = *f* (*explode s*)
by (*cases s*, *simp*)+

code-reserved *SML String*

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

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

end

13 Code-Integer: Pretty integer literals for code generation

theory *Code-Integer*
imports *Main*
begin

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

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

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

setup ⟪
fold (*Numeral.add-code* @{*const-name number-int-inst.number-of-int*}
true Code-Printer.literal-numeral) [*SML*, *OCaml*, *Haskell*, *Scala*]
#> *Numeral.add-code* @{*const-name number-int-inst.number-of-int*}

```
    true Code-Printer.literal-numeral Scala
⟩⟩

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


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

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

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

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

code-const op - :: int ⇒ int ⇒ int
  (SML IntInf.- ((-), (-)))
  (OCaml Big'-int.sub'-big'-int)
```

(*Haskell* **infixl** *6* −)
(*Scala* **infixl** *7* −)

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

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

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

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

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

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

Evaluation

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

end

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

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

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

lemma [*code*]:
nat-of-char = nat o int-of-char
unfolding *int-of-char-def* **by** (*simp add*: *expand-fun-eq*)

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

lemma [*code*]:
char-of-nat = char-of-int o int
unfolding *char-of-int-def* **by** (*simp add*: *expand-fun-eq*)

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

end

15 Continuity: Continuity and iterations (of set transformers)

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

15.1 Continuity for complete lattices

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

definition
continuous :: (*'a*::*complete-lattice* ⇒ *'a*::*complete-lattice*) ⇒ *bool* **where**

continuous F ⟷ (∀ *M*. *chain M* ⟶ *F* (*SUP i*. *M i*) = (*SUP i*. *F* (*M i*)))

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

15.2 Chains

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

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

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

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

lemma *up-chain-mono*: *up-chain F* ==> *x* ≤ *y* ==> *F x* ⊆ *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 = (∀ *i*. *F* (*Suc i*) ⊆ *F i*)

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

by (*simp add*: *down-chain-def*)

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

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

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

15.3 Continuity

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

lemma *up-contI*:
(!!*F*. *up-chain F* ==> *f* (⋃(*range F*)) = ⋃(*f* ' *range F*)) ==> *up-cont f*
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* = λ*i*. *if i* = *0 then x else y* **in** *up-contD*)
apply (*rule up-chainI*)
apply *simp*
apply (*drule Un-absorb1*)
apply (*auto split*:*split-if-asm*)
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 split:split-if-asm*)
done

15.4 Iteration

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

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

16 ContNotDenum: Non-denumerability of the Continuum.

theory *ContNotDenum*
imports *Complex-Main*
begin

16.1 Abstract

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

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

Outline: An elegant informal proof of this result uses Cantor's Diagonalisation argument. The proof presented here is not this one. First we formalise

some properties of closed intervals, then we prove the Nested Interval Property. This property relies on the completeness of the Real numbers and is the foundation for our argument. Informally it states that an intersection of countable closed intervals (where each successive interval is a subset of the last) is non-empty. We then assume a surjective function f:ℕ⇒ℝ exists and find a real x such that x is not in the range of f by generating a sequence of closed intervals then using the NIP.

16.2 Closed Intervals

This section formalises some properties of closed intervals.

16.2.1 Definition

definition
closed-int :: *real* ⇒ *real* ⇒ *real set* **where**
closed-int x y = {*z*. *x* ≤ *z* ∧ *z* ≤ *y*}

16.2.2 Properties

lemma *closed-int-subset*:
assumes *xy*: *x1* ≥ *x0 y1* ≤ *y0*
shows *closed-int x1 y1* ⊆ *closed-int x0 y0*
proof −
{
fix *x*::*real*
assume *x* ∈ *closed-int x1 y1*
hence *x* ≥ *x1* ∧ *x* ≤ *y1* **by** (*simp add*: *closed-int-def*)
with *xy* **have** *x* ≥ *x0* ∧ *x* ≤ *y0* **by** *auto*
hence *x* ∈ *closed-int x0 y0* **by** (*simp add*: *closed-int-def*)
}
thus *?thesis* **by** *auto*
qed

lemma *closed-int-least*:
assumes *a*: *a* ≤ *b*
shows *a* ∈ *closed-int a b* ∧ (∀ *x* ∈ *closed-int a b*. *a* ≤ *x*)
proof
from *a* **have** *a*∈{*x*. *a*≤*x* ∧ *x*≤*b*} **by** *simp*
thus *a* ∈ *closed-int a b* **by** (*unfold closed-int-def*)
next
have ∀ *x*∈{*x*. *a*≤*x* ∧ *x*≤*b*}. *a*≤*x* **by** *simp*
thus ∀ *x* ∈ *closed-int a b*. *a* ≤ *x* **by** (*unfold closed-int-def*)
qed

lemma *closed-int-most*:
assumes *a*: *a* ≤ *b*
shows *b* ∈ *closed-int a b* ∧ (∀ *x* ∈ *closed-int a b*. *x* ≤ *b*)

proof
from *a* **have** $b \in \{x.\ a \leq x \wedge x \leq b\}$ **by** *simp*
thus $b \in$ *closed-int a b* **by** (*unfold closed-int-def*)
next
have $\forall x \in \{x.\ a \leq x \wedge x \leq b\}.\ x \leq b$ **by** *simp*
thus $\forall x \in$ *closed-int a b*. $x \leq b$ **by** (*unfold closed-int-def*)
qed

lemma *closed-not-empty*:
shows $a \leq b \Longrightarrow \exists x.\ x \in$ *closed-int a b*
by (*auto dest*: *closed-int-least*)

lemma *closed-mem*:
assumes $a \leq c$ **and** $c \leq b$
shows $c \in$ *closed-int a b*
using *assms* **unfolding** *closed-int-def* **by** *auto*

lemma *closed-subset*:
assumes *ac*: $a \leq b \;\; c \leq d$
assumes *closed*: *closed-int a b* $\subseteq$ *closed-int c d*
shows $b \geq c$
proof −
from *closed* **have** $\forall x \in$*closed-int a b*. $x \in$*closed-int c d* **by** *auto*
hence $\forall x.\ a \leq x \wedge x \leq b \longrightarrow c \leq x \wedge x \leq d$ **by** (*unfold closed-int-def*, *auto*)
with *ac* **have** $c \leq b \wedge b \leq d$ **by** *simp*
thus *?thesis* **by** *auto*
qed

16.3 Nested Interval Property

theorem *NIP*:
fixes *f*::*nat* ⇒ *real set*
assumes *subset*: $\forall n.\ f\ (Suc\ n) \subseteq f\ n$
and *closed*: $\forall n.\ \exists a\ b.\ f\ n =$ *closed-int a b* $\wedge\ a \leq b$
shows $(\bigcap n.\ f\ n) \neq \{\}$
proof −
let *?g* = $\lambda n.\ (SOME\ c.\ c \in (f\ n) \wedge (\forall x \in (f\ n).\ c \leq x))$
have *ne*: $\forall n.\ \exists x.\ x \in (f\ n)$
proof
fix *n*
from *closed* **have** $\exists a\ b.\ f\ n =$ *closed-int a b* $\wedge\ a \leq b$ **by** *simp*
then obtain *a* **and** *b* **where** *fn*: $f\ n =$ *closed-int a b* $\wedge\ a \leq b$ **by** *auto*
hence $a \leq b$ **..**
with *closed-not-empty* **have** $\exists x.\ x \in$*closed-int a b* **by** *simp*
with *fn* **show** $\exists x.\ x \in (f\ n)$ **by** *simp*
qed

have *gdef*: $\forall n.\ (?g\ n) \in (f\ n) \wedge (\forall x \in (f\ n).\ (?g\ n) \leq x)$
proof

fix n
from *closed* **have** $\exists\, a\ b.\ f\ n = \textit{closed-int}\ a\ b \wedge a \leq b$ **..**
then obtain a **and** b **where** *ff*: $f\ n = \textit{closed-int}\ a\ b$ **and** $a \leq b$ **by** *auto*
hence $a \leq b$ **by** *simp*
hence $a \in \textit{closed-int}\ a\ b \wedge (\forall\, x \in \textit{closed-int}\ a\ b.\ a \leq x)$ **by** (*rule closed-int-least*)
with *ff* **have** $a \in (f\ n) \wedge (\forall\, x \in (f\ n).\ a \leq x)$ **by** *simp*
hence $\exists\, c.\ c \in (f\ n) \wedge (\forall\, x \in (f\ n).\ c \leq x)$ **..**
thus $(?g\ n) \in (f\ n) \wedge (\forall\, x \in (f\ n).\ (?g\ n) \leq x)$ **by** (*rule someI-ex*)
qed

— A denotes the set of all left-most points of all the intervals ...
moreover obtain A **where** *Adef*: $A = ?g\ `\ \mathbb{N}$ **by** *simp*
ultimately have $\exists\, x.\ x \in A$
proof −
have $(0::nat) \in \mathbb{N}$ **by** *simp*
moreover have $?g\ 0 = ?g\ 0$ **by** *simp*
ultimately have $?g\ 0 \in ?g\ `\ \mathbb{N}$ **by** (*rule rev-image-eqI*)
with *Adef* **have** $?g\ 0 \in A$ **by** *simp*
thus *?thesis* **..**
qed

— Now show that A is bounded above ...
moreover have $\exists\, y.\ \textit{isUb}\ (\textit{UNIV}::real\ set)\ A\ y$
proof −
{
fix n
from *ne* **have** *ex*: $\exists\, x.\ x \in (f\ n)$ **..**
from *gdef* **have** $(?g\ n) \in (f\ n) \wedge (\forall\, x \in (f\ n).\ (?g\ n) \leq x)$ **by** *simp*
moreover
from *closed* **have** $\exists\, a\ b.\ f\ n = \textit{closed-int}\ a\ b \wedge a \leq b$ **..**
then obtain a **and** b **where** $f\ n = \textit{closed-int}\ a\ b \wedge a \leq b$ **by** *auto*
hence $b \in (f\ n) \wedge (\forall\, x \in (f\ n).\ x \leq b)$ **using** *closed-int-most* **by** *blast*
ultimately have $\forall\, x \in (f\ n).\ (?g\ n) \leq b$ **by** *simp*
with *ex* **have** $(?g\ n) \leq b$ **by** *auto*
hence $\exists\, b.\ (?g\ n) \leq b$ **by** *auto*
}
hence *aux*: $\forall\, n.\ \exists\, b.\ (?g\ n) \leq b$ **..**

have *fs*: $\forall\, n::nat.\ f\ n \subseteq f\ 0$
proof (*rule allI*, *induct-tac* n)
show $f\ 0 \subseteq f\ 0$ **by** *simp*
next
fix n
assume $f\ n \subseteq f\ 0$
moreover from *subset* **have** $f\ (Suc\ n) \subseteq f\ n$ **..**
ultimately show $f\ (Suc\ n) \subseteq f\ 0$ **by** *simp*
qed
have $\forall\, n.\ (?g\ n) \in (f\ 0)$
proof

```
    fix n
    from gdef have (?g n)∈(f n) ∧ (∀x∈(f n). (?g n)≤x) by simp
    hence ?g n ∈ f n ..
    with fs show ?g n ∈ f 0 by auto
  qed
  moreover from closed
    obtain a and b where f 0 = closed-int a b and alb: a ≤ b by blast
  ultimately have ∀n. ?g n ∈ closed-int a b by auto
  with alb have ∀n. ?g n ≤ b using closed-int-most by blast
  with Adef have ∀y∈A. y≤b by auto
  hence A *<= b by (unfold setle-def)
  moreover have b ∈ (UNIV::real set) by simp
  ultimately have A *<= b ∧ b ∈ (UNIV::real set) by simp
  hence isUb (UNIV::real set) A b by (unfold isUb-def)
  thus ?thesis by auto
qed
— by the Axiom Of Completeness, A has a least upper bound ...
ultimately have ∃t. isLub UNIV A t by (rule reals-complete)

— denote this least upper bound as t ...
then obtain t where tdef: isLub UNIV A t ..

— and finally show that this least upper bound is in all the intervals...
have ∀n. t ∈ f n
proof
  fix n::nat
  from closed obtain a and b where
    int: f n = closed-int a b and alb: a ≤ b by blast

  have t ≥ a
  proof −
    have a ∈ A
    proof −

      from alb int have ain: a∈f n ∧ (∀x∈f n. a ≤ x)
        using closed-int-least by blast
      moreover have ∀e. e∈f n ∧ (∀x∈f n. e ≤ x) ⟶ e = a
      proof clarsimp
        fix e
        assume ein: e ∈ f n and lt: ∀x∈f n. e ≤ x
        from lt ain have aux: ∀x∈f n. a ≤ x ∧ e ≤ x by auto

        from ein aux have a ≤ e ∧ e ≤ e by auto
        moreover from ain aux have a ≤ a ∧ e ≤ a by auto
        ultimately show e = a by simp
      qed
      hence ⋀e. e∈f n ∧ (∀x∈f n. e ≤ x) ⟹ e = a by simp
      ultimately have (?g n) = a by (rule some-equality)
      moreover
```

```
    {
     have n = of-nat n by simp
     moreover have of-nat n ∈ ℕ by simp
     ultimately have n ∈ ℕ
      apply −
      apply (subst(asm) eq-sym-conv)
      apply (erule subst)
      .
    }
    with Adef have (?g n) ∈ A by auto
    ultimately show ?thesis by simp
  qed
  with tdef show a ≤ t by (rule isLubD2)
qed
moreover have t ≤ b
proof −
  have isUb UNIV A b
  proof −
    {
     from alb int have
       ain: b∈f n ∧ (∀x∈f n. x ≤ b) using closed-int-most by blast

     have subsetd: ∀m. ∀n. f (n + m) ⊆ f n
     proof (rule allI, induct-tac m)
       show ∀n. f (n + 0) ⊆ f n by simp
     next
       fix m n
       assume pp: ∀p. f (p + n) ⊆ f p
       {
         fix p
         from pp have f (p + n) ⊆ f p by simp
         moreover from subset have f (Suc (p + n)) ⊆ f (p + n) by auto
         hence f (p + (Suc n)) ⊆ f (p + n) by simp
         ultimately have f (p + (Suc n)) ⊆ f p by simp
       }
       thus ∀p. f (p + Suc n) ⊆ f p ..
     qed
     have subsetm: ∀α β. α ≥ β ⟶ (f α) ⊆ (f β)
     proof ((rule allI)+, rule impI)
       fix α::nat and β::nat
       assume β ≤ α
       hence ∃k. α = β + k by (simp only: le-iff-add)
       then obtain k where α = β + k ..
       moreover
       from subsetd have f (β + k) ⊆ f β by simp
       ultimately show f α ⊆ f β by auto
     qed

     fix m
```

```
        {
          assume m ≥ n
          with subsetm have f m ⊆ f n by simp
          with ain have ∀ x∈f m. x ≤ b by auto
          moreover
          from gdef have ?g m ∈ f m ∧ (∀ x∈f m. ?g m ≤ x) by simp
          ultimately have ?g m ≤ b by auto
        }
        moreover
        {
          assume ¬(m ≥ n)
          hence m < n by simp
          with subsetm have sub: (f n) ⊆ (f m) by simp
          from closed obtain ma and mb where
            f m = closed-int ma mb ∧ ma ≤ mb by blast
          hence one: ma ≤ mb and fm: f m = closed-int ma mb by auto
          from one alb sub fm int have ma ≤ b using closed-subset by blast
          moreover have (?g m) = ma
          proof −
            from gdef have ?g m ∈ f m ∧ (∀ x∈f m. ?g m ≤ x) ..
            moreover from one have
              ma ∈ closed-int ma mb ∧ (∀ x∈closed-int ma mb. ma ≤ x)
              by (rule closed-int-least)
            with fm have ma∈f m ∧ (∀ x∈f m. ma ≤ x) by simp
            ultimately have ma ≤ ?g m ∧ ?g m ≤ ma by auto
            thus ?g m = ma by auto
          qed
          ultimately have ?g m ≤ b by simp
        }
        ultimately have ?g m ≤ b by (rule case-split)
      }
      with Adef have ∀ y∈A. y≤b by auto
      hence A *<= b by (unfold setle-def)
      moreover have b ∈ (UNIV::real set) by simp
      ultimately have A *<= b ∧ b ∈ (UNIV::real set) by simp
      thus isUb (UNIV::real set) A b by (unfold isUb-def)
    qed
    with tdef show t ≤ b by (rule isLub-le-isUb)
  qed
  ultimately have t ∈ closed-int a b by (rule closed-mem)
  with int show t ∈ f n by simp
 qed
 hence t ∈ (⋂ n. f n) by auto
 thus ?thesis by auto
qed
```

16.4 Generating the intervals

16.4.1 Existence of non-singleton closed intervals

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

lemma *closed-subset-ex*:
fixes *c*::*real*
assumes *alb*: $a < b$
shows
$\exists\, ka\ kb.\ ka < kb \wedge closed\text{-}int\ ka\ kb \subseteq closed\text{-}int\ a\ b \wedge c \notin (closed\text{-}int\ ka\ kb)$
proof −
{
assume *clb*: $c < b$
{
assume *cla*: $c < a$
from *alb cla clb* **have** $c \notin closed\text{-}int\ a\ b$ **by** (*unfold closed-int-def*, *auto*)
with *alb* **have**
$a < b \wedge closed\text{-}int\ a\ b \subseteq closed\text{-}int\ a\ b \wedge c \notin closed\text{-}int\ a\ b$
by *auto*
hence
$\exists\, ka\ kb.\ ka < kb \wedge closed\text{-}int\ ka\ kb \subseteq closed\text{-}int\ a\ b \wedge c \notin (closed\text{-}int\ ka\ kb)$
by *auto*
}
moreover
{
assume *ncla*: $\neg(c < a)$
with *clb* **have** *cdef*: $a \leq c \wedge c < b$ **by** *simp*
obtain *ka* **where** *kadef*: $ka = (c + b)/2$ **by** *blast*

from *kadef clb* **have** *kalb*: $ka < b$ **by** *auto*
moreover from *kadef cdef* **have** *kagc*: $ka > c$ **by** *simp*
ultimately have $c \notin (closed\text{-}int\ ka\ b)$ **by** (*unfold closed-int-def*, *auto*)
moreover from *cdef kagc* **have** $ka \geq a$ **by** *simp*
hence $closed\text{-}int\ ka\ b \subseteq closed\text{-}int\ a\ b$ **by** (*unfold closed-int-def*, *auto*)
ultimately have
$ka < b \ \wedge closed\text{-}int\ ka\ b \subseteq closed\text{-}int\ a\ b \wedge c \notin closed\text{-}int\ ka\ b$
using *kalb* **by** *auto*
hence
$\exists\, ka\ kb.\ ka < kb \wedge closed\text{-}int\ ka\ kb \subseteq closed\text{-}int\ a\ b \wedge c \notin (closed\text{-}int\ ka\ kb)$
by *auto*

}
ultimately have
$\exists\, ka\ kb.\ ka < kb \wedge closed\text{-}int\ ka\ kb \subseteq closed\text{-}int\ a\ b \wedge c \notin (closed\text{-}int\ ka\ kb)$
by (*rule case-split*)
}
moreover

```
 {
  assume ¬ (c < b)
  hence cgeb: c ≥ b by simp

  obtain kb where kbdef: kb = (a + b)/2 by blast
  with alb have kblb: kb < b by auto
  with kbdef cgeb have a < kb ∧ kb < c by auto
  moreover hence c ∉ (closed-int a kb) by (unfold closed-int-def, auto)
  moreover from kblb have
    closed-int a kb ⊆ closed-int a b by (unfold closed-int-def, auto)
  ultimately have
    a < kb ∧  closed-int a kb ⊆ closed-int a b ∧ c∉closed-int a kb
    by simp
  hence
    ∃ka kb. ka < kb ∧ closed-int ka kb ⊆ closed-int a b ∧ c ∉ (closed-int ka kb)
    by auto
 }
 ultimately show ?thesis by (rule case-split)
qed
```

16.5 newInt: Interval generation

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

16.5.1 Definition

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

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

16.5.2 Properties

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

```
lemma newInt-ex:
  ∃ a b. a < b ∧
    newInt (Suc n) f = closed-int a b ∧
    newInt (Suc n) f ⊆ newInt n f ∧
```

```
   f (Suc n) ∉ newInt (Suc n) f
proof (induct n)
  case 0

  let ?e = SOME e. ∃ e1 e2.
   e1 < e2 ∧
   e = closed-int e1 e2 ∧
   e ⊆ closed-int (f 0 + 1) (f 0 + 2) ∧
   f (Suc 0) ∉ e

  have newInt (Suc 0) f = ?e by auto
  moreover
  have f 0 + 1 < f 0 + 2 by simp
  with closed-subset-ex have
    ∃ ka kb. ka < kb ∧ closed-int ka kb ⊆ closed-int (f 0 + 1) (f 0 + 2) ∧
     f (Suc 0) ∉ (closed-int ka kb) .
  hence
    ∃ e. ∃ ka kb. ka < kb ∧ e = closed-int ka kb ∧
     e ⊆ closed-int (f 0 + 1) (f 0 + 2) ∧ f (Suc 0) ∉ e by simp
  hence
    ∃ ka kb. ka < kb ∧ ?e = closed-int ka kb ∧
     ?e ⊆ closed-int (f 0 + 1) (f 0 + 2) ∧ f (Suc 0) ∉ ?e
    by (rule someI-ex)
  ultimately have ∃ e1 e2. e1 < e2 ∧
   newInt (Suc 0) f = closed-int e1 e2 ∧
   newInt (Suc 0) f ⊆ closed-int (f 0 + 1) (f 0 + 2) ∧
   f (Suc 0) ∉ newInt (Suc 0) f by simp
  thus
    ∃ a b. a < b ∧ newInt (Suc 0) f = closed-int a b ∧
     newInt (Suc 0) f ⊆ newInt 0 f ∧ f (Suc 0) ∉ newInt (Suc 0) f
    by simp
next
  case (Suc n)
  hence ∃ a b.
   a < b ∧
   newInt (Suc n) f = closed-int a b ∧
   newInt (Suc n) f ⊆ newInt n f ∧
   f (Suc n) ∉ newInt (Suc n) f by simp
  then obtain a and b where ab: a < b ∧
   newInt (Suc n) f = closed-int a b ∧
   newInt (Suc n) f ⊆ newInt n f ∧
   f (Suc n) ∉ newInt (Suc n) f by auto
  hence cab: closed-int a b = newInt (Suc n) f by simp

  let ?e = SOME e. ∃ e1 e2.
    e1 < e2 ∧
    e = closed-int e1 e2 ∧
    e ⊆ closed-int a b ∧
    f (Suc (Suc n)) ∉ e
```

```
  from cab have ni: newInt (Suc (Suc n)) f = ?e by auto

  from ab have a < b by simp
  with closed-subset-ex have
    ∃ ka kb. ka < kb ∧ closed-int ka kb ⊆ closed-int a b ∧
     f (Suc (Suc n)) ∉ closed-int ka kb .
  hence
    ∃ e. ∃ ka kb. ka < kb ∧ e = closed-int ka kb ∧
     closed-int ka kb ⊆ closed-int a b ∧ f (Suc (Suc n)) ∉ closed-int ka kb
    by simp
  hence
    ∃ e.  ∃ ka kb. ka < kb ∧ e = closed-int ka kb ∧
     e ⊆ closed-int a b ∧ f (Suc (Suc n)) ∉ e by simp
  hence
    ∃ ka kb. ka < kb ∧ ?e = closed-int ka kb ∧
     ?e ⊆ closed-int a b ∧ f (Suc (Suc n)) ∉ ?e by (rule someI-ex)
  with ab ni show
    ∃ ka kb. ka < kb ∧
     newInt (Suc (Suc n)) f = closed-int ka kb ∧
     newInt (Suc (Suc n)) f ⊆ newInt (Suc n) f ∧
     f (Suc (Suc n)) ∉ newInt (Suc (Suc n)) f by auto
qed

lemma newInt-subset:
  newInt (Suc n) f ⊆ newInt n f
  using newInt-ex by auto
```

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

```
lemma newInt-inter:
  ∀ n. f n ∉ (⋂ n. newInt n f)
proof
  fix n::nat
  {
    assume n0: n = 0
    moreover have newInt 0 f = closed-int (f 0 + 1) (f 0 + 2) by simp
    ultimately have f n ∉ newInt n f by (unfold closed-int-def, simp)
  }
  moreover
  {
    assume ¬ n = 0
    hence n > 0 by simp
    then obtain m where ndef: n = Suc m by (auto simp add: gr0-conv-Suc)

    from newInt-ex have
      ∃ a b. a < b ∧ (newInt (Suc m) f) = closed-int a b ∧
       newInt (Suc m) f ⊆ newInt m f ∧ f (Suc m) ∉ newInt (Suc m) f .
    then have f (Suc m) ∉ newInt (Suc m) f by auto
    with ndef have f n ∉ newInt n f by simp
```

```
  }
  ultimately have f n ∉ newInt n f by (rule case-split)
  thus f n ∉ (⋂ n. newInt n f) by auto
qed


lemma newInt-notempty:
  (⋂ n. newInt n f) ≠ {}
proof −
  let ?g = λn. newInt n f
  have ∀ n. ?g (Suc n) ⊆ ?g n
  proof
    fix n
    show ?g (Suc n) ⊆ ?g n by (rule newInt-subset)
  qed
  moreover have ∀ n. ∃ a b. ?g n = closed-int a b ∧ a ≤ b
  proof
    fix n::nat
    {
      assume n = 0
      then have
        ?g n = closed-int (f 0 + 1) (f 0 + 2) ∧ (f 0 + 1 ≤ f 0 + 2)
        by simp
      hence ∃ a b. ?g n = closed-int a b ∧ a ≤ b by blast
    }
    moreover
    {
      assume ¬ n = 0
      then have n > 0 by simp
      then obtain m where nd: n = Suc m by (auto simp add: gr0-conv-Suc)

      have
        ∃ a b. a < b ∧ (newInt (Suc m) f) = closed-int a b ∧
        (newInt (Suc m) f) ⊆ (newInt m f) ∧ (f (Suc m)) ∉ (newInt (Suc m) f)
        by (rule newInt-ex)
      then obtain a and b where
        a < b ∧ (newInt (Suc m) f) = closed-int a b by auto
      with nd have ?g n = closed-int a b ∧ a ≤ b by auto
      hence ∃ a b. ?g n = closed-int a b ∧ a ≤ b by blast
    }
    ultimately show ∃ a b. ?g n = closed-int a b ∧ a ≤ b by (rule case-split)
  qed
  ultimately show ?thesis by (rule NIP)
qed
```

16.6 Final Theorem

```
theorem real-non-denum:
  shows ¬ (∃f::nat⇒real. surj f)
```

proof — by contradiction
assume $\exists f{::}nat \Rightarrow real.\ surj\ f$
then obtain $f{::}nat \Rightarrow real$ **where** *surj f* **by** *auto*
hence *rangeF*: *range f* = *UNIV* **by** (*rule surj-range*)
— We now produce a real number x that is not in the range of f, using the properties of newInt.
have $\exists x.\ x \in (\bigcap n.\ newInt\ n\ f)$ **using** *newInt-notempty* **by** *blast*
moreover have $\forall n.\ f\ n \notin (\bigcap n.\ newInt\ n\ f)$ **by** (*rule newInt-inter*)
ultimately obtain *x* **where** $x \in (\bigcap n.\ newInt\ n\ f)$ **and** $\forall n.\ f\ n \neq x$ **by** *blast*
moreover from *rangeF* **have** $x \in range\ f$ **by** *simp*
ultimately show *False* **by** *blast*
qed

end

17 FrechetDeriv: Frechet Derivative

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

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

lemma *FDERIV-I*:
⟦*bounded-linear D*; (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *D h*) / *norm h*) −− *0* −−> *0*⟧
⟹ *FDERIV f x :> D*
by (*simp add*: *fderiv-def*)

lemma *FDERIV-D*:
FDERIV f x :> D ⟹ (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *D h*) / *norm h*) −− *0* −−> *0*
by (*simp add*: *fderiv-def*)

lemma *FDERIV-bounded-linear*: *FDERIV f x :> D* ⟹ *bounded-linear D*
by (*simp add*: *fderiv-def*)

lemma *bounded-linear-zero*:
bounded-linear (λ*x*::*'a*::*real-normed-vector*. *0*::*'b*::*real-normed-vector*)
proof
show (*0*::*'b*) = *0* + *0* **by** *simp*
fix *r* **show** (*0*::*'b*) = *scaleR r 0* **by** *simp*

have $\forall x::'a.\ norm\ (0::'b) \leq norm\ x * 0$ **by** *simp*
thus $\exists K.\ \forall x::'a.\ norm\ (0::'b) \leq norm\ x * K$ **..**
qed

lemma *FDERIV-const*: $FDERIV\ (\lambda x.\ k)\ x :> (\lambda h.\ 0)$
by (*simp add*: *fderiv-def bounded-linear-zero*)

lemma *bounded-linear-ident*:
$bounded\text{-}linear\ (\lambda x::'a::real\text{-}normed\text{-}vector.\ x)$
proof
fix $x\ y :: 'a$ **show** $x + y = x + y$ **by** *simp*
fix r **and** $x :: 'a$ **show** $scaleR\ r\ x = scaleR\ r\ x$ **by** *simp*
have $\forall x::'a.\ norm\ x \leq norm\ x * 1$ **by** *simp*
thus $\exists K.\ \forall x::'a.\ norm\ x \leq norm\ x * K$ **..**
qed

lemma *FDERIV-ident*: $FDERIV\ (\lambda x.\ x)\ x :> (\lambda h.\ h)$
by (*simp add*: *fderiv-def bounded-linear-ident*)

17.1 Addition

lemma *bounded-linear-add*:
assumes *bounded-linear f*
assumes *bounded-linear g*
shows $bounded\text{-}linear\ (\lambda x.\ f\ x + g\ x)$
proof −
interpret *f*: *bounded-linear f* **by** *fact*
interpret *g*: *bounded-linear g* **by** *fact*
show *?thesis* **apply** (*unfold-locales*)
apply (*simp only*: *f.add g.add add-ac*)
apply (*simp only*: *f.scaleR g.scaleR scaleR-right-distrib*)
apply (*rule f.pos-bounded* [*THEN exE*], *rename-tac Kf*)
apply (*rule g.pos-bounded* [*THEN exE*], *rename-tac Kg*)
apply (*rule-tac x*=*Kf* + *Kg* **in** *exI*, *safe*)
apply (*subst right-distrib*)
apply (*rule order-trans* [*OF norm-triangle-ineq*])
apply (*rule add-mono*, *erule spec*, *erule spec*)
done
qed

lemma *norm-ratio-ineq*:
fixes $x\ y :: 'a::real\text{-}normed\text{-}vector$
fixes $h :: 'b::real\text{-}normed\text{-}vector$
shows $norm\ (x + y)\ /\ norm\ h \leq norm\ x\ /\ norm\ h + norm\ y\ /\ norm\ h$
apply (*rule ord-le-eq-trans*)
apply (*rule divide-right-mono*)
apply (*rule norm-triangle-ineq*)
apply (*rule norm-ge-zero*)
apply (*rule add-divide-distrib*)

done

lemma *FDERIV-add*:
assumes *f*: *FDERIV f x :> F*
assumes *g*: *FDERIV g x :> G*
shows *FDERIV* (λ*x*. *f x* + *g x*) *x* :> (λ*h*. *F h* + *G h*)
proof (*rule FDERIV-I*)
show *bounded-linear* (λ*h*. *F h* + *G h*)
apply (*rule bounded-linear-add*)
apply (*rule FDERIV-bounded-linear* [*OF f*])
apply (*rule FDERIV-bounded-linear* [*OF g*])
done
next
have *f'*: (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *F h*) / *norm h*) −− *0* −−> *0*
using *f* **by** (*rule FDERIV-D*)
have *g'*: (λ*h*. *norm* (*g* (*x* + *h*) − *g x* − *G h*) / *norm h*) −− *0* −−> *0*
using *g* **by** (*rule FDERIV-D*)
from *f' g'*
have (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *F h*) / *norm h*
+ *norm* (*g* (*x* + *h*) − *g x* − *G h*) / *norm h*) −− *0* −−> *0*
by (*rule LIM-add-zero*)
thus (λ*h*. *norm* (*f* (*x* + *h*) + *g* (*x* + *h*) − (*f x* + *g x*) − (*F h* + *G h*))
/ *norm h*) −− *0* −−> *0*
apply (*rule real-LIM-sandwich-zero*)
apply (*simp add*: *divide-nonneg-pos*)
apply (*simp only*: *add-diff-add*)
apply (*rule norm-ratio-ineq*)
done
qed

17.2 Subtraction

lemma *bounded-linear-minus*:
assumes *bounded-linear f*
shows *bounded-linear* (λ*x*. − *f x*)
proof −
interpret *f*: *bounded-linear f* **by** *fact*
show *?thesis* **apply** (*unfold-locales*)
apply (*simp add*: *f.add*)
apply (*simp add*: *f.scaleR*)
apply (*simp add*: *f.bounded*)
done
qed

lemma *FDERIV-minus*:
FDERIV f x :> F ⟹ *FDERIV* (λ*x*. − *f x*) *x* :> (λ*h*. − *F h*)
apply (*rule FDERIV-I*)
apply (*rule bounded-linear-minus*)
apply (*erule FDERIV-bounded-linear*)

apply (*simp only*: *fderiv-def minus-diff-minus norm-minus-cancel*)
done

lemma *FDERIV-diff*:
⟦*FDERIV f x :> F*; *FDERIV g x :> G*⟧
⟹ *FDERIV* (λ*x*. *f x* − *g x*) *x :>* (λ*h*. *F h* − *G h*)
by (*simp only*: *diff-minus FDERIV-add FDERIV-minus*)

17.3 Continuity

lemma *FDERIV-isCont*:
assumes *f*: *FDERIV f x :> F*
shows *isCont f x*
proof −
from *f* **interpret** *F*: *bounded-linear F* **by** (*rule FDERIV-bounded-linear*)
have (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *F h*) / *norm h*) −− *0* −−> *0*
by (*rule FDERIV-D* [*OF f*])
hence (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *F h*) / *norm h* ∗ *norm h*) −− *0* −−> *0*
by (*intro LIM-mult-zero LIM-norm-zero LIM-ident*)
hence (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *F h*)) −− *0* −−> *0*
by (*simp cong*: *LIM-cong*)
hence (λ*h*. *f* (*x* + *h*) − *f x* − *F h*) −− *0* −−> *0*
by (*rule LIM-norm-zero-cancel*)
hence (λ*h*. *f* (*x* + *h*) − *f x* − *F h* + *F h*) −− *0* −−> *0*
by (*intro LIM-add-zero F.LIM-zero LIM-ident*)
hence (λ*h*. *f* (*x* + *h*) − *f x*) −− *0* −−> *0*
by *simp*
thus *isCont f x*
unfolding *isCont-iff* **by** (*rule LIM-zero-cancel*)
qed

17.4 Composition

lemma *real-divide-cancel-lemma*:
fixes *a b c* :: *real*
shows (*b* = *0* ⟹ *a* = *0*) ⟹ (*a* / *b*) ∗ (*b* / *c*) = *a* / *c*
by *simp*

lemma *bounded-linear-compose*:
assumes *bounded-linear f*
assumes *bounded-linear g*
shows *bounded-linear* (λ*x*. *f* (*g x*))
proof −
interpret *f*: *bounded-linear f* **by** *fact*
interpret *g*: *bounded-linear g* **by** *fact*
show *?thesis* **proof** (*unfold-locales*)
fix *x y* **show** *f* (*g* (*x* + *y*)) = *f* (*g x*) + *f* (*g y*)
by (*simp only*: *f.add g.add*)
next
fix *r x* **show** *f* (*g* (*scaleR r x*)) = *scaleR r* (*f* (*g x*))

```
    by (simp only: f.scaleR g.scaleR)
  next
    from f.pos-bounded
    obtain Kf where f: ⋀x. norm (f x) ≤ norm x * Kf and Kf: 0 < Kf by fast
    from g.pos-bounded
    obtain Kg where g: ⋀x. norm (g x) ≤ norm x * Kg by fast
    show ∃ K. ∀ x. norm (f (g x)) ≤ norm x * K
    proof (intro exI allI)
      fix x
      have norm (f (g x)) ≤ norm (g x) * Kf
        using f .
      also have ... ≤ (norm x * Kg) * Kf
        using g Kf [THEN order-less-imp-le] by (rule mult-right-mono)
      also have (norm x * Kg) * Kf = norm x * (Kg * Kf)
        by (rule mult-assoc)
      finally show norm (f (g x)) ≤ norm x * (Kg * Kf) .
    qed
  qed
qed

lemma FDERIV-compose:
  fixes f :: 'a::real-normed-vector ⇒ 'b::real-normed-vector
  fixes g :: 'b::real-normed-vector ⇒ 'c::real-normed-vector
  assumes f: FDERIV f x :> F
  assumes g: FDERIV g (f x) :> G
  shows FDERIV (λx. g (f x)) x :> (λh. G (F h))
proof (rule FDERIV-I)
  from FDERIV-bounded-linear [OF g] FDERIV-bounded-linear [OF f]
  show bounded-linear (λh. G (F h))
    by (rule bounded-linear-compose)
next
  let ?Rf = λh. f (x + h) - f x - F h
  let ?Rg = λk. g (f x + k) - g (f x) - G k
  let ?k = λh. f (x + h) - f x
  let ?Nf = λh. norm (?Rf h) / norm h
  let ?Ng = λh. norm (?Rg (?k h)) / norm (?k h)
  from f interpret F: bounded-linear F by (rule FDERIV-bounded-linear)
  from g interpret G: bounded-linear G by (rule FDERIV-bounded-linear)
  from F.bounded obtain kF where kF: ⋀x. norm (F x) ≤ norm x * kF by fast
  from G.bounded obtain kG where kG: ⋀x. norm (G x) ≤ norm x * kG by
fast

  let ?fun2 = λh. ?Nf h * kG + ?Ng h * (?Nf h + kF)

  show (λh. norm (g (f (x + h)) - g (f x) - G (F h)) / norm h) -- 0 --> 0
  proof (rule real-LIM-sandwich-zero)
    have Nf: ?Nf -- 0 --> 0
      using FDERIV-D [OF f] .
```

```
  have Ng1: isCont (λk. norm (?Rg k) / norm k) 0
    by (simp add: isCont-def FDERIV-D [OF g])
  have Ng2: ?k -- 0 --> 0
    apply (rule LIM-zero)
    apply (fold isCont-iff)
    apply (rule FDERIV-isCont [OF f])
    done
  have Ng: ?Ng -- 0 --> 0
    using isCont-LIM-compose [OF Ng1 Ng2] by simp

  have (λh. ?Nf h * kG + ?Ng h * (?Nf h + kF))
        -- 0 --> 0 * kG + 0 * (0 + kF)
    by (intro LIM-add LIM-mult LIM-const Nf Ng)
  thus (λh. ?Nf h * kG + ?Ng h * (?Nf h + kF)) -- 0 --> 0
    by simp
next
  fix h::'a assume h: h ≠ 0
  thus 0 ≤ norm (g (f (x + h)) - g (f x) - G (F h)) / norm h
    by (simp add: divide-nonneg-pos)
next
  fix h::'a assume h: h ≠ 0
  have g (f (x + h)) - g (f x) - G (F h) = G (?Rf h) + ?Rg (?k h)
    by (simp add: G.diff)
  hence norm (g (f (x + h)) - g (f x) - G (F h)) / norm h
        = norm (G (?Rf h) + ?Rg (?k h)) / norm h
    by (rule arg-cong)
  also have ... ≤ norm (G (?Rf h)) / norm h + norm (?Rg (?k h)) / norm h
    by (rule norm-ratio-ineq)
  also have ... ≤ ?Nf h * kG + ?Ng h * (?Nf h + kF)
  proof (rule add-mono)
    show norm (G (?Rf h)) / norm h ≤ ?Nf h * kG
      apply (rule ord-le-eq-trans)
      apply (rule divide-right-mono [OF kG norm-ge-zero])
      apply simp
      done
  next
    have norm (?Rg (?k h)) / norm h = ?Ng h * (norm (?k h) / norm h)
      apply (rule real-divide-cancel-lemma [symmetric])
      apply (simp add: G.zero)
      done
    also have ... ≤ ?Ng h * (?Nf h + kF)
    proof (rule mult-left-mono)
      have norm (?k h) / norm h = norm (?Rf h + F h) / norm h
        by simp
      also have ... ≤ ?Nf h + norm (F h) / norm h
        by (rule norm-ratio-ineq)
      also have ... ≤ ?Nf h + kF
        apply (rule add-left-mono)
        apply (subst pos-divide-le-eq, simp add: h)
```

```
        apply (subst mult-commute)
        apply (rule kF)
        done
      finally show norm (?k h) / norm h ≤ ?Nf h + kF .
    next
      show 0 ≤ ?Ng h
      apply (case-tac f (x + h) − f x = 0, simp)
      apply (rule divide-nonneg-pos [OF norm-ge-zero])
      apply simp
      done
    qed
    finally show norm (?Rg (?k h)) / norm h ≤ ?Ng h * (?Nf h + kF) .
  qed
  finally show norm (g (f (x + h)) − g (f x) − G (F h)) / norm h
      ≤ ?Nf h * kG + ?Ng h * (?Nf h + kF) .
 qed
qed
```

17.5 Product Rule

```
lemma (in bounded-bilinear) FDERIV-lemma:
  a' ** b' − a ** b − (a ** B + A ** b)
   = a ** (b' − b − B) + (a' − a − A) ** b' + A ** (b' − b)
by (simp add: diff-left diff-right)

lemma (in bounded-bilinear) FDERIV:
  fixes x :: 'd::real-normed-vector
  assumes f: FDERIV f x :> F
  assumes g: FDERIV g x :> G
  shows FDERIV (λx. f x ** g x) x :> (λh. f x ** G h + F h ** g x)
proof (rule FDERIV-I)
  show bounded-linear (λh. f x ** G h + F h ** g x)
    apply (rule bounded-linear-add)
    apply (rule bounded-linear-compose [OF bounded-linear-right])
    apply (rule FDERIV-bounded-linear [OF g])
    apply (rule bounded-linear-compose [OF bounded-linear-left])
    apply (rule FDERIV-bounded-linear [OF f])
    done
next
  from bounded-linear.bounded [OF FDERIV-bounded-linear [OF f]]
  obtain KF where norm-F: ⋀x. norm (F x) ≤ norm x * KF by fast

  from pos-bounded obtain K where K: 0 < K and norm-prod:
    ⋀a b. norm (a ** b) ≤ norm a * norm b * K by fast

  let ?Rf = λh. f (x + h) − f x − F h
  let ?Rg = λh. g (x + h) − g x − G h

  let ?fun1 = λh.
```

```
      norm (f x ** ?Rg h + ?Rf h ** g (x + h) + F h ** (g (x + h) - g x)) /
      norm h

  let ?fun2 = λh.
      norm (f x) * (norm (?Rg h) / norm h) * K +
      norm (?Rf h) / norm h * norm (g (x + h)) * K +
      KF * norm (g (x + h) - g x) * K

  have ?fun1 -- 0 --> 0
  proof (rule real-LIM-sandwich-zero)
    from f g isCont-iff [THEN iffD1, OF FDERIV-isCont [OF g]]
    have ?fun2 -- 0 -->
        norm (f x) * 0 * K + 0 * norm (g x) * K + KF * norm (0::'b) * K
      by (intro LIM-add LIM-mult LIM-const LIM-norm LIM-zero FDERIV-D)
    thus ?fun2 -- 0 --> 0
      by simp
  next
    fix h::'d assume h ≠ 0
    thus 0 ≤ ?fun1 h
      by (simp add: divide-nonneg-pos)
  next
    fix h::'d assume h ≠ 0
    have ?fun1 h ≤ (norm (f x) * norm (?Rg h) * K +
        norm (?Rf h) * norm (g (x + h)) * K +
        norm h * KF * norm (g (x + h) - g x) * K) / norm h
      by (intro
        divide-right-mono mult-mono'
        order-trans [OF norm-triangle-ineq add-mono]
        order-trans [OF norm-prod mult-right-mono]
        mult-nonneg-nonneg order-refl norm-ge-zero norm-F
        K [THEN order-less-imp-le]
      )
    also have ... = ?fun2 h
      by (simp add: add-divide-distrib)
    finally show ?fun1 h ≤ ?fun2 h .
  qed
  thus (λh.
    norm (f (x + h) ** g (x + h) - f x ** g x - (f x ** G h + F h ** g x))
    / norm h) -- 0 --> 0
    by (simp only: FDERIV-lemma)
qed

lemmas FDERIV-mult = mult.FDERIV

lemmas FDERIV-scaleR = scaleR.FDERIV
```

17.6 Powers

```
lemma FDERIV-power-Suc:
```

fixes x :: $'a$::{*real-normed-algebra*,*comm-ring-1*}
shows *FDERIV* $(\lambda x.\ x\ \hat{}\ Suc\ n)\ x :> (\lambda h.\ (1 + of\text{-}nat\ n) * x\ \hat{}\ n * h)$
apply (*induct n*)
apply (*simp add*: *FDERIV-ident*)
apply (*drule FDERIV-mult* [*OF FDERIV-ident*])
apply (*simp only*: *of-nat-Suc left-distrib mult-1-left*)
apply (*simp only*: *power-Suc right-distrib add-ac mult-ac*)
done

lemma *FDERIV-power*:
fixes x :: $'a$::{*real-normed-algebra*,*comm-ring-1*}
shows *FDERIV* $(\lambda x.\ x\ \hat{}\ n)\ x :> (\lambda h.\ of\text{-}nat\ n * x\ \hat{}\ (n - 1) * h)$
apply (*cases n*)
apply (*simp add*: *FDERIV-const*)
apply (*simp add*: *FDERIV-power-Suc del*: *power-Suc*)
done

17.7 Inverse

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

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

lemma *FDERIV-inverse*:
fixes x :: $'a$::*real-normed-div-algebra*
assumes *x*: $x \neq 0$
shows *FDERIV inverse x* :> $(\lambda h.\ -\ (inverse\ x * h * inverse\ x))$
(**is** *FDERIV ?inv - :> -*)
proof (*rule FDERIV-I*)
show *bounded-linear* $(\lambda h.\ -\ (?inv\ x * h * ?inv\ x))$
apply (*rule bounded-linear-minus*)
apply (*rule bounded-linear-mult-const*)
apply (*rule bounded-linear-const-mult*)
apply (*rule bounded-linear-ident*)
done
next
show $(\lambda h.\ norm\ (?inv\ (x + h) - ?inv\ x - -\ (?inv\ x * h * ?inv\ x))\ /\ norm\ h)$
$-- 0 --> 0$
proof (*rule LIM-equal2*)
show $0 < norm\ x$ **using** *x* **by** *simp*
next
fix h::$'a$
assume *1*: $h \neq 0$
assume $norm\ (h - 0) < norm\ x$
hence $h \neq -x$ **by** *clarsimp*
hence *2*: $x + h \neq 0$
apply (*rule contrapos-nn*)

```
    apply (rule sym)
    apply (erule minus-unique)
    done
  show norm (?inv (x + h) - ?inv x - - (?inv x * h * ?inv x)) / norm h
      = norm ((?inv (x + h) - ?inv x) * h * ?inv x) / norm h
    apply (subst inverse-diff-inverse [OF 2 x])
    apply (subst minus-diff-minus)
    apply (subst norm-minus-cancel)
    apply (simp add: left-diff-distrib)
    done
 next
  show (λh. norm ((?inv (x + h) - ?inv x) * h * ?inv x) / norm h)
      -- 0 --> 0
  proof (rule real-LIM-sandwich-zero)
    show (λh. norm (?inv (x + h) - ?inv x) * norm (?inv x))
        -- 0 --> 0
      apply (rule LIM-mult-left-zero)
      apply (rule LIM-norm-zero)
      apply (rule LIM-zero)
      apply (rule LIM-offset-zero)
      apply (rule LIM-inverse)
      apply (rule LIM-ident)
      apply (rule x)
      done
  next
    fix h::'a assume h: h ≠ 0
    show 0 ≤ norm ((?inv (x + h) - ?inv x) * h * ?inv x) / norm h
      apply (rule divide-nonneg-pos)
      apply (rule norm-ge-zero)
      apply (simp add: h)
      done
  next
    fix h::'a assume h: h ≠ 0
    have norm ((?inv (x + h) - ?inv x) * h * ?inv x) / norm h
        ≤ norm (?inv (x + h) - ?inv x) * norm h * norm (?inv x) / norm h
      apply (rule divide-right-mono [OF - norm-ge-zero])
      apply (rule order-trans [OF norm-mult-ineq])
      apply (rule mult-right-mono [OF - norm-ge-zero])
      apply (rule norm-mult-ineq)
      done
    also have ... = norm (?inv (x + h) - ?inv x) * norm (?inv x)
      by simp
    finally show norm ((?inv (x + h) - ?inv x) * h * ?inv x) / norm h
        ≤ norm (?inv (x + h) - ?inv x) * norm (?inv x) .
  qed
 qed
qed
```

17.8 Alternate definition

lemma *field-fderiv-def*:
fixes $x :: 'a$::*real-normed-field* **shows**
*FDERIV f x :> (λh. h * D) = (λh. (f (x + h) − f x) / h) −− 0 −−> D*
apply (*unfold fderiv-def*)
apply (*simp add*: *mult.bounded-linear-left*)
apply (*simp cong*: *LIM-cong add*: *nonzero-norm-divide* [*symmetric*])
apply (*subst diff-divide-distrib*)
apply (*subst times-divide-eq-left* [*symmetric*])
apply (*simp cong*: *LIM-cong*)
apply (*simp add*: *LIM-norm-zero-iff LIM-zero-iff*)
done

end

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

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

18.1 Real inner product spaces

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

setup ⟪ *Sign.add-const-constraint*
(@{*const-name open*}, *SOME* @{*typ 'a*::*open set ⇒ bool*}) ⟫

setup ⟪ *Sign.add-const-constraint*
(@{*const-name dist*}, *SOME* @{*typ 'a*::*dist ⇒ 'a ⇒ real*}) ⟫

setup ⟪ *Sign.add-const-constraint*
(@{*const-name norm*}, *SOME* @{*typ 'a*::*norm ⇒ real*}) ⟫

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

lemma *inner-zero-left* [*simp*]: *inner 0 x = 0*
using *inner-add-left* [*of 0 0 x*] **by** *simp*

lemma *inner-minus-left* [*simp*]: *inner* (− *x*) *y* = − *inner x y*
using *inner-add-left* [*of x* − *x y*] **by** *simp*

lemma *inner-diff-left*: *inner* (*x* − *y*) *z* = *inner x z* − *inner y z*
by (*simp add*: *diff-minus inner-add-left*)

Transfer distributivity rules to right argument.

lemma *inner-add-right*: *inner x* (*y* + *z*) = *inner x y* + *inner x z*
using *inner-add-left* [*of y z x*] **by** (*simp only*: *inner-commute*)

lemma *inner-scaleR-right* [*simp*]: *inner x* (*scaleR r y*) = *r* ∗ (*inner x y*)
using *inner-scaleR-left* [*of r y x*] **by** (*simp only*: *inner-commute*)

lemma *inner-zero-right* [*simp*]: *inner x* 0 = 0
using *inner-zero-left* [*of x*] **by** (*simp only*: *inner-commute*)

lemma *inner-minus-right* [*simp*]: *inner x* (− *y*) = − *inner x y*
using *inner-minus-left* [*of y x*] **by** (*simp only*: *inner-commute*)

lemma *inner-diff-right*: *inner x* (*y* − *z*) = *inner x y* − *inner x z*
using *inner-diff-left* [*of y z x*] **by** (*simp only*: *inner-commute*)

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

Legacy theorem names

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

lemma *inner-gt-zero-iff* [*simp*]: 0 < *inner x x* ⟷ *x* ≠ 0
by (*simp add*: *order-less-le*)

lemma *power2-norm-eq-inner*: (*norm x*)^2 = *inner x x*
by (*simp add*: *norm-eq-sqrt-inner*)

lemma *Cauchy-Schwarz-ineq*:
(*inner x y*)^2 ≤ *inner x x* ∗ *inner y y*
proof (*cases*)
assume *y* = 0
thus *?thesis* **by** *simp*
next
assume *y*: *y* ≠ 0
let *?r* = *inner x y* / *inner y y*
have 0 ≤ *inner* (*x* − *scaleR ?r y*) (*x* − *scaleR ?r y*)
by (*rule inner-ge-zero*)
also have ... = *inner x x* − *inner y x* ∗ *?r*

by (*simp add*: *inner-diff*)
also have $\ldots = inner\ x\ x - (inner\ x\ y)^2\ /\ inner\ y\ y$
by (*simp add*: *power2-eq-square inner-commute*)
finally have $0 \leq inner\ x\ x - (inner\ x\ y)^2\ /\ inner\ y\ y$ **.**
hence $(inner\ x\ y)^2\ /\ inner\ y\ y \leq inner\ x\ x$
by (*simp add*: *le-diff-eq*)
thus $(inner\ x\ y)^2 \leq inner\ x\ x * inner\ y\ y$
by (*simp add*: *pos-divide-le-eq y*)
qed

lemma *Cauchy-Schwarz-ineq2*:
$|inner\ x\ y| \leq norm\ x * norm\ y$
proof (*rule power2-le-imp-le*)
have $(inner\ x\ y)^2 \leq inner\ x\ x * inner\ y\ y$
using *Cauchy-Schwarz-ineq* **.**
thus $|inner\ x\ y|^2 \leq (norm\ x * norm\ y)^2$
by (*simp add*: *power-mult-distrib power2-norm-eq-inner*)
show $0 \leq norm\ x * norm\ y$
unfolding *norm-eq-sqrt-inner*
by (*intro mult-nonneg-nonneg real-sqrt-ge-zero inner-ge-zero*)
qed

subclass *real-normed-vector*
proof
fix $a :: real$ **and** $x\ y :: 'a$
show $0 \leq norm\ x$
unfolding *norm-eq-sqrt-inner* **by** *simp*
show $norm\ x = 0 \longleftrightarrow x = 0$
unfolding *norm-eq-sqrt-inner* **by** *simp*
show $norm\ (x + y) \leq norm\ x + norm\ y$
proof (*rule power2-le-imp-le*)
have $inner\ x\ y \leq norm\ x * norm\ y$
by (*rule order-trans* [*OF abs-ge-self Cauchy-Schwarz-ineq2*])
thus $(norm\ (x + y))^2 \leq (norm\ x + norm\ y)^2$
unfolding *power2-sum power2-norm-eq-inner*
by (*simp add*: *inner-add inner-commute*)
show $0 \leq norm\ x + norm\ y$
unfolding *norm-eq-sqrt-inner*
by (*simp add*: *add-nonneg-nonneg*)
qed
have $sqrt\ (a^2 * inner\ x\ x) = |a| * sqrt\ (inner\ x\ x)$
by (*simp add*: *real-sqrt-mult-distrib*)
then show $norm\ (a *_R x) = |a| * norm\ x$
unfolding *norm-eq-sqrt-inner*
by (*simp add*: *power2-eq-square mult-assoc*)
qed

end

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

setup ⟪ *Sign.add-const-constraint*
(@{*const-name open*}, *SOME* @{*typ 'a::topological-space set ⇒ bool*}) ⟫

setup ⟪ *Sign.add-const-constraint*
(@{*const-name dist*}, *SOME* @{*typ 'a::metric-space ⇒ 'a ⇒ real*}) ⟫

setup ⟪ *Sign.add-const-constraint*
(@{*const-name norm*}, *SOME* @{*typ 'a::real-normed-vector ⇒ real*}) ⟫

interpretation *inner*:
bounded-bilinear inner::'a::real-inner ⇒ 'a ⇒ real
proof
fix *x y z* :: *'a* **and** *r* :: *real*
show *inner* (*x* + *y*) *z* = *inner x z* + *inner y z*
by (*rule inner-add-left*)
show *inner x* (*y* + *z*) = *inner x y* + *inner x z*
by (*rule inner-add-right*)
show *inner* (*scaleR r x*) *y* = *scaleR r* (*inner x y*)
unfolding *real-scaleR-def* **by** (*rule inner-scaleR-left*)
show *inner x* (*scaleR r y*) = *scaleR r* (*inner x y*)
unfolding *real-scaleR-def* **by** (*rule inner-scaleR-right*)
show ∃ *K*. ∀ *x y*::*'a*. *norm* (*inner x y*) ≤ *norm x* ∗ *norm y* ∗ *K*
proof
show ∀ *x y*::*'a*. *norm* (*inner x y*) ≤ *norm x* ∗ *norm y* ∗ *1*
by (*simp add*: *Cauchy-Schwarz-ineq2*)
qed
qed

interpretation *inner-left*:
bounded-linear λ*x*::*'a*::*real-inner*. *inner x y*
by (*rule inner.bounded-linear-left*)

interpretation *inner-right*:
bounded-linear λ*y*::*'a*::*real-inner*. *inner x y*
by (*rule inner.bounded-linear-right*)

18.2 Class instances

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

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

instance proof
fix *x y z r* :: *real*
show *inner x y* = *inner y x*
unfolding *inner-real-def* **by** (*rule mult-commute*)
show *inner* (*x* + *y*) *z* = *inner x z* + *inner y z*
unfolding *inner-real-def* **by** (*rule left-distrib*)

show *inner* (*scaleR r x*) *y* = *r* ∗ *inner x y*
unfolding *inner-real-def real-scaleR-def* **by** (*rule mult-assoc*)
show *0* ≤ *inner x x*
unfolding *inner-real-def* **by** *simp*
show *inner x x* = *0* ⟷ *x* = *0*
unfolding *inner-real-def* **by** *simp*
show *norm x* = *sqrt* (*inner x x*)
unfolding *inner-real-def* **by** *simp*
qed

end

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

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

instance proof
fix *x y z* :: *complex* **and** *r* :: *real*
show *inner x y* = *inner y x*
unfolding *inner-complex-def* **by** (*simp add*: *mult-commute*)
show *inner* (*x* + *y*) *z* = *inner x z* + *inner y z*
unfolding *inner-complex-def* **by** (*simp add*: *left-distrib*)
show *inner* (*scaleR r x*) *y* = *r* ∗ *inner x y*
unfolding *inner-complex-def* **by** (*simp add*: *right-distrib*)
show *0* ≤ *inner x x*
unfolding *inner-complex-def* **by** (*simp add*: *add-nonneg-nonneg*)
show *inner x x* = *0* ⟷ *x* = *0*
unfolding *inner-complex-def*
by (*simp add*: *add-nonneg-eq-0-iff complex-Re-Im-cancel-iff*)
show *norm x* = *sqrt* (*inner x x*)
unfolding *inner-complex-def complex-norm-def*
by (*simp add*: *power2-eq-square*)
qed

end

18.3 Gradient derivative

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

lemma *deriv-fderiv*: *DERIV f x* :> *D* ⟷ *FDERIV f x* :> (λ*h*. *h* ∗ *D*)
by (*simp only*: *deriv-def field-fderiv-def*)

lemma *gderiv-deriv* [*simp*]: *GDERIV f x :> D ⟷ DERIV f x :> D*
by (*simp only*: *gderiv-def deriv-fderiv inner-real-def*)

lemma *GDERIV-DERIV-compose*:
⟦*GDERIV f x :> df*; *DERIV g (f x) :> dg*⟧
⟹ *GDERIV* (λ*x*. *g* (*f x*)) *x :> scaleR dg df*
unfolding *gderiv-def deriv-fderiv*
apply (*drule* (*1*) *FDERIV-compose*)
apply (*simp add*: *mult-ac*)
done

lemma *FDERIV-subst*: ⟦*FDERIV f x :> df*; *df* = *d*⟧ ⟹ *FDERIV f x :> d*
by *simp*

lemma *GDERIV-subst*: ⟦*GDERIV f x :> df*; *df* = *d*⟧ ⟹ *GDERIV f x :> d*
by *simp*

lemma *GDERIV-const*: *GDERIV* (λ*x*. *k*) *x :> 0*
unfolding *gderiv-def inner-right.zero* **by** (*rule FDERIV-const*)

lemma *GDERIV-add*:
⟦*GDERIV f x :> df*; *GDERIV g x :> dg*⟧
⟹ *GDERIV* (λ*x*. *f x* + *g x*) *x :> df* + *dg*
unfolding *gderiv-def inner-right.add* **by** (*rule FDERIV-add*)

lemma *GDERIV-minus*:
GDERIV f x :> df ⟹ *GDERIV* (λ*x*. − *f x*) *x :> − df*
unfolding *gderiv-def inner-right.minus* **by** (*rule FDERIV-minus*)

lemma *GDERIV-diff*:
⟦*GDERIV f x :> df*; *GDERIV g x :> dg*⟧
⟹ *GDERIV* (λ*x*. *f x* − *g x*) *x :> df* − *dg*
unfolding *gderiv-def inner-right.diff* **by** (*rule FDERIV-diff*)

lemma *GDERIV-scaleR*:
⟦*DERIV f x :> df*; *GDERIV g x :> dg*⟧
⟹ *GDERIV* (λ*x*. *scaleR* (*f x*) (*g x*)) *x*
:> (*scaleR* (*f x*) *dg* + *scaleR df* (*g x*))
unfolding *gderiv-def deriv-fderiv inner-right.add inner-right.scaleR*
apply (*rule FDERIV-subst*)
apply (*erule* (*1*) *scaleR.FDERIV*)
apply (*simp add*: *mult-ac*)
done

lemma *GDERIV-mult*:
⟦*GDERIV f x :> df*; *GDERIV g x :> dg*⟧
⟹ *GDERIV* (λ*x*. *f x* ∗ *g x*) *x :> scaleR* (*f x*) *dg* + *scaleR* (*g x*) *df*
unfolding *gderiv-def*

apply (*rule FDERIV-subst*)
apply (*erule (1) FDERIV-mult*)
apply (*simp add*: *inner-add mult-ac*)
done

lemma *GDERIV-inverse*:
⟦*GDERIV f x :> df*; *f x ≠ 0*⟧
⟹ *GDERIV* (λ*x*. *inverse* (*f x*)) *x :> − (inverse* (*f x*))^2 $*_R$ *df*
apply (*erule GDERIV-DERIV-compose*)
apply (*erule DERIV-inverse* [*folded numeral-2-eq-2*])
done

lemma *GDERIV-norm*:
assumes *x ≠ 0* **shows** *GDERIV* (λ*x*. *norm x*) *x :> sgn x*
proof −
have *1*: *FDERIV* (λ*x*. *inner x x*) *x :>* (λ*h*. *inner x h* + *inner h x*)
by (*intro inner.FDERIV FDERIV-ident*)
have *2*: (λ*h*. *inner x h* + *inner h x*) = (λ*h*. *inner h* (*scaleR 2 x*))
by (*simp add*: *expand-fun-eq inner-commute*)
have *0 < inner x x* **using** ‹*x ≠ 0*› **by** *simp*
then have *3*: *DERIV sqrt* (*inner x x*) :> (*inverse* (*sqrt* (*inner x x*)) / *2*)
by (*rule DERIV-real-sqrt*)
have *4*: (*inverse* (*sqrt* (*inner x x*)) / *2*) $*_R$ *2* $*_R$ *x* = *sgn x*
by (*simp add*: *sgn-div-norm norm-eq-sqrt-inner*)
show *?thesis*
unfolding *norm-eq-sqrt-inner*
apply (*rule GDERIV-subst* [*OF - 4*])
apply (*rule GDERIV-DERIV-compose* [**where** *g*=*sqrt* **and** *df*=*scaleR 2 x*])
apply (*subst gderiv-def*)
apply (*rule FDERIV-subst* [*OF - 2*])
apply (*rule 1*)
apply (*rule 3*)
done
qed

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

end

19 Product-plus: Additive group operations on product types

theory *Product-plus*
imports *Main*
begin

19.1 Operations

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

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

instance ..
end

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

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

instance ..
end

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

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

instance ..
end

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

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

instance ..
end

lemma *fst-zero* [*simp*]: $fst\ 0 = 0$
unfolding *zero-prod-def* **by** *simp*

lemma *snd-zero* [*simp*]: $snd\ 0 = 0$
unfolding *zero-prod-def* **by** *simp*

lemma *fst-add* [*simp*]: $fst\ (x + y) = fst\ x + fst\ y$
unfolding *plus-prod-def* **by** *simp*

lemma *snd-add* [*simp*]: $snd\ (x + y) = snd\ x + snd\ y$
unfolding *plus-prod-def* **by** *simp*

lemma *fst-diff* [*simp*]: $fst\ (x - y) = fst\ x - fst\ y$

unfolding *minus-prod-def* **by** *simp*

lemma *snd-diff* [*simp*]: *snd* $(x - y) =$ *snd* $x -$ *snd* y
unfolding *minus-prod-def* **by** *simp*

lemma *fst-uminus* [*simp*]: *fst* $(- x) = -$ *fst* x
unfolding *uminus-prod-def* **by** *simp*

lemma *snd-uminus* [*simp*]: *snd* $(- x) = -$ *snd* x
unfolding *uminus-prod-def* **by** *simp*

lemma *add-Pair* [*simp*]: $(a, b) + (c, d) = (a + c, b + d)$
unfolding *plus-prod-def* **by** *simp*

lemma *diff-Pair* [*simp*]: $(a, b) - (c, d) = (a - c, b - d)$
unfolding *minus-prod-def* **by** *simp*

lemma *uminus-Pair* [*simp*, *code*]: $- (a, b) = (- a, - b)$
unfolding *uminus-prod-def* **by** *simp*

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

19.2 Class instances

instance $*$ :: (*semigroup-add*, *semigroup-add*) *semigroup-add*
by *default* (*simp add*: *expand-prod-eq add-assoc*)

instance $*$ :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
by *default* (*simp add*: *expand-prod-eq add-commute*)

instance $*$ :: (*monoid-add*, *monoid-add*) *monoid-add*
by *default* (*simp-all add*: *expand-prod-eq*)

instance $*$ :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
by *default* (*simp add*: *expand-prod-eq*)

instance $*$ ::
(*cancel-semigroup-add*, *cancel-semigroup-add*) *cancel-semigroup-add*
by *default* (*simp-all add*: *expand-prod-eq*)

instance $*$ ::
(*cancel-ab-semigroup-add*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
by *default* (*simp add*: *expand-prod-eq*)

instance $*$ ::
(*cancel-comm-monoid-add*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
..

instance $*$ :: (*group-add*, *group-add*) *group-add*

by *default* (*simp-all add*: *expand-prod-eq diff-minus*)

instance $*$ :: (*ab-group-add*, *ab-group-add*) *ab-group-add*
by *default* (*simp-all add*: *expand-prod-eq*)

lemma *fst-setsum*: *fst* $(\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ fst\ (f\ x))$
by (*cases finite A*, *induct set*: *finite*, *simp-all*)

lemma *snd-setsum*: *snd* $(\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ snd\ (f\ x))$
by (*cases finite A*, *induct set*: *finite*, *simp-all*)

end

20 Product-Vector: Cartesian Products as Vector Spaces

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

20.1 Product is a real vector space

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

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

lemma *fst-scaleR* [*simp*]: *fst* (*scaleR r A*) = *scaleR r* (*fst A*)
unfolding *scaleR-prod-def* **by** *simp*

lemma *snd-scaleR* [*simp*]: *snd* (*scaleR r A*) = *scaleR r* (*snd A*)
unfolding *scaleR-prod-def* **by** *simp*

lemma *scaleR-Pair* [*simp*]: *scaleR r* (*a*, *b*) = (*scaleR r a*, *scaleR r b*)
unfolding *scaleR-prod-def* **by** *simp*

instance proof
fix *a b* :: *real* **and** *x y* :: $'a \times 'b$
show *scaleR a* (*x* + *y*) = *scaleR a x* + *scaleR a y*
by (*simp add*: *expand-prod-eq scaleR-right-distrib*)
show *scaleR* (*a* + *b*) *x* = *scaleR a x* + *scaleR b x*
by (*simp add*: *expand-prod-eq scaleR-left-distrib*)
show *scaleR a* (*scaleR b x*) = *scaleR* (*a* $*$ *b*) *x*
by (*simp add*: *expand-prod-eq*)
show *scaleR 1 x* = *x*
by (*simp add*: *expand-prod-eq*)

qed

end

20.2 Product is a topological space

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

definition *open-prod-def*:
open (S :: ($'a \times 'b$) *set*) $\longleftrightarrow$
($\forall x{\in}S.\ \exists A\ B.$ *open* $A \wedge$ *open* $B \wedge x \in A \times B \wedge A \times B \subseteq S$)

lemma *open-prod-elim*:
assumes *open* S **and** $x \in S$
obtains $A\ B$ **where** *open* A **and** *open* B **and** $x \in A \times B$ **and** $A \times B \subseteq S$
using *assms* **unfolding** *open-prod-def* **by** *fast*

lemma *open-prod-intro*:
assumes $\bigwedge x.\ x \in S \Longrightarrow \exists A\ B.$ *open* $A \wedge$ *open* $B \wedge x \in A \times B \wedge A \times B \subseteq S$
shows *open* S
using *assms* **unfolding** *open-prod-def* **by** *fast*

instance proof
show *open* (*UNIV* :: ($'a \times 'b$) *set*)
unfolding *open-prod-def* **by** *auto*
next
fix $S\ T$ :: ($'a \times 'b$) *set*
assume *open* S *open* T
show *open* ($S \cap T$)
proof (*rule open-prod-intro*)
fix x **assume** x: $x \in S \cap T$
from x **have** $x \in S$ **by** *simp*
obtain *Sa Sb* **where** A: *open Sa open Sb* $x \in Sa \times Sb$ $Sa \times Sb \subseteq S$
using ‹*open* S› **and** ‹$x \in S$› **by** (*rule open-prod-elim*)
from x **have** $x \in T$ **by** *simp*
obtain *Ta Tb* **where** B: *open Ta open Tb* $x \in Ta \times Tb$ $Ta \times Tb \subseteq T$
using ‹*open* T› **and** ‹$x \in T$› **by** (*rule open-prod-elim*)
let *?A* $= Sa \cap Ta$ **and** *?B* $= Sb \cap Tb$
have *open ?A* $\wedge$ *open ?B* $\wedge\ x \in$ *?A* $\times$ *?B* $\wedge$ *?A* $\times$ *?B* $\subseteq S \cap T$
using $A\ B$ **by** (*auto simp add*: *open-Int*)
thus $\exists A\ B.$ *open* $A \wedge$ *open* $B \wedge x \in A \times B \wedge A \times B \subseteq S \cap T$
by *fast*
qed
next
fix K :: ($'a \times 'b$) *set set*
assume $\forall S{\in}K.$ *open* S **thus** *open* ($\bigcup K$)
unfolding *open-prod-def* **by** *fast*

qed

end

lemma *open-Times*: *open S* $\Longrightarrow$ *open T* $\Longrightarrow$ *open* (*S* $\times$ *T*)
unfolding *open-prod-def* **by** *auto*

lemma *fst-vimage-eq-Times*: *fst* $-`$ *S* = *S* $\times$ *UNIV*
by *auto*

lemma *snd-vimage-eq-Times*: *snd* $-`$ *S* = *UNIV* $\times$ *S*
by *auto*

lemma *open-vimage-fst*: *open S* $\Longrightarrow$ *open* (*fst* $-`$ *S*)
by (*simp add*: *fst-vimage-eq-Times open-Times*)

lemma *open-vimage-snd*: *open S* $\Longrightarrow$ *open* (*snd* $-`$ *S*)
by (*simp add*: *snd-vimage-eq-Times open-Times*)

lemma *closed-vimage-fst*: *closed S* $\Longrightarrow$ *closed* (*fst* $-`$ *S*)
unfolding *closed-open vimage-Compl* [*symmetric*]
by (*rule open-vimage-fst*)

lemma *closed-vimage-snd*: *closed S* $\Longrightarrow$ *closed* (*snd* $-`$ *S*)
unfolding *closed-open vimage-Compl* [*symmetric*]
by (*rule open-vimage-snd*)

lemma *closed-Times*: *closed S* $\Longrightarrow$ *closed T* $\Longrightarrow$ *closed* (*S* $\times$ *T*)
proof −
  have *S* $\times$ *T* = (*fst* $-`$ *S*) $\cap$ (*snd* $-`$ *T*) **by** *auto*
  thus *closed S* $\Longrightarrow$ *closed T* $\Longrightarrow$ *closed* (*S* $\times$ *T*)
    by (*simp add*: *closed-vimage-fst closed-vimage-snd closed-Int*)
qed

lemma *openI*:
  assumes $\bigwedge$*x*. *x* $\in$ *S* $\Longrightarrow$ $\exists$ *T*. *open T* $\wedge$ *x* $\in$ *T* $\wedge$ *T* $\subseteq$ *S*
  shows *open S*
proof −
  have *open* ($\bigcup$ {*T*. *open T* $\wedge$ *T* $\subseteq$ *S*}) **by** *auto*
  moreover have $\bigcup$ {*T*. *open T* $\wedge$ *T* $\subseteq$ *S*} = *S* **by** (*auto dest*!: *assms*)
  ultimately show *open S* **by** *simp*
qed

lemma *subset-fst-imageI*: *A* $\times$ *B* $\subseteq$ *S* $\Longrightarrow$ *y* $\in$ *B* $\Longrightarrow$ *A* $\subseteq$ *fst* ` *S*
  unfolding *image-def subset-eq* **by** *force*

lemma *subset-snd-imageI*: *A* $\times$ *B* $\subseteq$ *S* $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *B* $\subseteq$ *snd* ` *S*
  unfolding *image-def subset-eq* **by** *force*

lemma *open-image-fst*: **assumes** *open S* **shows** *open* (*fst ' S*)
proof (*rule openI*)
  fix *x* **assume** *x* ∈ *fst ' S*
  then obtain *y* **where** (*x*, *y*) ∈ *S* **by** *auto*
  then obtain *A B* **where** *open A open B x* ∈ *A y* ∈ *B A* × *B* ⊆ *S*
    using ‹*open S*› **unfolding** *open-prod-def* **by** *auto*
  from ‹*A* × *B* ⊆ *S*› ‹*y* ∈ *B*› **have** *A* ⊆ *fst ' S* **by** (*rule subset-fst-imageI*)
  with ‹*open A*› ‹*x* ∈ *A*› **have** *open A* ∧ *x* ∈ *A* ∧ *A* ⊆ *fst ' S* **by** *simp*
  then show ∃ *T*. *open T* ∧ *x* ∈ *T* ∧ *T* ⊆ *fst ' S* **by** − (*rule exI*)
qed

lemma *open-image-snd*: **assumes** *open S* **shows** *open* (*snd ' S*)
proof (*rule openI*)
  fix *y* **assume** *y* ∈ *snd ' S*
  then obtain *x* **where** (*x*, *y*) ∈ *S* **by** *auto*
  then obtain *A B* **where** *open A open B x* ∈ *A y* ∈ *B A* × *B* ⊆ *S*
    using ‹*open S*› **unfolding** *open-prod-def* **by** *auto*
  from ‹*A* × *B* ⊆ *S*› ‹*x* ∈ *A*› **have** *B* ⊆ *snd ' S* **by** (*rule subset-snd-imageI*)
  with ‹*open B*› ‹*y* ∈ *B*› **have** *open B* ∧ *y* ∈ *B* ∧ *B* ⊆ *snd ' S* **by** *simp*
  then show ∃ *T*. *open T* ∧ *y* ∈ *T* ∧ *T* ⊆ *snd ' S* **by** − (*rule exI*)
qed

20.3 Product is a metric space

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

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

lemma *dist-Pair-Pair*: *dist* (*a*, *b*) (*c*, *d*) = *sqrt* ((*dist a c*)^2 + (*dist b d*)^2)
  unfolding *dist-prod-def* **by** *simp*

lemma *dist-fst-le*: *dist* (*fst x*) (*fst y*) ≤ *dist x y*
unfolding *dist-prod-def* **by** (*rule real-sqrt-sum-squares-ge1*)

lemma *dist-snd-le*: *dist* (*snd x*) (*snd y*) ≤ *dist x y*
unfolding *dist-prod-def* **by** (*rule real-sqrt-sum-squares-ge2*)

instance proof
  fix *x y* :: *'a* × *'b*
  show *dist x y* = *0* ⟷ *x* = *y*
    unfolding *dist-prod-def expand-prod-eq* **by** *simp*
next
  fix *x y z* :: *'a* × *'b*
  show *dist x y* ≤ *dist x z* + *dist y z*
    unfolding *dist-prod-def*
    by (*intro order-trans* [*OF - real-sqrt-sum-squares-triangle-ineq*]

real-sqrt-le-mono add-mono power-mono dist-triangle2 zero-le-dist)
next

fix $S :: ('a \times 'b)$ *set*
show *open* $S \longleftrightarrow (\forall x{\in}S.\ \exists e{>}0.\ \forall y.\ dist\ y\ x < e \longrightarrow y \in S)$
proof
assume *open* S **show** $\forall x{\in}S.\ \exists e{>}0.\ \forall y.\ dist\ y\ x < e \longrightarrow y \in S$
proof
fix x **assume** $x \in S$
obtain $A\ B$ **where** *open* A *open* B $x \in A \times B$ $A \times B \subseteq S$
using ‹*open* S› **and** ‹$x \in S$› **by** (*rule open-prod-elim*)
obtain r **where** $r{:}\ 0 < r$ $\forall y.\ dist\ y\ (fst\ x) < r \longrightarrow y \in A$
using ‹*open* A› **and** ‹$x \in A \times B$› **unfolding** *open-dist* **by** *auto*
obtain s **where** $s{:}\ 0 < s$ $\forall y.\ dist\ y\ (snd\ x) < s \longrightarrow y \in B$
using ‹*open* B› **and** ‹$x \in A \times B$› **unfolding** *open-dist* **by** *auto*
let *?e* $= min\ r\ s$
have $0 <$ *?e* $\wedge$ $(\forall y.\ dist\ y\ x <$ *?e* $\longrightarrow y \in S)$
proof (*intro allI impI conjI*)
show $0 < min\ r\ s$ **by** (*simp add*: $r(1)\ s(1)$)
next
fix y **assume** $dist\ y\ x < min\ r\ s$
hence $dist\ y\ x < r$ **and** $dist\ y\ x < s$
by *simp-all*
hence $dist\ (fst\ y)\ (fst\ x) < r$ **and** $dist\ (snd\ y)\ (snd\ x) < s$
by (*auto intro*: *le-less-trans dist-fst-le dist-snd-le*)
hence $fst\ y \in A$ **and** $snd\ y \in B$
by (*simp-all add*: $r(2)\ s(2)$)
hence $y \in A \times B$ **by** (*induct y*, *simp*)
with ‹$A \times B \subseteq S$› **show** $y \in S$ **..**
qed
thus $\exists e{>}0.\ \forall y.\ dist\ y\ x < e \longrightarrow y \in S$ **..**
qed
next
assume $\forall x{\in}S.\ \exists e{>}0.\ \forall y.\ dist\ y\ x < e \longrightarrow y \in S$ **thus** *open* S
unfolding *open-prod-def open-dist*
apply *safe*
apply (*drule* (*1*) *bspec*)
apply *clarify*
apply (*subgoal-tac* $\exists r{>}0.\ \exists s{>}0.\ e = sqrt\ (r^2 + s^2)$)
apply *clarify*
apply (*rule-tac* $x{=}\{y.\ dist\ y\ a < r\}$ **in** *exI*)
apply (*rule-tac* $x{=}\{y.\ dist\ y\ b < s\}$ **in** *exI*)
apply (*rule conjI*)
apply *clarify*
apply (*rule-tac* $x{=}r - dist\ x\ a$ **in** *exI*, *rule conjI*, *simp*)
apply *clarify*
apply (*simp add*: *less-diff-eq*)

```
    apply (erule le-less-trans [OF dist-triangle])
    apply (rule conjI)
    apply clarify
    apply (rule-tac x=s − dist x b in exI, rule conjI, simp)
    apply clarify
    apply (simp add: less-diff-eq)
    apply (erule le-less-trans [OF dist-triangle])
    apply (rule conjI)
    apply simp
    apply (clarify, rename-tac c d)
    apply (drule spec, erule mp)
    apply (simp add: dist-Pair-Pair add-strict-mono power-strict-mono)
    apply (rule-tac x=e / sqrt 2 in exI, simp add: divide-pos-pos)
    apply (rule-tac x=e / sqrt 2 in exI, simp add: divide-pos-pos)
    apply (simp add: power-divide)
    done
  qed
qed

end
```

20.4 Continuity of operations

```
lemma tendsto-fst [tendsto-intros]:
  assumes (f ---> a) net
  shows ((λx. fst (f x)) ---> fst a) net
proof (rule topological-tendstoI)
  fix S assume open S fst a ∈ S
  then have open (fst -` S) a ∈ fst -` S
    unfolding open-prod-def
    apply simp-all
    apply clarify
    apply (rule exI, erule conjI)
    apply (rule exI, rule conjI [OF open-UNIV])
    apply auto
    done
  with assms have eventually (λx. f x ∈ fst -` S) net
    by (rule topological-tendstoD)
  then show eventually (λx. fst (f x) ∈ S) net
    by simp
qed

lemma tendsto-snd [tendsto-intros]:
  assumes (f ---> a) net
  shows ((λx. snd (f x)) ---> snd a) net
proof (rule topological-tendstoI)
  fix S assume open S snd a ∈ S
  then have open (snd -` S) a ∈ snd -` S
    unfolding open-prod-def
```

```
    apply simp-all
    apply clarify
    apply (rule exI, rule conjI [OF open-UNIV])
    apply (rule exI, erule conjI)
    apply auto
    done
  with assms have eventually (λx. f x ∈ snd −‘ S) net
    by (rule topological-tendstoD)
  then show eventually (λx. snd (f x) ∈ S) net
    by simp
qed

lemma tendsto-Pair [tendsto-intros]:
  assumes (f ———> a) net and (g ———> b) net
  shows ((λx. (f x, g x)) ———> (a, b)) net
proof (rule topological-tendstoI)
  fix S assume open S (a, b) ∈ S
  then obtain A B where open A open B a ∈ A b ∈ B A × B ⊆ S
    unfolding open-prod-def by auto
  have eventually (λx. f x ∈ A) net
    using ‹(f ———> a) net› ‹open A› ‹a ∈ A›
    by (rule topological-tendstoD)
  moreover
  have eventually (λx. g x ∈ B) net
    using ‹(g ———> b) net› ‹open B› ‹b ∈ B›
    by (rule topological-tendstoD)
  ultimately
  show eventually (λx. (f x, g x) ∈ S) net
    by (rule eventually-elim2)
       (simp add: subsetD [OF ‹A × B ⊆ S›])
qed

lemma Cauchy-fst: Cauchy X ⟹ Cauchy (λn. fst (X n))
unfolding Cauchy-def by (fast elim: le-less-trans [OF dist-fst-le])

lemma Cauchy-snd: Cauchy X ⟹ Cauchy (λn. snd (X n))
unfolding Cauchy-def by (fast elim: le-less-trans [OF dist-snd-le])

lemma Cauchy-Pair:
  assumes Cauchy X and Cauchy Y
  shows Cauchy (λn. (X n, Y n))
proof (rule metric-CauchyI)
  fix r :: real assume 0 < r
  then have 0 < r / sqrt 2 (is 0 < ?s)
    by (simp add: divide-pos-pos)
  obtain M where M: ∀ m≥M. ∀ n≥M. dist (X m) (X n) < ?s
    using metric-CauchyD [OF ‹Cauchy X› ‹0 < ?s›] ..
  obtain N where N: ∀ m≥N. ∀ n≥N. dist (Y m) (Y n) < ?s
    using metric-CauchyD [OF ‹Cauchy Y› ‹0 < ?s›] ..
```

have $\forall\, m{\geq}max\ M\ N.\ \forall\, n{\geq}max\ M\ N.\ dist\ (X\ m,\ Y\ m)\ (X\ n,\ Y\ n) < r$
using *M N* **by** (*simp add*: *real-sqrt-sum-squares-less dist-Pair-Pair*)
then show $\exists\, n0.\ \forall\, m{\geq}n0.\ \forall\, n{\geq}n0.\ dist\ (X\ m,\ Y\ m)\ (X\ n,\ Y\ n) < r$ **..**
qed

lemma *isCont-Pair* [*simp*]:
$[\![isCont\ f\ x;\ isCont\ g\ x]\!] \Longrightarrow isCont\ (\lambda x.\ (f\ x,\ g\ x))\ x$
unfolding *isCont-def* **by** (*rule tendsto-Pair*)

20.5 Product is a complete metric space

instance $*$ :: (*complete-space*, *complete-space*) *complete-space*
proof
fix *X* :: $nat \Rightarrow 'a \times 'b$ **assume** *Cauchy X*
have *1*: $(\lambda n.\ fst\ (X\ n)) \longrightarrow lim\ (\lambda n.\ fst\ (X\ n))$
using *Cauchy-fst* [*OF* ‹*Cauchy X*›]
by (*simp add*: *Cauchy-convergent-iff convergent-LIMSEQ-iff*)
have *2*: $(\lambda n.\ snd\ (X\ n)) \longrightarrow lim\ (\lambda n.\ snd\ (X\ n))$
using *Cauchy-snd* [*OF* ‹*Cauchy X*›]
by (*simp add*: *Cauchy-convergent-iff convergent-LIMSEQ-iff*)
have $X \longrightarrow (lim\ (\lambda n.\ fst\ (X\ n)),\ lim\ (\lambda n.\ snd\ (X\ n)))$
using *tendsto-Pair* [*OF 1 2*] **by** *simp*
then show *convergent X*
by (*rule convergentI*)
qed

20.6 Product is a normed vector space

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

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

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

lemma *norm-Pair*: $norm\ (a,\ b) = sqrt\ ((norm\ a)^2 + (norm\ b)^2)$
unfolding *norm-prod-def* **by** *simp*

instance proof
fix *r* :: *real* **and** *x y* :: $'a \times 'b$
show $0 \leq norm\ x$
unfolding *norm-prod-def* **by** *simp*
show $norm\ x = 0 \longleftrightarrow x = 0$
unfolding *norm-prod-def*
by (*simp add*: *expand-prod-eq*)
show $norm\ (x + y) \leq norm\ x + norm\ y$
unfolding *norm-prod-def*

```
    apply (rule order-trans [OF - real-sqrt-sum-squares-triangle-ineq])
    apply (simp add: add-mono power-mono norm-triangle-ineq)
    done
  show norm (scaleR r x) = |r| * norm x
    unfolding norm-prod-def
    apply (simp add: power-mult-distrib)
    apply (simp add: right-distrib [symmetric])
    apply (simp add: real-sqrt-mult-distrib)
    done
  show sgn x = scaleR (inverse (norm x)) x
    by (rule sgn-prod-def)
  show dist x y = norm (x - y)
    unfolding dist-prod-def norm-prod-def
    by (simp add: dist-norm)
qed

end

instance * :: (banach, banach) banach ..
```

20.7 Product is an inner product space

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

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

lemma inner-Pair [simp]: inner (a, b) (c, d) = inner a c + inner b d
  unfolding inner-prod-def by simp

instance proof
  fix r :: real
  fix x y z :: 'a::real-inner * 'b::real-inner
  show inner x y = inner y x
    unfolding inner-prod-def
    by (simp add: inner-commute)
  show inner (x + y) z = inner x z + inner y z
    unfolding inner-prod-def
    by (simp add: inner-add-left)
  show inner (scaleR r x) y = r * inner x y
    unfolding inner-prod-def
    by (simp add: right-distrib)
  show 0 ≤ inner x x
    unfolding inner-prod-def
    by (intro add-nonneg-nonneg inner-ge-zero)
  show inner x x = 0 ⟷ x = 0
    unfolding inner-prod-def expand-prod-eq
    by (simp add: add-nonneg-eq-0-iff)
```

show *norm x = sqrt (inner x x)*
unfolding *norm-prod-def inner-prod-def*
by (*simp add*: *power2-norm-eq-inner*)
qed

end

20.8 Pair operations are linear

interpretation *fst*: *bounded-linear fst*
apply (*unfold-locales*)
apply (*rule fst-add*)
apply (*rule fst-scaleR*)
apply (*rule-tac x=1* **in** *exI*, *simp add*: *norm-Pair*)
done

interpretation *snd*: *bounded-linear snd*
apply (*unfold-locales*)
apply (*rule snd-add*)
apply (*rule snd-scaleR*)
apply (*rule-tac x=1* **in** *exI*, *simp add*: *norm-Pair*)
done

TODO: move to NthRoot

lemma *sqrt-add-le-add-sqrt*:
assumes *x*: *0 ≤ x* **and** *y*: *0 ≤ y*
shows *sqrt (x + y) ≤ sqrt x + sqrt y*
apply (*rule power2-le-imp-le*)
apply (*simp add*: *real-sum-squared-expand add-nonneg-nonneg x y*)
apply (*simp add*: *mult-nonneg-nonneg x y*)
apply (*simp add*: *add-nonneg-nonneg x y*)
done

lemma *bounded-linear-Pair*:
assumes *f*: *bounded-linear f*
assumes *g*: *bounded-linear g*
shows *bounded-linear* (λ*x*. (*f x*, *g x*))
proof
interpret *f*: *bounded-linear f* **by** *fact*
interpret *g*: *bounded-linear g* **by** *fact*
fix *x y* **and** *r* :: *real*
show (*f* (*x + y*), *g* (*x + y*)) = (*f x*, *g x*) + (*f y*, *g y*)
by (*simp add*: *f.add g.add*)
show (*f* (*r* ∗*R x*), *g* (*r* ∗*R x*)) = *r* ∗*R* (*f x*, *g x*)
by (*simp add*: *f.scaleR g.scaleR*)
obtain *Kf* **where** *0 < Kf* **and** *norm-f*: ⋀*x*. *norm* (*f x*) ≤ *norm x* ∗ *Kf*
using *f.pos-bounded* **by** *fast*
obtain *Kg* **where** *0 < Kg* **and** *norm-g*: ⋀*x*. *norm* (*g x*) ≤ *norm x* ∗ *Kg*
using *g.pos-bounded* **by** *fast*
have ∀*x*. *norm* (*f x*, *g x*) ≤ *norm x* ∗ (*Kf* + *Kg*)

apply (*rule allI*)
apply (*simp add*: *norm-Pair*)
apply (*rule order-trans* [*OF sqrt-add-le-add-sqrt*], *simp*, *simp*)
apply (*simp add*: *right-distrib*)
apply (*rule add-mono* [*OF norm-f norm-g*])
done
then show $\exists K.\ \forall x.\ norm\ (f\ x,\ g\ x) \leq norm\ x * K$ **..**
qed

20.9 Frechet derivatives involving pairs

lemma *FDERIV-Pair*:
assumes *f*: *FDERIV f x :> f'* **and** *g*: *FDERIV g x :> g'*
shows *FDERIV* $(\lambda x.\ (f\ x,\ g\ x))\ x :> (\lambda h.\ (f'\ h,\ g'\ h))$
apply (*rule FDERIV-I*)
apply (*rule bounded-linear-Pair*)
apply (*rule FDERIV-bounded-linear* [*OF f*])
apply (*rule FDERIV-bounded-linear* [*OF g*])
apply (*simp add*: *norm-Pair*)
apply (*rule real-LIM-sandwich-zero*)
apply (*rule LIM-add-zero*)
apply (*rule FDERIV-D* [*OF f*])
apply (*rule FDERIV-D* [*OF g*])
apply (*rename-tac h*)
apply (*simp add*: *divide-nonneg-pos*)
apply (*rename-tac h*)
apply (*subst add-divide-distrib* [*symmetric*])
apply (*rule divide-right-mono* [*OF - norm-ge-zero*])
apply (*rule order-trans* [*OF sqrt-add-le-add-sqrt*])
apply *simp*
apply *simp*
apply *simp*
done

end

21 Convex: Convexity in real vector spaces

theory *Convex*
imports *Product-Vector*
begin

21.1 Convexity.

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

lemma *convex-alt*:
convex s $\longleftrightarrow$ $(\forall x{\in}s.\ \forall y{\in}s.\ \forall u.\ 0 \le u \wedge u \le 1 \longrightarrow ((1 - u) *_R x + u *_R y) \in s)$
(**is** - $\longleftrightarrow$ *?alt*)
proof
assume *alt*[*rule-format*]: *?alt*
{ fix *x y* **and** *u v* :: *real* **assume** *mem*: $x \in s$ $y \in s$
assume $0 \le u$ $0 \le v$ $u + v = 1$
moreover hence $u = 1 - v$ **by** *auto*
ultimately have $u *_R x + v *_R y \in s$ **using** *alt*[*OF mem*] **by** *auto* **}**
thus *convex s* **unfolding** *convex-def* **by** *auto*
qed (*auto simp*: *convex-def*)

lemma *mem-convex*:
assumes *convex s* $a \in s$ $b \in s$ $0 \le u$ $u \le 1$
shows $((1 - u) *_R a + u *_R b) \in s$
using *assms* **unfolding** *convex-alt* **by** *auto*

lemma *convex-empty*[*intro*]: *convex* {}
unfolding *convex-def* **by** *simp*

lemma *convex-singleton*[*intro*]: *convex* {*a*}
unfolding *convex-def* **by** (*auto simp*: *scaleR-left-distrib*[*symmetric*])

lemma *convex-UNIV*[*intro*]: *convex UNIV*
unfolding *convex-def* **by** *auto*

lemma *convex-Inter*: $(\forall s{\in}f.\ convex\ s) ==> convex(\bigcap f)$
unfolding *convex-def* **by** *auto*

lemma *convex-Int*: *convex s* $\Longrightarrow$ *convex t* $\Longrightarrow$ *convex* $(s \cap t)$
unfolding *convex-def* **by** *auto*

lemma *convex-halfspace-le*: *convex* $\{x.\ inner\ a\ x \le b\}$
unfolding *convex-def*
by (*auto simp*: *inner-add inner-scaleR intro*!: *convex-bound-le*)

lemma *convex-halfspace-ge*: *convex* $\{x.\ inner\ a\ x \ge b\}$
proof −
have $*$:$\{x.\ inner\ a\ x \ge b\} = \{x.\ inner\ (-a)\ x \le -b\}$ **by** *auto*
show *?thesis* **unfolding** $*$ **using** *convex-halfspace-le*[*of* $-a$ $-b$] **by** *auto*
qed

lemma *convex-hyperplane*: *convex* $\{x.\ inner\ a\ x = b\}$
proof−
have $*$:$\{x.\ inner\ a\ x = b\} = \{x.\ inner\ a\ x \le b\} \cap \{x.\ inner\ a\ x \ge b\}$ **by** *auto*
show *?thesis* **using** *convex-halfspace-le convex-halfspace-ge*
by (*auto intro*!: *convex-Int simp*: $*$)

qed

lemma *convex-halfspace-lt*: *convex* $\{x.\ inner\ a\ x < b\}$
unfolding *convex-def*
by (*auto simp*: *convex-bound-lt inner-add*)

lemma *convex-halfspace-gt*: *convex* $\{x.\ inner\ a\ x > b\}$
using *convex-halfspace-lt*[*of* $-a$ $-b$] **by** *auto*

lemma *convex-real-interval*:
fixes *a b* :: *real*
shows *convex* $\{a..\}$ **and** *convex* $\{..b\}$
and *convex* $\{a<..\}$ **and** *convex* $\{..<b\}$
and *convex* $\{a..b\}$ **and** *convex* $\{a<..b\}$
and *convex* $\{a..<b\}$ **and** *convex* $\{a<..<b\}$
proof −
have $\{a..\} = \{x.\ a \leq inner\ 1\ x\}$ **by** *auto*
thus *1*: *convex* $\{a..\}$ **by** (*simp only*: *convex-halfspace-ge*)
have $\{..b\} = \{x.\ inner\ 1\ x \leq b\}$ **by** *auto*
thus *2*: *convex* $\{..b\}$ **by** (*simp only*: *convex-halfspace-le*)
have $\{a<..\} = \{x.\ a < inner\ 1\ x\}$ **by** *auto*
thus *3*: *convex* $\{a<..\}$ **by** (*simp only*: *convex-halfspace-gt*)
have $\{..<b\} = \{x.\ inner\ 1\ x < b\}$ **by** *auto*
thus *4*: *convex* $\{..<b\}$ **by** (*simp only*: *convex-halfspace-lt*)
have $\{a..b\} = \{a..\} \cap \{..b\}$ **by** *auto*
thus *convex* $\{a..b\}$ **by** (*simp only*: *convex-Int 1 2*)
have $\{a<..b\} = \{a<..\} \cap \{..b\}$ **by** *auto*
thus *convex* $\{a<..b\}$ **by** (*simp only*: *convex-Int 3 2*)
have $\{a..<b\} = \{a..\} \cap \{..<b\}$ **by** *auto*
thus *convex* $\{a..<b\}$ **by** (*simp only*: *convex-Int 1 4*)
have $\{a<..<b\} = \{a<..\} \cap \{..<b\}$ **by** *auto*
thus *convex* $\{a<..<b\}$ **by** (*simp only*: *convex-Int 3 4*)
qed

21.2 Explicit expressions for convexity in terms of arbitrary sums.

lemma *convex-setsum*:
fixes *C* :: *'a*::*real-vector set*
assumes *finite s* **and** *convex C* **and** $(\sum\ i \in s.\ a\ i) = 1$
assumes $\bigwedge i.\ i \in s \Longrightarrow a\ i \geq 0$ **and** $\bigwedge i.\ i \in s \Longrightarrow y\ i \in C$
shows $(\sum\ j \in s.\ a\ j *_R y\ j) \in C$
using *assms*
proof (*induct s arbitrary*:*a rule*:*finite-induct*)
case *empty* **thus** *?case* **by** *auto*
next
case (*insert i s*) **note** *asms* = *this*
{ assume *a i* = *1*
hence $(\sum\ j \in s.\ a\ j) = 0$

```
    using asms by auto
  hence ⋀ j. j ∈ s ⟹ a j = 0
    using setsum-nonneg-0[where 'b=real] asms by fastsimp
  hence ?case using asms by auto }
 moreover
 { assume asm: a i ≠ 1
  from asms have yai: y i ∈ C a i ≥ 0 by auto
  have fis: finite (insert i s) using asms by auto
  hence ai1: a i ≤ 1 using setsum-nonneg-leq-bound[of insert i s a 1] asms by
simp
  hence a i < 1 using asm by auto
  hence i0: 1 − a i > 0 by auto
  let ?a j = a j / (1 − a i)
  { fix j assume j ∈ s
   hence ?a j ≥ 0
    using i0 asms divide-nonneg-pos
    by fastsimp } note a-nonneg = this
  have (∑ j ∈ insert i s. a j) = 1 using asms by auto
  hence (∑ j ∈ s. a j) = 1 − a i using setsum.insert asms by fastsimp
  hence (∑ j ∈ s. a j) / (1 − a i) = 1 using i0 by auto
  hence a1: (∑ j ∈ s. ?a j) = 1 unfolding divide.setsum by simp
  from this asms
  have (∑j∈s. ?a j *R y j) ∈ C using a-nonneg by fastsimp
  hence a i *R y i + (1 − a i) *R (∑ j ∈ s. ?a j *R y j) ∈ C
    using asms[unfolded convex-def, rule-format] yai ai1 by auto
  hence a i *R y i + (∑ j ∈ s. (1 − a i) *R (?a j *R y j)) ∈ C
    using scaleR-right.setsum[of (1 − a i) λ j. ?a j *R y j s] by auto
  hence a i *R y i + (∑ j ∈ s. a j *R y j) ∈ C using i0 by auto
  hence ?case using setsum.insert asms by auto }
 ultimately show ?case by auto
qed

lemma convex:
 shows convex s ⟷ (∀(k::nat) u x. (∀ i. 1≤i ∧ i≤k ⟶ 0 ≤ u i ∧ x i ∈s) ∧
(setsum u {1..k} = 1)
        ⟶ setsum (λi. u i *R x i) {1..k} ∈ s)
proof safe
 fix k :: nat fix u :: nat ⇒ real fix x
 assume convex s
  ∀ i. 1 ≤ i ∧ i ≤ k ⟶ 0 ≤ u i ∧ x i ∈ s
  setsum u {1..k} = 1
 from this convex-setsum[of {1 .. k} s]
 show (∑j∈{1 .. k}. u j *R x j) ∈ s by auto
next
 assume asm: ∀ k u x. (∀ i :: nat. 1 ≤ i ∧ i ≤ k ⟶ 0 ≤ u i ∧ x i ∈ s) ∧
setsum u {1..k} = 1
  ⟶ (∑ i = 1..k. u i *R (x i :: 'a)) ∈ s
 { fix μ :: real fix x y :: 'a assume xy: x ∈ s y ∈ s assume mu: μ ≥ 0 μ ≤ 1
  let ?u i = if (i :: nat) = 1 then μ else 1 − μ
```

```
    let ?x i = if (i :: nat) = 1 then x else y
    have {1 :: nat .. 2} ∩ - {x. x = 1} = {2} by auto
    hence card: card ({1 :: nat .. 2} ∩ - {x. x = 1}) = 1 by simp
    hence setsum ?u {1 .. 2} = 1
      using setsum-cases[of {(1 :: nat) .. 2} λ x. x = 1 λ x. μ λ x. 1 - μ]
      by auto
    from this asm[rule-format, of 2 ?u ?x]
    have s: (∑j ∈ {1..2}. ?u j *R ?x j) ∈ s
      using mu xy by auto
    have grarr: (∑j ∈ {Suc (Suc 0)..2}. ?u j *R ?x j) = (1 - μ) *R y
      using setsum-head-Suc[of Suc (Suc 0) 2 λ j. (1 - μ) *R y] by auto
    from setsum-head-Suc[of Suc 0 2 λ j. ?u j *R ?x j, simplified this]
    have (∑j ∈ {1..2}. ?u j *R ?x j) = μ *R x + (1 - μ) *R y by auto
    hence (1 - μ) *R y + μ *R x ∈ s using s by (auto simp:add-commute) }
  thus convex s unfolding convex-alt by auto
qed


lemma convex-explicit:
  fixes s :: 'a::real-vector set
  shows convex s ⟷
  (∀ t u. finite t ∧ t ⊆ s ∧ (∀ x∈t. 0 ≤ u x) ∧ setsum u t = 1 ⟶ setsum (λx. u
x *R x) t ∈ s)
proof safe
  fix t fix u :: 'a ⇒ real
  assume convex s finite t
    t ⊆ s ∀ x∈t. 0 ≤ u x setsum u t = 1
  thus (∑ x∈t. u x *R x) ∈ s
    using convex-setsum[of t s u λ x. x] by auto
next
  assume asm0: ∀ t. ∀ u. finite t ∧ t ⊆ s ∧ (∀ x∈t. 0 ≤ u x)
    ∧ setsum u t = 1 ⟶ (∑ x∈t. u x *R x) ∈ s
  show convex s
    unfolding convex-alt
  proof safe
    fix x y fix μ :: real
    assume asm: x ∈ s y ∈ s 0 ≤ μ μ ≤ 1
    { assume x ≠ y
      hence (1 - μ) *R x + μ *R y ∈ s
        using asm0[rule-format, of {x, y} λ z. if z = x then 1 - μ else μ]
          asm by auto }
    moreover
    { assume x = y
      hence (1 - μ) *R x + μ *R y ∈ s
        using asm0[rule-format, of {x, y} λ z. 1]
          asm by (auto simp:field-simps real-vector.scale-left-diff-distrib) }
    ultimately show (1 - μ) *R x + μ *R y ∈ s by blast
  qed
qed
```

lemma *convex-finite*: **assumes** *finite s*
shows *convex s* $\longleftrightarrow$ $(\forall u.\ (\forall x{\in}s.\ 0 \le u\ x) \wedge$ *setsum u s* $= 1$
$\longrightarrow$ *setsum* $(\lambda x.\ u\ x *_R x)\ s \in s)$
unfolding *convex-explicit*
proof (*safe elim*!: *conjE*)
fix *t u* **assume** *sum*: $\forall u.\ (\forall x{\in}s.\ 0 \le u\ x) \wedge$ *setsum u s* $= 1 \longrightarrow (\sum x{\in}s.\ u\ x *_R x) \in s$
and *as*: *finite t t* $\subseteq s\ \forall x{\in}t.\ 0 \le u\ x$ *setsum u t* $= (1{::}real)$
have $*{:}s \cap t = t$ **using** *as*(*2*) **by** *auto*
have *if-distrib-arg*: $\bigwedge P\ f\ g\ x.$ (*if P then f else g*) *x* $=$ (*if P then f x else g x*) **by** *simp*
show $(\sum x{\in}t.\ u\ x *_R x) \in s$
using *sum*[*THEN spec*[**where** $x{=}\lambda x.$ *if* $x{\in}t$ *then u x else 0*]] *as* $*$
by (*auto simp*: *assms setsum-cases if-distrib if-distrib-arg*)
qed (*erule-tac* $x{=}s$ **in** *allE*, *erule-tac* $x{=}u$ **in** *allE*, *auto*)

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

lemma *convex-on-subset*: *convex-on t f* $\Longrightarrow s \subseteq t \Longrightarrow$ *convex-on s f*
unfolding *convex-on-def* **by** *auto*

lemma *convex-add*[*intro*]:
assumes *convex-on s f convex-on s g*
shows *convex-on s* $(\lambda x.\ f\ x + g\ x)$
proof−
{ fix *x y* **assume** $x{\in}s\ y{\in}s$ **moreover**
fix *u v* $::real$ **assume** $0 \le u\ 0 \le v\ u + v = 1$
ultimately have $f\ (u *_R x + v *_R y) + g\ (u *_R x + v *_R y) \le (u * f\ x + v * f\ y) + (u * g\ x + v * g\ y)$
using *assms* **unfolding** *convex-on-def* **by** (*auto simp add*:*add-mono*)
hence $f\ (u *_R x + v *_R y) + g\ (u *_R x + v *_R y) \le u * (f\ x + g\ x) + v * (f\ y + g\ y)$ **by** (*simp add*: *field-simps*) **}**
thus *?thesis* **unfolding** *convex-on-def* **by** *auto*
qed

lemma *convex-cmul*[*intro*]:
assumes $0 \le (c{::}real)$ *convex-on s f*
shows *convex-on s* $(\lambda x.\ c * f\ x)$
proof−
have $*{:}\bigwedge u\ c\ fx\ v\ fy\ ::real.\ u * (c * fx) + v * (c * fy) = c * (u * fx + v * fy)$ **by** (*simp add*: *field-simps*)
show *?thesis* **using** *assms*(*2*) **and** *mult-mono1*[*OF - assms*(*1*)] **unfolding** *convex-on-def* **and** $*$ **by** *auto*
qed

lemma *convex-lower*:
assumes *convex-on s f* $x \in s$ $y \in s$ $0 \leq u$ $0 \leq v$ $u + v = 1$
shows $f\ (u *_R x + v *_R y) \leq max\ (f\ x)\ (f\ y)$
proof−
let *?m* = $max\ (f\ x)\ (f\ y)$
have $u * f\ x + v * f\ y \leq u * max\ (f\ x)\ (f\ y) + v * max\ (f\ x)\ (f\ y)$
using *assms*(*4*,*5*) **by**(*auto simp add*: *mult-mono1 add-mono*)
also have ... = $max\ (f\ x)\ (f\ y)$ **using** *assms*(*6*) **unfolding** *distrib*[*THEN sym*]
by *auto*
finally show *?thesis*
using *assms* **unfolding** *convex-on-def* **by** *fastsimp*
qed

lemma *convex-distance*[*intro*]:
fixes *s* :: *'a*::*real-normed-vector set*
shows *convex-on s* ($\lambda x.$ *dist a x*)
proof(*auto simp add*: *convex-on-def dist-norm*)
fix *x y* **assume** *x*∈*s y*∈*s*
fix *u v* ::*real* **assume** *0* ≤ *u 0* ≤ *v u* + *v* = *1*
have *a* = *u* $*_R$ *a* + *v* $*_R$ *a* **unfolding** *scaleR-left-distrib*[*THEN sym*] **and**
‹*u*+*v*=*1*› **by** *simp*
hence ∗:*a* − (*u* $*_R$ *x* + *v* $*_R$ *y*) = (*u* $*_R$ (*a* − *x*)) + (*v* $*_R$ (*a* − *y*))
by (*auto simp add*: *algebra-simps*)
show *norm* (*a* − (*u* $*_R$ *x* + *v* $*_R$ *y*)) ≤ *u* ∗ *norm* (*a* − *x*) + *v* ∗ *norm* (*a* − *y*)
unfolding ∗ **using** *norm-triangle-ineq*[*of u* $*_R$ (*a* − *x*) *v* $*_R$ (*a* − *y*)]
using ‹*0* ≤ *u*› ‹*0* ≤ *v*› **by** *auto*
qed

21.3 Arithmetic operations on sets preserve convexity.

lemma *convex-scaling*:
assumes *convex s*
shows*convex* (($\lambda x.$ *c* $*_R$ *x*) ‘ *s*)
using *assms* **unfolding** *convex-def image-iff*
proof *safe*
fix *x xa y xb* :: *'a*::*real-vector* **fix** *u v* :: *real*
assume *asm*: ∀ *x*∈*s*. ∀ *y*∈*s*. ∀ *u*≥*0*. ∀ *v*≥*0*. *u* + *v* = *1* ⟶ *u* $*_R$ *x* + *v* $*_R$ *y* ∈ *s*
xa ∈ *s xb* ∈ *s 0* ≤ *u 0* ≤ *v u* + *v* = *1*
show ∃ *x*∈*s*. *u* $*_R$ *c* $*_R$ *xa* + *v* $*_R$ *c* $*_R$ *xb* = *c* $*_R$ *x*
using *bexI*[*of* - *u* $*_R$ *xa* +*v* $*_R$ *xb*] *asm* **by** (*auto simp add*: *algebra-simps*)
qed

lemma *convex-negations*: *convex s* ⟹ *convex* (($\lambda x.$ −*x*)‘ *s*)
using *assms* **unfolding** *convex-def image-iff*
proof *safe*
fix *x xa y xb* :: *'a*::*real-vector* **fix** *u v* :: *real*
assume *asm*: ∀ *x*∈*s*. ∀ *y*∈*s*. ∀ *u*≥*0*. ∀ *v*≥*0*. *u* + *v* = *1* ⟶ *u* $*_R$ *x* + *v* $*_R$ *y* ∈ *s*
xa ∈ *s xb* ∈ *s 0* ≤ *u 0* ≤ *v u* + *v* = *1*

show $\exists x{\in}s.\ u *_R - xa + v *_R - xb = - x$
using *bexI*[*of* - $u *_R xa + v *_R xb$] *asm* **by** *auto*
qed

lemma *convex-sums*:
assumes *convex s convex t*
shows *convex* $\{x + y |\ x\ y.\ x \in s \wedge y \in t\}$
using *assms* **unfolding** *convex-def image-iff*
proof *safe*
fix *xa xb ya yb* **assume** *xy*:$xa{\in}s\ xb{\in}s\ ya{\in}t\ yb{\in}t$
fix *u v* ::*real* **assume** *uv*:$0 \le u\ 0 \le v\ u + v = 1$
show $\exists x\ y.\ u *_R (xa + ya) + v *_R (xb + yb) = x + y \wedge x \in s \wedge y \in t$
using *exI*[*of* - $u *_R xa + v *_R xb$] *exI*[*of* - $u *_R ya + v *_R yb$]
assms[*unfolded convex-def*] *uv xy* **by** (*auto simp add*:*scaleR-right-distrib*)
qed

lemma *convex-differences*:
assumes *convex s convex t*
shows *convex* $\{x - y |\ x\ y.\ x \in s \wedge y \in t\}$
proof −
have $\{x - y |\ x\ y.\ x \in s \wedge y \in t\} = \{x + y\ | x\ y.\ x \in s \wedge y \in uminus\ `\ t\}$
proof *safe*
fix *x x' y* **assume** $x' \in s\ y \in t$
thus $\exists x\ y'.\ x' - y = x + y' \wedge x \in s \wedge y' \in uminus\ `\ t$
using *exI*[*of* - x'] *exI*[*of* - $-y$] **by** *auto*
next
fix *x x' y y'* **assume** $x' \in s\ y' \in t$
thus $\exists x\ y.\ x' + - y' = x - y \wedge x \in s \wedge y \in t$
using *exI*[*of* - x'] *exI*[*of* - y'] **by** *auto*
qed
thus *?thesis* **using** *convex-sums*[*OF assms*(*1*) *convex-negations*[*OF assms*(*2*)]]
by *auto*
qed

lemma *convex-translation*: **assumes** *convex s* **shows** *convex* $((\lambda x.\ a + x)\ `\ s)$
proof− **have** $\{a + y\ | y.\ y \in s\} = (\lambda x.\ a + x)\ `\ s$ **by** *auto*
thus *?thesis* **using** *convex-sums*[*OF convex-singleton*[*of a*] *assms*] **by** *auto* **qed**

lemma *convex-affinity*: **assumes** *convex s* **shows** *convex* $((\lambda x.\ a + c *_R x)\ `\ s)$
proof− **have** $(\lambda x.\ a + c *_R x)\ `\ s = op + a\ `\ op *_R c\ `\ s$ **by** *auto*
thus *?thesis* **using** *convex-translation*[*OF convex-scaling*[*OF assms*], *of a c*] **by**
auto **qed**

lemma *convex-linear-image*:
assumes *c*:*convex s* **and** *l*:*bounded-linear f*
shows *convex*(*f* ` *s*)
proof(*auto simp add*: *convex-def*)
interpret *f*: *bounded-linear f* **by** *fact*
fix *x y* **assume** *xy*:$x \in s\ y \in s$

fix $u\ v$ ::*real* **assume** *uv*:$0 \leq u\ 0 \leq v\ u + v = 1$
show $u *_R f\ x + v *_R f\ y \in f\ `\ s$ **unfolding** *image-iff*
using *bexI*[*of* - $u *_R x + v *_R y$] *f.add f.scaleR*
c[*unfolded convex-def*] *xy uv* **by** *auto*
qed

lemma *pos-is-convex*:
shows *convex* $\{0 :: real <..\}$
unfolding *convex-alt*
proof *safe*
fix $y\ x\ \mu :: real$
assume *asms*: $y > 0\ x > 0\ \mu \geq 0\ \mu \leq 1$
{ **assume** $\mu = 0$
hence $\mu *_R x + (1 - \mu) *_R y = y$ **by** *simp*
hence $\mu *_R x + (1 - \mu) *_R y > 0$ **using** *asms* **by** *simp* }
moreover
{ **assume** $\mu = 1$
hence $\mu *_R x + (1 - \mu) *_R y > 0$ **using** *asms* **by** *simp* }
moreover
{ **assume** $\mu \neq 1\ \mu \neq 0$
hence $\mu > 0\ (1 - \mu) > 0$ **using** *asms* **by** *auto*
hence $\mu *_R x + (1 - \mu) *_R y > 0$ **using** *asms*
by (*auto simp add*: *add-pos-pos mult-pos-pos*) }
ultimately show $(1 - \mu) *_R y + \mu *_R x > 0$ **using** *assms* **by** *fastsimp*
qed

lemma *convex-on-setsum*:
fixes $a :: 'a \Rightarrow real$
fixes $y :: 'a \Rightarrow 'b$::*real-vector*
fixes $f :: 'b \Rightarrow real$
assumes *finite s* $s \neq \{\}$
assumes *convex-on C f*
assumes *convex C*
assumes $(\sum\ i \in s.\ a\ i) = 1$
assumes $\bigwedge i.\ i \in s \Longrightarrow a\ i \geq 0$
assumes $\bigwedge i.\ i \in s \Longrightarrow y\ i \in C$
shows $f\ (\sum\ i \in s.\ a\ i *_R y\ i) \leq (\sum\ i \in s.\ a\ i * f\ (y\ i))$
using *assms*
proof (*induct s arbitrary*:*a rule*:*finite-ne-induct*)
case (*singleton i*)
hence *ai*: $a\ i = 1$ **by** *auto*
thus *?case* **by** *auto*
next
case (*insert i s*) **note** *asms* = *this*
hence *convex-on C f* **by** *simp*
from *this*[*unfolded convex-on-def*, *rule-format*]
have *conv*: $\bigwedge x\ y\ \mu.\ [\![x \in C;\ y \in C;\ 0 \leq \mu;\ \mu \leq 1]\!]$
$\Longrightarrow f\ (\mu *_R x + (1 - \mu) *_R y) \leq \mu * f\ x + (1 - \mu) * f\ y$

```
    by simp
  { assume a i = 1
    hence (∑ j ∈ s. a j) = 0
      using asms by auto
    hence ⋀ j. j ∈ s ⟹ a j = 0
      using setsum-nonneg-0[where 'b=real] asms by fastsimp
    hence ?case using asms by auto }
  moreover
  { assume asm: a i ≠ 1
    from asms have yai: y i ∈ C a i ≥ 0 by auto
    have fis: finite (insert i s) using asms by auto
    hence ai1: a i ≤ 1 using setsum-nonneg-leq-bound[of insert i s a] asms by
simp
    hence a i < 1 using asm by auto
    hence i0: 1 − a i > 0 by auto
    let ?a j = a j / (1 − a i)
    { fix j assume j ∈ s
      hence ?a j ≥ 0
        using i0 asms divide-nonneg-pos
        by fastsimp } note a-nonneg = this
    have (∑ j ∈ insert i s. a j) = 1 using asms by auto
    hence (∑ j ∈ s. a j) = 1 − a i using setsum.insert asms by fastsimp
    hence (∑ j ∈ s. a j) / (1 − a i) = 1 using i0 by auto
    hence a1: (∑ j ∈ s. ?a j) = 1 unfolding divide.setsum by simp
    have convex C using asms by auto
    hence asum: (∑ j ∈ s. ?a j *R y j) ∈ C
      using asms convex-setsum[OF ‹finite s›
        ‹convex C› a1 a-nonneg] by auto
    have asum-le: f (∑ j ∈ s. ?a j *R y j) ≤ (∑ j ∈ s. ?a j * f (y j))
      using a-nonneg a1 asms by blast
   have f (∑ j ∈ insert i s. a j *R y j) = f ((∑ j ∈ s. a j *R y j) + a i *R y i)
      using setsum.insert[of s i λ j. a j *R y j, OF ‹finite s› ‹i ∉ s›] asms
      by (auto simp only:add-commute)
    also have ... = f (((1 − a i) * inverse (1 − a i)) *R (∑ j ∈ s. a j *R y j)
+ a i *R y i)
      using i0 by auto
    also have ... = f ((1 − a i) *R (∑ j ∈ s. (a j * inverse (1 − a i)) *R y j)
+ a i *R y i)
      using scaleR-right.setsum[of inverse (1 − a i) λ j. a j *R y j s, symmetric]
by (auto simp:algebra-simps)
    also have ... = f ((1 − a i) *R (∑ j ∈ s. ?a j *R y j) + a i *R y i)
      by (auto simp: divide-inverse)
    also have ... ≤ (1 − a i) *R f ((∑ j ∈ s. ?a j *R y j)) + a i * f (y i)
      using conv[of y i (∑ j ∈ s. ?a j *R y j) a i, OF yai(1) asum yai(2) ai1]
      by (auto simp add:add-commute)
    also have ... ≤ (1 − a i) * (∑ j ∈ s. ?a j * f (y j)) + a i * f (y i)
      using add-right-mono[OF mult-left-mono[of - - 1 − a i,
        OF asum-le less-imp-le[OF i0]], of a i * f (y i)] by simp
    also have ... = (∑ j ∈ s. (1 − a i) * ?a j * f (y j)) + a i * f (y i)
```

unfolding *mult-right.setsum*[*of* $1 - a\ i\ \lambda\ j.\ ?a\ j * f\ (y\ j)$] **using** *i0* **by** *auto*
also have $\ldots = (\sum\ j \in s.\ a\ j * f\ (y\ j)) + a\ i * f\ (y\ i)$ **using** *i0* **by** *auto*
also have $\ldots = (\sum\ j \in insert\ i\ s.\ a\ j * f\ (y\ j))$ **using** *asms* **by** *auto*
finally have $f\ (\sum\ j \in insert\ i\ s.\ a\ j *_R\ y\ j) \leq (\sum\ j \in insert\ i\ s.\ a\ j * f\ (y\ j))$
by *simp* **}**
ultimately show *?case* **by** *auto*
qed

lemma *convex-on-alt*:
fixes $C :: 'a$::*real-vector set*
assumes *convex C*
shows *convex-on C f* =
$(\forall\ x \in C.\ \forall\ y \in C.\ \forall\ \mu :: real.\ \mu \geq 0 \wedge \mu \leq 1 \longrightarrow f\ (\mu *_R\ x + (1 - \mu) *_R\ y) \leq \mu * f\ x + (1 - \mu) * f\ y)$
proof *safe*
fix $x\ y$ **fix** $\mu :: real$
assume *asms*: *convex-on C f* $x \in C\ y \in C\ 0 \leq \mu\ \mu \leq 1$
from *this*[*unfolded convex-on-def*, *rule-format*]
have $\bigwedge\ u\ v.\ [\![0 \leq u;\ 0 \leq v;\ u + v = 1]\!] \Longrightarrow f\ (u *_R\ x + v *_R\ y) \leq u * f\ x + v * f\ y$ **by** *auto*
from *this*[*of* $\mu\ 1 - \mu$, *simplified*] *asms*
show $f\ (\mu *_R\ x + (1 - \mu) *_R\ y) \leq \mu * f\ x + (1 - \mu) * f\ y$ **by** *auto*
next
assume *asm*: $\forall x \in C.\ \forall y \in C.\ \forall \mu.\ 0 \leq \mu \wedge \mu \leq 1 \longrightarrow f\ (\mu *_R\ x + (1 - \mu) *_R\ y) \leq \mu * f\ x + (1 - \mu) * f\ y$
{fix $x\ y$ **fix** $u\ v :: real$
assume *lasm*: $x \in C\ y \in C\ u \geq 0\ v \geq 0\ u + v = 1$
hence[*simp*]: $1 - u = v$ **by** *auto*
from *asm*[*rule-format*, *of x y u*]
have $f\ (u *_R\ x + v *_R\ y) \leq u * f\ x + v * f\ y$ **using** *lasm* **by** *auto* **}**
thus *convex-on C f* **unfolding** *convex-on-def* **by** *auto*
qed

lemma *pos-convex-function*:
fixes $f :: real \Rightarrow real$
assumes *convex C*
assumes *leq*: $\bigwedge\ x\ y.\ [\![x \in C\ ;\ y \in C]\!] \Longrightarrow f'\ x * (y - x) \leq f\ y - f\ x$
shows *convex-on C f*
unfolding *convex-on-alt*[*OF assms*(*1*)]
using *assms*
proof *safe*
fix $x\ y\ \mu :: real$
let $?x = \mu *_R\ x + (1 - \mu) *_R\ y$
assume *asm*: *convex C* $x \in C\ y \in C\ \mu \geq 0\ \mu \leq 1$
hence $1 - \mu \geq 0$ **by** *auto*
hence *xpos*: $?x \in C$ **using** *asm* **unfolding** *convex-alt* **by** *fastsimp*

```
  have geq: μ * (f x - f ?x) + (1 - μ) * (f y - f ?x)
          ≥ μ * f' ?x * (x - ?x) + (1 - μ) * f' ?x * (y - ?x)
    using add-mono[OF mult-mono1[OF leq[OF xpos asm(2)] ‹μ ≥ 0›]
      mult-mono1[OF leq[OF xpos asm(3)] ‹1 - μ ≥ 0›]] by auto
  hence μ * f x + (1 - μ) * f y - f ?x ≥ 0
    by (auto simp add:field-simps)
  thus f (μ *R x + (1 - μ) *R y) ≤ μ * f x + (1 - μ) * f y
    using convex-on-alt by auto
qed

lemma atMostAtLeast-subset-convex:
  fixes C :: real set
  assumes convex C
  assumes x ∈ C y ∈ C x < y
  shows {x .. y} ⊆ C
proof safe
  fix z assume zasm: z ∈ {x .. y}
  { assume asm: x < z z < y
    let ?μ = (y - z) / (y - x)
    have 0 ≤ ?μ ?μ ≤ 1 using assms asm by (auto simp add:field-simps)
    hence comb: ?μ * x + (1 - ?μ) * y ∈ C
        using assms iffD1[OF convex-alt, rule-format, of C y x ?μ] by (simp
add:algebra-simps)
    have ?μ * x + (1 - ?μ) * y = (y - z) * x / (y - x) + (1 - (y - z) / (y -
x)) * y
      by (auto simp add:field-simps)
    also have ... = ((y - z) * x + (y - x - (y - z)) * y) / (y - x)
      using assms unfolding add-divide-distrib by (auto simp:field-simps)
    also have ... = z
      using assms by (auto simp:field-simps)
    finally have z ∈ C
      using comb by auto } note less = this
  show z ∈ C using zasm less assms
    unfolding atLeastAtMost-iff le-less by auto
qed

lemma f''-imp-f':
  fixes f :: real ⇒ real
  assumes convex C
  assumes f': ⋀ x. x ∈ C ⟹ DERIV f x :> (f' x)
  assumes f'': ⋀ x. x ∈ C ⟹ DERIV f' x :> (f'' x)
  assumes pos: ⋀ x. x ∈ C ⟹ f'' x ≥ 0
  assumes x ∈ C y ∈ C
  shows f' x * (y - x) ≤ f y - f x
using assms
proof -
  { fix x y :: real assume asm: x ∈ C y ∈ C y > x
    hence ge: y - x > 0 y - x ≥ 0 by auto
    from asm have le: x - y < 0 x - y ≤ 0 by auto
```

then obtain *z1* **where** *z1*: $z1 > x\ z1 < y\ f\ y - f\ x = (y - x) * f'\ z1$
using *subsetD*[*OF atMostAtLeast-subset-convex*[*OF* ‹*convex C*› ‹$x \in C$› ‹$y \in C$› ‹$x < y$›],
THEN f', *THEN MVT2*[*OF* ‹$x < y$›, *rule-format*, *unfolded atLeastAtMost-iff*[*symmetric*]]]
by *auto*
hence $z1 \in C$ **using** *atMostAtLeast-subset-convex*
‹*convex C*› ‹$x \in C$› ‹$y \in C$› ‹$x < y$› **by** *fastsimp*
from *z1* **have** *z1'*: $f\ x - f\ y = (x - y) * f'\ z1$
by (*simp add*:*field-simps*)
obtain *z2* **where** *z2*: $z2 > x\ z2 < z1\ f'\ z1 - f'\ x = (z1 - x) * f''\ z2$
using *subsetD*[*OF atMostAtLeast-subset-convex*[*OF* ‹*convex C*› ‹$x \in C$› ‹*z1* $\in C$› ‹$x < z1$›],
THEN f'', *THEN MVT2*[*OF* ‹$x < z1$›, *rule-format*, *unfolded atLeastAtMost-iff*[*symmetric*]]]
z1
by *auto*
obtain *z3* **where** *z3*: $z3 > z1\ z3 < y\ f'\ y - f'\ z1 = (y - z1) * f''\ z3$
using *subsetD*[*OF atMostAtLeast-subset-convex*[*OF* ‹*convex C*› ‹*z1* $\in C$› ‹*y*
$\in C$› ‹$z1 < y$›],
THEN f'', *THEN MVT2*[*OF* ‹$z1 < y$›, *rule-format*, *unfolded atLeastAtMost-iff*[*symmetric*]]]
z1
by *auto*
have $f'\ y - (f\ x - f\ y)\ /\ (x - y) = f'\ y - f'\ z1$
using *asm z1'* **by** *auto*
also have $\ldots = (y - z1) * f''\ z3$ **using** *z3* **by** *auto*
finally have *cool'*: $f'\ y - (f\ x - f\ y)\ /\ (x - y) = (y - z1) * f''\ z3$ **by** *simp*
have *A'*: $y - z1 \geq 0$ **using** *z1* **by** *auto*
have $z3 \in C$ **using** *z3 asm atMostAtLeast-subset-convex*
‹*convex C*› ‹$x \in C$› ‹*z1* $\in C$› ‹$x < z1$› **by** *fastsimp*
hence *B'*: $f''\ z3 \geq 0$ **using** *assms* **by** *auto*
from *A' B'* **have** $(y - z1) * f''\ z3 \geq 0$ **using** *mult-nonneg-nonneg* **by** *auto*
from *cool' this* **have** $f'\ y - (f\ x - f\ y)\ /\ (x - y) \geq 0$ **by** *auto*
from *mult-right-mono-neg*[*OF this le*(*2*)]
have $f'\ y * (x - y) - (f\ x - f\ y)\ /\ (x - y) * (x - y) \leq 0 * (x - y)$
by (*simp add*: *algebra-simps*)
hence $f'\ y * (x - y) - (f\ x - f\ y) \leq 0$ **using** *le* **by** *auto*
hence *res*: $f'\ y * (x - y) \leq f\ x - f\ y$ **by** *auto*
have $(f\ y - f\ x)\ /\ (y - x) - f'\ x = f'\ z1 - f'\ x$
using *asm z1* **by** *auto*
also have $\ldots = (z1 - x) * f''\ z2$ **using** *z2* **by** *auto*
finally have *cool*: $(f\ y - f\ x)\ /\ (y - x) - f'\ x = (z1 - x) * f''\ z2$ **by** *simp*
have *A*: $z1 - x \geq 0$ **using** *z1* **by** *auto*
have $z2 \in C$ **using** *z2 z1 asm atMostAtLeast-subset-convex*
‹*convex C*› ‹*z1* $\in C$› ‹$y \in C$› ‹$z1 < y$› **by** *fastsimp*
hence *B*: $f''\ z2 \geq 0$ **using** *assms* **by** *auto*
from *A B* **have** $(z1 - x) * f''\ z2 \geq 0$ **using** *mult-nonneg-nonneg* **by** *auto*
from *cool this* **have** $(f\ y - f\ x)\ /\ (y - x) - f'\ x \geq 0$ **by** *auto*
from *mult-right-mono*[*OF this ge*(*2*)]
have $(f\ y - f\ x)\ /\ (y - x) * (y - x) - f'\ x * (y - x) \geq 0 * (y - x)$
by (*simp add*: *algebra-simps*)

```
    hence f y − f x − f' x ∗ (y − x) ≥ 0 using ge by auto
    hence f y − f x ≥ f' x ∗ (y − x) f' y ∗ (x − y) ≤ f x − f y
      using res by auto } note less-imp = this
  { fix x y :: real assume x ∈ C y ∈ C x ≠ y
    hencef y − f x ≥ f' x ∗ (y − x)
    unfolding neq-iff using less-imp by auto } note neq-imp = this
  moreover
  { fix x y :: real assume asm: x ∈ C y ∈ C x = y
    hence f y − f x ≥ f' x ∗ (y − x) by auto }
  ultimately show ?thesis using assms by blast
qed

lemma f''-ge0-imp-convex:
  fixes f :: real ⇒ real
  assumes conv: convex C
  assumes f': ⋀ x. x ∈ C ⟹ DERIV f x :> (f' x)
  assumes f'': ⋀ x. x ∈ C ⟹ DERIV f' x :> (f'' x)
  assumes pos: ⋀ x. x ∈ C ⟹ f'' x ≥ 0
  shows convex-on C f
using f''-imp-f'[OF conv f' f'' pos] assms pos-convex-function by fastsimp

lemma minus-log-convex:
  fixes b :: real
  assumes b > 1
  shows convex-on {0 <..} (λ x. − log b x)
proof −
  have ⋀ z. z > 0 ⟹ DERIV (log b) z :> 1 / (ln b ∗ z) using DERIV-log by
auto
  hence f': ⋀ z. z > 0 ⟹ DERIV (λ z. − log b z) z :> − 1 / (ln b ∗ z)
    using DERIV-minus by auto
  have ⋀ z :: real. z > 0 ⟹ DERIV inverse z :> − (inverse z ^ Suc (Suc 0))
    using less-imp-neq[THEN not-sym, THEN DERIV-inverse] by auto
  from this[THEN DERIV-cmult, of - − 1 / ln b]
  have ⋀ z :: real. z > 0 ⟹ DERIV (λ z. (− 1 / ln b) ∗ inverse z) z :> (− 1
/ ln b) ∗ (− (inverse z ^ Suc (Suc 0)))
    by auto
  hence f''0: ⋀ z :: real. z > 0 ⟹ DERIV (λ z. − 1 / (ln b ∗ z)) z :> 1 / (ln
b ∗ z ∗ z)
    unfolding inverse-eq-divide by (auto simp add: mult-assoc)
  have f''-ge0: ⋀ z :: real. z > 0 ⟹ 1 / (ln b ∗ z ∗ z) ≥ 0
   using ‹b > 1› by (auto intro!:less-imp-le simp add:divide-pos-pos[of 1] mult-pos-pos)
  from f''-ge0-imp-convex[OF pos-is-convex,
    unfolded greaterThan-iff, OF f' f''0 f''-ge0]
  show ?thesis by auto
qed

end
```

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

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

22.1 Type $nat \times nat$

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

definition
triangle :: *nat* ⇒ *nat*
where
triangle n = *n* ∗ *Suc n div 2*

lemma *triangle-0* [*simp*]: *triangle 0* = *0*
unfolding *triangle-def* **by** *simp*

lemma *triangle-Suc* [*simp*]: *triangle* (*Suc n*) = *triangle n* + *Suc n*
unfolding *triangle-def* **by** *simp*

definition
prod-encode :: *nat* × *nat* ⇒ *nat*
where
prod-encode = (λ(*m*, *n*). *triangle* (*m* + *n*) + *m*)

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

fun
prod-decode-aux :: *nat* ⇒ *nat* ⇒ *nat* × *nat*
where
prod-decode-aux k m =
(*if m* ≤ *k then* (*m*, *k* − *m*) *else prod-decode-aux* (*Suc k*) (*m* − *Suc k*))

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

definition
prod-decode :: *nat* ⇒ *nat* × *nat*
where
prod-decode = *prod-decode-aux 0*

lemma *prod-encode-prod-decode-aux*:
prod-encode (*prod-decode-aux k m*) = *triangle k* + *m*
apply (*induct k m rule*: *prod-decode-aux.induct*)
apply (*subst prod-decode-aux.simps*)
apply (*simp add*: *prod-encode-def*)
done

lemma *prod-decode-inverse* [*simp*]: *prod-encode* (*prod-decode n*) = *n*
unfolding *prod-decode-def* **by** (*simp add*: *prod-encode-prod-decode-aux*)

lemma *prod-decode-triangle-add*:
prod-decode (*triangle k* + *m*) = *prod-decode-aux k m*
apply (*induct k arbitrary*: *m*)
apply (*simp add*: *prod-decode-def*)
apply (*simp only*: *triangle-Suc add-assoc*)
apply (*subst prod-decode-aux.simps*, *simp*)
done

lemma *prod-encode-inverse* [*simp*]: *prod-decode* (*prod-encode x*) = *x*
unfolding *prod-encode-def*
apply (*induct x*)
apply (*simp add*: *prod-decode-triangle-add*)
apply (*subst prod-decode-aux.simps*, *simp*)
done

lemma *inj-prod-encode*: *inj-on prod-encode A*
by (*rule inj-on-inverseI*, *rule prod-encode-inverse*)

lemma *inj-prod-decode*: *inj-on prod-decode A*
by (*rule inj-on-inverseI*, *rule prod-decode-inverse*)

lemma *surj-prod-encode*: *surj prod-encode*
by (*rule surjI*, *rule prod-decode-inverse*)

lemma *surj-prod-decode*: *surj prod-decode*
by (*rule surjI*, *rule prod-encode-inverse*)

lemma *bij-prod-encode*: *bij prod-encode*
by (*rule bijI* [*OF inj-prod-encode surj-prod-encode*])

lemma *bij-prod-decode*: *bij prod-decode*
by (*rule bijI* [*OF inj-prod-decode surj-prod-decode*])

lemma *prod-encode-eq*: *prod-encode x* = *prod-encode y* ⟷ *x* = *y*
by (*rule inj-prod-encode* [*THEN inj-eq*])

lemma *prod-decode-eq*: *prod-decode x* = *prod-decode y* ⟷ *x* = *y*
by (*rule inj-prod-decode* [*THEN inj-eq*])

Ordering properties

lemma *le-prod-encode-1*: *a* ≤ *prod-encode* (*a*, *b*)
unfolding *prod-encode-def* **by** *simp*

lemma *le-prod-encode-2*: *b* ≤ *prod-encode* (*a*, *b*)
unfolding *prod-encode-def* **by** (*induct b*, *simp-all*)

22.2 Type *nat* + *nat*

definition

sum-encode :: *nat* + *nat* ⇒ *nat*
where
sum-encode x = (*case x of Inl a* ⇒ *2* ∗ *a* | *Inr b* ⇒ *Suc* (*2* ∗ *b*))

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

lemma *sum-encode-inverse* [*simp*]: *sum-decode* (*sum-encode x*) = *x*
unfolding *sum-decode-def sum-encode-def*
by (*induct x*) *simp-all*

lemma *sum-decode-inverse* [*simp*]: *sum-encode* (*sum-decode n*) = *n*
unfolding *sum-decode-def sum-encode-def numeral-2-eq-2*
by (*simp add*: *even-nat-div-two-times-two odd-nat-div-two-times-two-plus-one*
del: *mult-Suc*)

lemma *inj-sum-encode*: *inj-on sum-encode A*
by (*rule inj-on-inverseI*, *rule sum-encode-inverse*)

lemma *inj-sum-decode*: *inj-on sum-decode A*
by (*rule inj-on-inverseI*, *rule sum-decode-inverse*)

lemma *surj-sum-encode*: *surj sum-encode*
by (*rule surjI*, *rule sum-decode-inverse*)

lemma *surj-sum-decode*: *surj sum-decode*
by (*rule surjI*, *rule sum-encode-inverse*)

lemma *bij-sum-encode*: *bij sum-encode*
by (*rule bijI* [*OF inj-sum-encode surj-sum-encode*])

lemma *bij-sum-decode*: *bij sum-decode*
by (*rule bijI* [*OF inj-sum-decode surj-sum-decode*])

lemma *sum-encode-eq*: *sum-encode x* = *sum-encode y* ⟷ *x* = *y*
by (*rule inj-sum-encode* [*THEN inj-eq*])

lemma *sum-decode-eq*: *sum-decode x* = *sum-decode y* ⟷ *x* = *y*
by (*rule inj-sum-decode* [*THEN inj-eq*])

22.3 Type *int*

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

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

lemma *int-encode-inverse* [*simp*]: *int-decode* (*int-encode x*) = *x*
unfolding *int-decode-def int-encode-def* **by** *simp*

lemma *int-decode-inverse* [*simp*]: *int-encode* (*int-decode n*) = *n*
unfolding *int-decode-def int-encode-def* **using** *sum-decode-inverse* [*of n*]
by (*cases sum-decode n*, *simp-all*)

lemma *inj-int-encode*: *inj-on int-encode A*
by (*rule inj-on-inverseI*, *rule int-encode-inverse*)

lemma *inj-int-decode*: *inj-on int-decode A*
by (*rule inj-on-inverseI*, *rule int-decode-inverse*)

lemma *surj-int-encode*: *surj int-encode*
by (*rule surjI*, *rule int-decode-inverse*)

lemma *surj-int-decode*: *surj int-decode*
by (*rule surjI*, *rule int-encode-inverse*)

lemma *bij-int-encode*: *bij int-encode*
by (*rule bijI* [*OF inj-int-encode surj-int-encode*])

lemma *bij-int-decode*: *bij int-decode*
by (*rule bijI* [*OF inj-int-decode surj-int-decode*])

lemma *int-encode-eq*: *int-encode x* = *int-encode y* ⟷ *x* = *y*
by (*rule inj-int-encode* [*THEN inj-eq*])

lemma *int-decode-eq*: *int-decode x* = *int-decode y* ⟷ *x* = *y*
by (*rule inj-int-decode* [*THEN inj-eq*])

22.4 Type *nat list*

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

function
list-decode :: *nat* ⇒ *nat list*
where
list-decode 0 = []
| *list-decode* (*Suc n*) = (*case prod-decode n of* (*x*, *y*) ⇒ *x* # *list-decode y*)

by *pat-completeness auto*

termination *list-decode*
apply (*relation measure id*, *simp-all*)
apply (*drule arg-cong* [**where** *f*=*prod-encode*])
apply (*simp add*: *le-imp-less-Suc le-prod-encode-2*)
done

lemma *list-encode-inverse* [*simp*]: *list-decode* (*list-encode x*) = *x*
by (*induct x rule*: *list-encode.induct*) *simp-all*

lemma *list-decode-inverse* [*simp*]: *list-encode* (*list-decode n*) = *n*
apply (*induct n rule*: *list-decode.induct*, *simp*)
apply (*simp split*: *prod.split*)
apply (*simp add*: *prod-decode-eq* [*symmetric*])
done

lemma *inj-list-encode*: *inj-on list-encode A*
by (*rule inj-on-inverseI*, *rule list-encode-inverse*)

lemma *inj-list-decode*: *inj-on list-decode A*
by (*rule inj-on-inverseI*, *rule list-decode-inverse*)

lemma *surj-list-encode*: *surj list-encode*
by (*rule surjI*, *rule list-decode-inverse*)

lemma *surj-list-decode*: *surj list-decode*
by (*rule surjI*, *rule list-encode-inverse*)

lemma *bij-list-encode*: *bij list-encode*
by (*rule bijI* [*OF inj-list-encode surj-list-encode*])

lemma *bij-list-decode*: *bij list-decode*
by (*rule bijI* [*OF inj-list-decode surj-list-decode*])

lemma *list-encode-eq*: *list-encode x* = *list-encode y* ⟷ *x* = *y*
by (*rule inj-list-encode* [*THEN inj-eq*])

lemma *list-decode-eq*: *list-decode x* = *list-decode y* ⟷ *x* = *y*
by (*rule inj-list-decode* [*THEN inj-eq*])

22.5 Finite sets of naturals

22.5.1 Preliminaries

lemma *finite-vimage-Suc-iff*: *finite* (*Suc* −‘ *F*) ⟷ *finite F*
apply (*safe intro*!: *finite-vimageI inj-Suc*)
apply (*rule finite-subset* [**where** *B*=*insert 0* (*Suc* ‘ *Suc* −‘ *F*)])
apply (*rule subsetI*, *case-tac x*, *simp*, *simp*)
apply (*rule finite-insert* [*THEN iffD2*])

apply (*erule finite-imageI*)
done

lemma *vimage-Suc-insert-0*: *Suc −‘ insert 0 A = Suc −‘ A*
by *auto*

lemma *vimage-Suc-insert-Suc*:
Suc −‘ insert (Suc n) A = insert n (Suc −‘ A)
by *auto*

lemma *even-nat-Suc-div-2*: *even x ⟹ Suc x div 2 = x div 2*
by (*simp only*: *numeral-2-eq-2 even-nat-plus-one-div-two*)

lemma *div2-even-ext-nat*:
⟦x div 2 = y div 2; *even x = even y⟧ ⟹ (x::nat) = y*
apply (*rule mod-div-equality* [*of x 2*, *THEN subst*])
apply (*rule mod-div-equality* [*of y 2*, *THEN subst*])
apply (*case-tac even x*)
apply (*simp add*: *numeral-2-eq-2 even-nat-equiv-def*)
apply (*simp add*: *numeral-2-eq-2 odd-nat-equiv-def*)
done

22.5.2 From sets to naturals

definition
set-encode :: *nat set ⇒ nat*
where
set-encode = setsum (*op ^ 2*)

lemma *set-encode-empty* [*simp*]: *set-encode {} = 0*
by (*simp add*: *set-encode-def*)

lemma *set-encode-insert* [*simp*]:
⟦finite A; *n ∉ A⟧ ⟹ set-encode* (*insert n A*) = *2^n + set-encode A*
by (*simp add*: *set-encode-def*)

lemma *even-set-encode-iff*: *finite A ⟹ even* (*set-encode A*) *⟷ 0 ∉ A*
unfolding *set-encode-def* **by** (*induct set*: *finite*, *auto*)

lemma *set-encode-vimage-Suc*: *set-encode* (*Suc −‘ A*) = *set-encode A div 2*
apply (*cases finite A*)
apply (*erule finite-induct*, *simp*)
apply (*case-tac x*)
apply (*simp add*: *even-nat-Suc-div-2 even-set-encode-iff vimage-Suc-insert-0*)
apply (*simp add*: *finite-vimageI add-commute vimage-Suc-insert-Suc*)
apply (*simp add*: *set-encode-def finite-vimage-Suc-iff*)
done

lemmas *set-encode-div-2 = set-encode-vimage-Suc* [*symmetric*]

22.5.3 From naturals to sets

definition
set-decode :: *nat* ⇒ *nat set*
where
set-decode x = {*n*. *odd* (*x div 2* ^ *n*)}

lemma *set-decode-0* [*simp*]: *0* ∈ *set-decode x* ⟷ *odd x*
by (*simp add*: *set-decode-def*)

lemma *set-decode-Suc* [*simp*]:
Suc n ∈ *set-decode x* ⟷ *n* ∈ *set-decode* (*x div 2*)
by (*simp add*: *set-decode-def div-mult2-eq*)

lemma *set-decode-zero* [*simp*]: *set-decode 0* = {}
by (*simp add*: *set-decode-def*)

lemma *set-decode-div-2*: *set-decode* (*x div 2*) = *Suc* −‘ *set-decode x*
by *auto*

lemma *set-decode-plus-power-2*:
n ∉ *set-decode z* ⟹ *set-decode* (*2* ^ *n* + *z*) = *insert n* (*set-decode z*)
apply (*induct n arbitrary*: *z*, *simp-all*)
apply (*rule set-ext*, *induct-tac x*, *simp*, *simp add*: *even-nat-Suc-div-2*)
apply (*rule set-ext*, *induct-tac x*, *simp*, *simp add*: *add-commute*)
done

lemma *finite-set-decode* [*simp*]: *finite* (*set-decode n*)
apply (*induct n rule*: *nat-less-induct*)
apply (*case-tac n* = *0*, *simp*)
apply (*drule-tac x*=*n div 2* **in** *spec*, *simp*)
apply (*simp add*: *set-decode-div-2*)
apply (*simp add*: *finite-vimage-Suc-iff*)
done

22.5.4 Proof of isomorphism

lemma *set-decode-inverse* [*simp*]: *set-encode* (*set-decode n*) = *n*
apply (*induct n rule*: *nat-less-induct*)
apply (*case-tac n* = *0*, *simp*)
apply (*drule-tac x*=*n div 2* **in** *spec*, *simp*)
apply (*simp add*: *set-decode-div-2 set-encode-vimage-Suc*)
apply (*erule div2-even-ext-nat*)
apply (*simp add*: *even-set-encode-iff*)
done

lemma *set-encode-inverse* [*simp*]: *finite A* ⟹ *set-decode* (*set-encode A*) = *A*
apply (*erule finite-induct*, *simp-all*)
apply (*simp add*: *set-decode-plus-power-2*)
done

lemma *inj-on-set-encode*: *inj-on set-encode* (*Collect finite*)
by (*rule inj-on-inverseI* [**where** *g*=*set-decode*], *simp*)

lemma *set-encode-eq*:
⟦*finite A*; *finite B*⟧ ⟹ *set-encode A* = *set-encode B* ⟷ *A* = *B*
by (*rule iffI*, *simp add*: *inj-onD* [*OF inj-on-set-encode*], *simp*)

end

23 Countable: Encoding (almost) everything into natural numbers

theory *Countable*
imports *Main Rat Nat-Bijection*
begin

23.1 The class of countable types

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

lemma *countable-classI*:
fixes *f* :: *'a* ⇒ *nat*
assumes ⋀*x y*. *f x* = *f y* ⟹ *x* = *y*
shows *OFCLASS*(*'a*, *countable-class*)
proof (*intro-classes*, *rule exI*)
show *inj f*
by (*rule injI* [*OF assms*]) *assumption*
qed

23.2 Conversion functions

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

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

lemma *inj-to-nat* [*simp*]: *inj to-nat*
by (*rule exE-some* [*OF ex-inj*]) (*simp add*: *to-nat-def*)

lemma *surj-from-nat* [*simp*]: *surj from-nat*
unfolding *from-nat-def* **by** (*simp add*: *inj-imp-surj-inv*)

lemma *to-nat-split* [*simp*]: *to-nat x* = *to-nat y* ⟷ *x* = *y*
using *injD* [*OF inj-to-nat*] **by** *auto*

lemma *from-nat-to-nat* [*simp*]:
from-nat (*to-nat x*) = *x*
by (*simp add*: *from-nat-def*)

23.3 Countable types

instance *nat* :: *countable*
by (*rule countable-classI* [*of id*]) *simp*

subclass (**in** *finite*) *countable*
proof
have *finite* (*UNIV*::'*a set*) **by** (*rule finite-UNIV*)
with *finite-conv-nat-seg-image* [*of UNIV*::'*a set*]
obtain *n* **and** *f* :: *nat* ⇒ '*a*
where *UNIV* = *f* ' {*i*. *i* < *n*} **by** *auto*
then have *surj f* **unfolding** *surj-def* **by** *auto*
then have *inj* (*inv f*) **by** (*rule surj-imp-inj-inv*)
then show ∃ *to-nat* :: '*a* ⇒ *nat*. *inj to-nat* **by** (*rule exI*[*of inj*])
qed

Pairs

instance ∗ :: (*countable*, *countable*) *countable*
by (*rule countable-classI* [*of* λ(*x*, *y*). *prod-encode* (*to-nat x*, *to-nat y*)])
(*auto simp add*: *prod-encode-eq*)

Sums

instance +:: (*countable*, *countable*) *countable*
by (*rule countable-classI* [*of* (λ*x*. *case x of Inl a* ⇒ *to-nat* (*False*, *to-nat a*)
| *Inr b* ⇒ *to-nat* (*True*, *to-nat b*))])
(*simp split*: *sum.split-asm*)

Integers

instance *int* :: *countable*
by (*rule countable-classI* [*of int-encode*])
(*simp add*: *int-encode-eq*)

Options

instance *option* :: (*countable*) *countable*
by (*rule countable-classI* [*of option-case 0* (*Suc* ∘ *to-nat*)])
(*simp split*: *option.split-asm*)

Lists

instance *list* :: (*countable*) *countable*
by (*rule countable-classI* [*of list-encode* ∘ *map to-nat*])
(*simp add*: *list-encode-eq*)

Functions

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

```
  obtain xs :: 'a list where xs: set xs = UNIV
    using finite-list [OF finite-UNIV] ..
  show ∃ to-nat::('a ⇒ 'b) ⇒ nat. inj to-nat
  proof
    show inj (λf. to-nat (map f xs))
      by (rule injI, simp add: xs expand-fun-eq)
  qed
qed
```

23.4 The Rationals are Countably Infinite

```
definition nat-to-rat-surj :: nat ⇒ rat where
nat-to-rat-surj n = (let (a,b) = prod-decode n
                     in Fract (int-decode a) (int-decode b))

lemma surj-nat-to-rat-surj: surj nat-to-rat-surj
unfolding surj-def
proof
  fix r::rat
  show ∃ n. r = nat-to-rat-surj n
  proof (cases r)
    fix i j assume [simp]: r = Fract i j and j > 0
    have r = (let m = int-encode i; n = int-encode j
              in nat-to-rat-surj(prod-encode (m,n)))
      by (simp add: Let-def nat-to-rat-surj-def)
    thus ∃ n. r = nat-to-rat-surj n by(auto simp:Let-def)
  qed
qed

lemma Rats-eq-range-nat-to-rat-surj: ℚ = range nat-to-rat-surj
by (simp add: Rats-def surj-nat-to-rat-surj surj-range)

context field-char-0
begin

lemma Rats-eq-range-of-rat-o-nat-to-rat-surj:
  ℚ = range (of-rat o nat-to-rat-surj)
using surj-nat-to-rat-surj
by (auto simp: Rats-def image-def surj-def)
   (blast intro: arg-cong[where f = of-rat])

lemma surj-of-rat-nat-to-rat-surj:
  r∈ℚ ⟹ ∃ n. r = of-rat(nat-to-rat-surj n)
by(simp add: Rats-eq-range-of-rat-o-nat-to-rat-surj image-def)

end

instance rat :: countable
proof
```

```
  show ∃ to-nat::rat ⇒ nat. inj to-nat
  proof
    have surj nat-to-rat-surj
      by (rule surj-nat-to-rat-surj)
    then show inj (inv nat-to-rat-surj)
      by (rule surj-imp-inj-inv)
  qed
qed

end
```

24 Diagonalize: A constructive version of Cantor's first diagonalization argument.

```
theory Diagonalize
imports Main
begin
```

24.1 Summation from *0* to *n*

```
definition sum :: nat ⇒ nat where
  sum n = n * Suc n div 2

lemma sum-0:
  sum 0 = 0
  unfolding sum-def by simp

lemma sum-Suc:
  sum (Suc n) = Suc n + sum n
  unfolding sum-def by simp

lemma sum2:
  2 * sum n = n * Suc n
proof −
  have 2 dvd n * Suc n
  proof (cases 2 dvd n)
    case True then show ?thesis by simp
  next
    case False then have 2 dvd Suc n by arith
    then show ?thesis by (simp del: mult-Suc-right)
  qed
  then have n * Suc n div 2 * 2 = n * Suc n
    by (rule dvd-div-mult-self [of 2::nat])
  then show ?thesis by (simp add: sum-def)
qed

lemma sum-strict-mono:
```

strict-mono sum
proof (*rule strict-monoI*)
fix *m n* :: *nat*
assume *m* < *n*
then have *m* * *Suc m* < *n* * *Suc n* **by** (*intro mult-strict-mono*) *simp-all*
then have *2* * *sum m* < *2* * *sum n* **by** (*simp add*: *sum2*)
then show *sum m* < *sum n* **by** *auto*
qed

lemma *sum-not-less-self*:
n ≤ *sum n*
proof −
have *2* * *n* ≤ *n* * *Suc n* **by** *auto*
with *sum2* **have** *2* * *n* ≤ *2* * *sum n* **by** *simp*
then show *?thesis* **by** *simp*
qed

lemma *sum-rest-aux*:
assumes *q* ≤ *n*
assumes *sum m* ≤ *sum n* + *q*
shows *m* ≤ *n*
proof (*rule ccontr*)
assume ¬ *m* ≤ *n*
then have *n* < *m* **by** *simp*
then have *m* ≥ *Suc n* **by** *simp*
then have *m* = *m* − *Suc n* + *Suc n* **by** *simp*
then have *m* = *Suc* (*n* + (*m* − *Suc n*)) **by** *simp*
then obtain *r* **where** *m* = *Suc* (*n* + *r*) **by** *auto*
with ‹*sum m* ≤ *sum n* + *q*› **have** *sum* (*Suc* (*n* + *r*)) ≤ *sum n* + *q* **by** *simp*
then have *sum* (*n* + *r*) + *Suc* (*n* + *r*) ≤ *sum n* + *q* **unfolding** *sum-Suc* **by** *simp*
with ‹*m* = *Suc* (*n* + *r*)› **have** *sum* (*n* + *r*) + *m* ≤ *sum n* + *q* **by** *simp*
have *sum n* ≤ *sum* (*n* + *r*) **unfolding** *strict-mono-less-eq* [*OF sum-strict-mono*] **by** *simp*
moreover from ‹*q* ≤ *n*› ‹*n* < *m*› **have** *q* < *m* **by** *simp*
ultimately have *sum n* + *q* < *sum* (*n* + *r*) + *m* **by** *auto*
with ‹*sum* (*n* + *r*) + *m* ≤ *sum n* + *q*› **show** *False*
by *auto*
qed

lemma *sum-rest*:
assumes *q* ≤ *n*
shows *sum m* ≤ *sum n* + *q* ⟷ *m* ≤ *n*
using *assms* **apply** (*auto intro*: *sum-rest-aux*)
apply (*simp add*: *strict-mono-less-eq* [*OF sum-strict-mono*, *symmetric*, *of m n*])
done

24.2 Diagonalization: an injective embedding of two *nat*s to one *nat*

definition *diagonalize* :: *nat* ⇒ *nat* ⇒ *nat* **where**
diagonalize m n = *sum* (*m* + *n*) + *m*

lemma *diagonalize-inject*:
assumes *diagonalize a b* = *diagonalize c d*
shows *a* = *c* **and** *b* = *d*
proof −
from *assms* **have** *diageq*: *sum* (*a* + *b*) + *a* = *sum* (*c* + *d*) + *c*
by (*simp add*: *diagonalize-def*)
have *a* + *b* = *c* + *d* ∨ *a* + *b* ≥ *Suc* (*c* + *d*) ∨ *c* + *d* ≥ *Suc* (*a* + *b*) **by** *arith*
then have *a* = *c* ∧ *b* = *d*
proof (*elim disjE*)
assume *sumeq*: *a* + *b* = *c* + *d*
then have *a* = *c* **using** *diageq* **by** *auto*
moreover from *sumeq this* **have** *b* = *d* **by** *auto*
ultimately show *?thesis* **..**
next
assume *a* + *b* ≥ *Suc* (*c* + *d*)
with *strict-mono-less-eq* [*OF sum-strict-mono*]
have *sum* (*a* + *b*) ≥ *sum* (*Suc* (*c* + *d*)) **by** *simp*
with *diageq* **show** *?thesis* **by** (*simp add*: *sum-Suc*)
next
assume *c* + *d* ≥ *Suc* (*a* + *b*)
with *strict-mono-less-eq* [*OF sum-strict-mono*]
have *sum* (*c* + *d*) ≥ *sum* (*Suc* (*a* + *b*)) **by** *simp*
with *diageq* **show** *?thesis* **by** (*simp add*: *sum-Suc*)
qed
then show *a* = *c* **and** *b* = *d* **by** *auto*
qed

24.3 The reverse diagonalization: reconstruction a pair of *nat*s from one *nat*

The inverse of the *sum* function

definition *tupelize* :: *nat* ⇒ *nat* × *nat* **where**
tupelize q = (*let d* = *Max* {*d*. *sum d* ≤ *q*}; *m* = *q* − *sum d*
in (*m*, *d* − *m*))

lemma *tupelize-diagonalize*:
tupelize (*diagonalize m n*) = (*m*, *n*)
proof −
from *sum-rest*
have ⋀*r*. *sum r* ≤ *sum* (*m* + *n*) + *m* ⟷ *r* ≤ *m* + *n* **by** *simp*
then have *Max* {*d*. *sum d* ≤ (*sum* (*m* + *n*) + *m*)} = *m* + *n*
by (*auto intro*: *Max-eqI*)
then show *?thesis*

by (*simp add*: *tupelize-def diagonalize-def*)
qed

lemma *snd-tupelize*:
snd (*tupelize n*) ≤ *n*
proof −
have *sum 0* ≤ *n* **by** (*simp add*: *sum-0*)
then have *Max* {*m* :: *nat*. *sum m* ≤ *n*} ≤ *Max* {*m* :: *nat*. *m* ≤ *n*}
by (*intro Max-mono* [*of* {*m*. *sum m* ≤ *n*} {*m*. *m* ≤ *n*}])
(*auto intro*: *Max-mono order-trans sum-not-less-self*)
also have *Max* {*m* :: *nat*. *m* ≤ *n*} ≤ *n*
by (*subst Max-le-iff*) *auto*
finally have *Max* {*m*. *sum m* ≤ *n*} ≤ *n* **.**
then show *?thesis* **by** (*simp add*: *tupelize-def Let-def*)
qed

end

25 More-List: Operations on lists beyond the standard List theory

theory *More-List*
imports *Main*
begin

hide-const (**open**) *Finite-Set.fold*

Repairing code generator setup

declare (**in** *lattice*) *Inf-fin-set-fold* [*code-unfold del*]
declare (**in** *lattice*) *Sup-fin-set-fold* [*code-unfold del*]
declare (**in** *linorder*) *Min-fin-set-fold* [*code-unfold del*]
declare (**in** *linorder*) *Max-fin-set-fold* [*code-unfold del*]
declare (**in** *complete-lattice*) *Inf-set-fold* [*code-unfold del*]
declare (**in** *complete-lattice*) *Sup-set-fold* [*code-unfold del*]
declare *rev-foldl-cons* [*code del*]

Fold combinator with canonical argument order

primrec *fold* :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*a list* ⇒ '*b* ⇒ '*b* **where**
fold f [] = *id*
| *fold f* (*x* # *xs*) = *fold f xs* ∘ *f x*

lemma *foldl-fold*:
foldl f s xs = *fold* (λ*x s*. *f s x*) *xs s*
by (*induct xs arbitrary*: *s*) *simp-all*

lemma *foldr-fold-rev*:
foldr f xs = *fold f* (*rev xs*)

by (*simp add*: *foldr-foldl foldl-fold expand-fun-eq*)

lemma *fold-rev-conv* [*code-unfold*]:
fold f (*rev xs*) = *foldr f xs*
by (*simp add*: *foldr-fold-rev*)

lemma *fold-cong* [*fundef-cong*, *recdef-cong*]:
$a = b \Longrightarrow xs = ys \Longrightarrow (\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x)$
$\Longrightarrow$ *fold f xs a* = *fold g ys b*
by (*induct ys arbitrary*: *a b xs*) *simp-all*

lemma *fold-id*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = id$
shows *fold f xs* = *id*
using *assms* **by** (*induct xs*) *simp-all*

lemma *fold-apply*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow h \circ g\ x = f\ x \circ h$
shows $h \circ$ *fold g xs* = *fold f xs* $\circ\ h$
using *assms* **by** (*induct xs*) (*simp-all add*: *expand-fun-eq*)

lemma *fold-invariant*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow Q\ x$ **and** $P\ s$
and $\bigwedge x\ s.\ Q\ x \Longrightarrow P\ s \Longrightarrow P\ (f\ x\ s)$
shows P (*fold f xs s*)
using *assms* **by** (*induct xs arbitrary*: *s*) *simp-all*

lemma *fold-weak-invariant*:
assumes $P\ s$
and $\bigwedge s\ x.\ x \in set\ xs \Longrightarrow P\ s \Longrightarrow P\ (f\ x\ s)$
shows P (*fold f xs s*)
using *assms* **by** (*induct xs arbitrary*: *s*) *simp-all*

lemma *fold-append* [*simp*]:
fold f (*xs* @ *ys*) = *fold f ys* $\circ$ *fold f xs*
by (*induct xs*) *simp-all*

lemma *fold-map* [*code-unfold*]:
fold g (*map f xs*) = *fold* (*g o f*) *xs*
by (*induct xs*) *simp-all*

lemma *fold-rev*:
assumes $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow f\ y \circ f\ x = f\ x \circ f\ y$
shows *fold f* (*rev xs*) = *fold f xs*
using *assms* **by** (*induct xs*) (*simp-all del*: *o-apply add*: *fold-apply*)

lemma *foldr-fold*:
assumes $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow f\ y \circ f\ x = f\ x \circ f\ y$
shows *foldr f xs* = *fold f xs*

using *assms* **unfolding** *foldr-fold-rev* **by** (*rule fold-rev*)

lemma *fold-Cons-rev*:
fold Cons xs = *append* (*rev xs*)
by (*induct xs*) *simp-all*

lemma *rev-conv-fold* [*code*]:
rev xs = *fold Cons xs* []
by (*simp add*: *fold-Cons-rev*)

lemma *fold-append-concat-rev*:
fold append xss = *append* (*concat* (*rev xss*))
by (*induct xss*) *simp-all*

lemma *concat-conv-foldr* [*code*]:
concat xss = *foldr append xss* []
by (*simp add*: *fold-append-concat-rev foldr-fold-rev*)

lemma *fold-plus-listsum-rev*:
fold plus xs = *plus* (*listsum* (*rev xs*))
by (*induct xs*) (*simp-all add*: *add.assoc*)

lemma *listsum-conv-foldr* [*code*]:
listsum xs = *foldr plus xs 0*
by (*fact listsum-foldr*)

lemma *sort-key-conv-fold*:
assumes *inj-on f* (*set xs*)
shows *sort-key f xs* = *fold* (*insort-key f*) *xs* []
proof −
have *fold* (*insort-key f*) (*rev xs*) = *fold* (*insort-key f*) *xs*
proof (*rule fold-rev*, *rule ext*)
fix *zs*
fix *x y*
assume *x* ∈ *set xs y* ∈ *set xs*
with *assms* **have** ∗: *f y* = *f x* ⟹ *y* = *x* **by** (*auto dest*: *inj-onD*)
show (*insort-key f y* ∘ *insort-key f x*) *zs* = (*insort-key f x* ∘ *insort-key f y*) *zs*
by (*induct zs*) (*auto dest*: ∗)
qed
then show *?thesis* **by** (*simp add*: *sort-key-def foldr-fold-rev*)
qed

lemma *sort-conv-fold*:
sort xs = *fold insort xs* []
by (*rule sort-key-conv-fold*) *simp*

Finite-Set.fold and *fold*

lemma (**in** *fun-left-comm*) *fold-set-remdups*:
Finite-Set.fold f y (*set xs*) = *fold f* (*remdups xs*) *y*

by (*rule sym*, *induct xs arbitrary*: *y*) (*simp-all add*: *fold-fun-comm insert-absorb*)

lemma (**in** *fun-left-comm-idem*) *fold-set*:
Finite-Set.fold f y (*set xs*) = *fold f xs y*
by (*rule sym*, *induct xs arbitrary*: *y*) (*simp-all add*: *fold-fun-comm*)

lemma (**in** *ab-semigroup-idem-mult*) *fold1-set*:
assumes *xs* ≠ []
shows *Finite-Set.fold1 times* (*set xs*) = *fold times* (*tl xs*) (*hd xs*)
proof −
interpret *fun-left-comm-idem times* **by** (*fact fun-left-comm-idem*)
from *assms* **obtain** *y ys* **where** *xs*: *xs* = *y* # *ys*
by (*cases xs*) *auto*
show *?thesis*
proof (*cases set ys* = {})
case *True* **with** *xs* **show** *?thesis* **by** *simp*
next
case *False*
then have *fold1 times* (*insert y* (*set ys*)) = *Finite-Set.fold times y* (*set ys*)
by (*simp only*: *finite-set fold1-eq-fold-idem*)
with *xs* **show** *?thesis* **by** (*simp add*: *fold-set mult-commute*)
qed
qed

lemma (**in** *lattice*) *Inf-fin-set-fold*:
Inf-fin (*set* (*x* # *xs*)) = *fold inf xs x*
proof −
interpret *ab-semigroup-idem-mult inf* :: *'a* ⇒ *'a* ⇒ *'a*
by (*fact ab-semigroup-idem-mult-inf*)
show *?thesis*
by (*simp add*: *Inf-fin-def fold1-set del*: *set.simps*)
qed

lemma (**in** *lattice*) *Inf-fin-set-foldr* [*code-unfold*]:
Inf-fin (*set* (*x* # *xs*)) = *foldr inf xs x*
by (*simp add*: *Inf-fin-set-fold ac-simps foldr-fold expand-fun-eq del*: *set.simps*)

lemma (**in** *lattice*) *Sup-fin-set-fold*:
Sup-fin (*set* (*x* # *xs*)) = *fold sup xs x*
proof −
interpret *ab-semigroup-idem-mult sup* :: *'a* ⇒ *'a* ⇒ *'a*
by (*fact ab-semigroup-idem-mult-sup*)
show *?thesis*
by (*simp add*: *Sup-fin-def fold1-set del*: *set.simps*)
qed

lemma (**in** *lattice*) *Sup-fin-set-foldr* [*code-unfold*]:
Sup-fin (*set* (*x* # *xs*)) = *foldr sup xs x*
by (*simp add*: *Sup-fin-set-fold ac-simps foldr-fold expand-fun-eq del*: *set.simps*)

lemma (**in** *linorder*) *Min-fin-set-fold*:
Min (*set* (*x* # *xs*)) = *fold min xs x*
proof −
interpret *ab-semigroup-idem-mult min* :: *'a* ⇒ *'a* ⇒ *'a*
by (*fact ab-semigroup-idem-mult-min*)
show *?thesis*
by (*simp add*: *Min-def fold1-set del*: *set.simps*)
qed

lemma (**in** *linorder*) *Min-fin-set-foldr* [*code-unfold*]:
Min (*set* (*x* # *xs*)) = *foldr min xs x*
by (*simp add*: *Min-fin-set-fold ac-simps foldr-fold expand-fun-eq del*: *set.simps*)

lemma (**in** *linorder*) *Max-fin-set-fold*:
Max (*set* (*x* # *xs*)) = *fold max xs x*
proof −
interpret *ab-semigroup-idem-mult max* :: *'a* ⇒ *'a* ⇒ *'a*
by (*fact ab-semigroup-idem-mult-max*)
show *?thesis*
by (*simp add*: *Max-def fold1-set del*: *set.simps*)
qed

lemma (**in** *linorder*) *Max-fin-set-foldr* [*code-unfold*]:
Max (*set* (*x* # *xs*)) = *foldr max xs x*
by (*simp add*: *Max-fin-set-fold ac-simps foldr-fold expand-fun-eq del*: *set.simps*)

lemma (**in** *complete-lattice*) *Inf-set-fold*:
Inf (*set xs*) = *fold inf xs top*
proof −
interpret *fun-left-comm-idem inf* :: *'a* ⇒ *'a* ⇒ *'a*
by (*fact fun-left-comm-idem-inf*)
show *?thesis* **by** (*simp add*: *Inf-fold-inf fold-set inf-commute*)
qed

lemma (**in** *complete-lattice*) *Inf-set-foldr* [*code-unfold*]:
Inf (*set xs*) = *foldr inf xs top*
by (*simp add*: *Inf-set-fold ac-simps foldr-fold expand-fun-eq*)

lemma (**in** *complete-lattice*) *Sup-set-fold*:
Sup (*set xs*) = *fold sup xs bot*
proof −
interpret *fun-left-comm-idem sup* :: *'a* ⇒ *'a* ⇒ *'a*
by (*fact fun-left-comm-idem-sup*)
show *?thesis* **by** (*simp add*: *Sup-fold-sup fold-set sup-commute*)
qed

lemma (**in** *complete-lattice*) *Sup-set-foldr* [*code-unfold*]:
Sup (*set xs*) = *foldr sup xs bot*

by (*simp add*: *Sup-set-fold ac-simps foldr-fold expand-fun-eq*)

lemma (**in** *complete-lattice*) *INFI-set-fold*:
INFI (*set xs*) *f* = *fold* (*inf* ∘ *f*) *xs top*
unfolding *INFI-def set-map* [*symmetric*] *Inf-set-fold fold-map* **..**

lemma (**in** *complete-lattice*) *SUPR-set-fold*:
SUPR (*set xs*) *f* = *fold* (*sup* ∘ *f*) *xs bot*
unfolding *SUPR-def set-map* [*symmetric*] *Sup-set-fold fold-map* **..**

nth-map

definition *nth-map* :: *nat* ⇒ (*'a* ⇒ *'a*) ⇒ *'a list* ⇒ *'a list* **where**
nth-map n f xs = (*if n* < *length xs then*
take n xs @ [*f* (*xs* ! *n*)] @ *drop* (*Suc n*) *xs*
else xs)

lemma *nth-map-id*:
n ≥ *length xs* ⟹ *nth-map n f xs* = *xs*
by (*simp add*: *nth-map-def*)

lemma *nth-map-unfold*:
n < *length xs* ⟹ *nth-map n f xs* = *take n xs* @ [*f* (*xs* ! *n*)] @ *drop* (*Suc n*) *xs*
by (*simp add*: *nth-map-def*)

lemma *nth-map-Nil* [*simp*]:
nth-map n f [] = []
by (*simp add*: *nth-map-def*)

lemma *nth-map-zero* [*simp*]:
nth-map 0 f (*x* # *xs*) = *f x* # *xs*
by (*simp add*: *nth-map-def*)

lemma *nth-map-Suc* [*simp*]:
nth-map (*Suc n*) *f* (*x* # *xs*) = *x* # *nth-map n f xs*
by (*simp add*: *nth-map-def*)

end

26 More-Set: Relating (finite) sets and lists

theory *More-Set*
imports *Main More-List*
begin

26.1 Various additional set functions

definition *is-empty* :: *'a set* ⇒ *bool* **where**
is-empty A ⟷ *A* = {}

definition *remove* :: *'a* ⇒ *'a set* ⇒ *'a set* **where**
remove x A = *A* − {*x*}

lemma *fun-left-comm-idem-remove*:
fun-left-comm-idem remove
proof −
have *rem*: *remove* = (λ*x A*. *A* − {*x*}) **by** (*simp add*: *expand-fun-eq remove-def*)
show *?thesis* **by** (*simp only*: *fun-left-comm-idem-remove rem*)
qed

lemma *minus-fold-remove*:
assumes *finite A*
shows *B* − *A* = *Finite-Set.fold remove B A*
proof −
have *rem*: *remove* = (λ*x A*. *A* − {*x*}) **by** (*simp add*: *expand-fun-eq remove-def*)
show *?thesis* **by** (*simp only*: *rem assms minus-fold-remove*)
qed

definition *project* :: (*'a* ⇒ *bool*) ⇒ *'a set* ⇒ *'a set* **where**
project P A = {*a*∈*A*. *P a*}

26.2 Basic set operations

lemma *is-empty-set*:
is-empty (*set xs*) ⟷ *null xs*
by (*simp add*: *is-empty-def null-empty*)

lemma *ball-set*:
(∀ *x*∈*set xs*. *P x*) ⟷ *list-all P xs*
by (*rule list-ball-code*)

lemma *bex-set*:
(∃ *x*∈*set xs*. *P x*) ⟷ *list-ex P xs*
by (*rule list-bex-code*)

lemma *empty-set*:
{} = *set* []
by *simp*

lemma *insert-set-compl*:
insert x (− *set xs*) = − *set* (*removeAll x xs*)
by *auto*

lemma *remove-set-compl*:
remove x (− *set xs*) = − *set* (*List.insert x xs*)
by (*auto simp del*: *mem-def simp add*: *remove-def List.insert-def*)

lemma *image-set*:

image f (set xs) = set (map f xs)
by *simp*

lemma *project-set*:
project P (set xs) = set (filter P xs)
by (*auto simp add*: *project-def*)

26.3 Functorial set operations

lemma *union-set*:
set xs ∪ A = fold Set.insert xs A
proof −
interpret *fun-left-comm-idem Set.insert*
by (*fact fun-left-comm-idem-insert*)
show *?thesis* **by** (*simp add*: *union-fold-insert fold-set*)
qed

lemma *union-set-foldr*:
set xs ∪ A = foldr Set.insert xs A
proof −
have ⋀*x y* :: *'a*. *insert y ∘ insert x = insert x ∘ insert y*
by (*auto intro*: *ext*)
then show *?thesis* **by** (*simp add*: *union-set foldr-fold*)
qed

lemma *minus-set*:
A − set xs = fold remove xs A
proof −
interpret *fun-left-comm-idem remove*
by (*fact fun-left-comm-idem-remove*)
show *?thesis*
by (*simp add*: *minus-fold-remove* [*of - A*] *fold-set*)
qed

lemma *minus-set-foldr*:
A − set xs = foldr remove xs A
proof −
have ⋀*x y* :: *'a*. *remove y ∘ remove x = remove x ∘ remove y*
by (*auto simp add*: *remove-def intro*: *ext*)
then show *?thesis* **by** (*simp add*: *minus-set foldr-fold*)
qed

26.4 Derived set operations

lemma *member*:
a ∈ A ⟷ (∃ *x*∈*A*. *a = x*)
by *simp*

lemma *subset-eq*:
A ⊆ B ⟷ (∀ *x*∈*A*. *x ∈ B*)

by (*fact subset-eq*)

lemma *subset*:
$A \subset B \longleftrightarrow A \subseteq B \wedge \neg\ B \subseteq A$
by (*fact less-le-not-le*)

lemma *set-eq*:
$A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
by (*fact eq-iff*)

lemma *inter*:
$A \cap B = project\ (\lambda x.\ x \in A)\ B$
by (*auto simp add*: *project-def*)

26.5 Various lemmas

lemma *not-set-compl*:
$Not \circ set\ xs = -\ set\ xs$
by (*simp add*: *fun-Compl-def bool-Compl-def comp-def expand-fun-eq*)

end

27 Fset: Executable finite sets

theory *Fset*
imports *More-Set More-List*
begin

declare *mem-def* [*simp*]

27.1 Lifting

datatype *'a fset = Fset 'a set*

primrec *member* :: *'a fset* ⇒ *'a set* **where**
member (*Fset A*) = *A*

lemma *member-inject* [*simp*]:
member A = *member B* ⟹ *A* = *B*
by (*cases A*, *cases B*) *simp*

lemma *Fset-member* [*simp*]:
Fset (*member A*) = *A*
by (*cases A*) *simp*

definition *Set* :: *'a list* ⇒ *'a fset* **where**
Set xs = *Fset* (*set xs*)

lemma *member-Set* [*simp*]:
member (*Set xs*) = *set xs*
by (*simp add*: *Set-def*)

definition *Coset* :: *'a list* ⇒ *'a fset* **where**
Coset xs = *Fset* (− *set xs*)

lemma *member-Coset* [*simp*]:
member (*Coset xs*) = − *set xs*
by (*simp add*: *Coset-def*)

code-datatype *Set Coset*

lemma *member-code* [*code*]:
member (*Set xs*) = *List.member xs*
member (*Coset xs*) = *Not* ∘ *List.member xs*
by (*simp-all add*: *expand-fun-eq mem-iff fun-Compl-def bool-Compl-def*)

lemma *member-image-UNIV* [*simp*]:
member ‘ *UNIV* = *UNIV*
proof −
have ⋀*A* :: *'a set*. ∃ *B* :: *'a fset*. *A* = *member B*
proof
fix *A* :: *'a set*
show *A* = *member* (*Fset A*) **by** *simp*
qed
then show *?thesis* **by** (*simp add*: *image-def*)
qed

27.2 Lattice instantiation

instantiation *fset* :: (*type*) *boolean-algebra*
begin

definition *less-eq-fset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* **where**
[*simp*]: *A* ≤ *B* ⟷ *member A* ⊆ *member B*

definition *less-fset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* **where**
[*simp*]: *A* < *B* ⟷ *member A* ⊂ *member B*

definition *inf-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **where**
[*simp*]: *inf A B* = *Fset* (*member A* ∩ *member B*)

definition *sup-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **where**
[*simp*]: *sup A B* = *Fset* (*member A* ∪ *member B*)

definition *bot-fset* :: *'a fset* **where**
[*simp*]: *bot* = *Fset* {}

definition *top-fset* :: *'a fset* **where**
[*simp*]: *top* = *Fset UNIV*

definition *uminus-fset* :: *'a fset* ⇒ *'a fset* **where**
[*simp*]: − *A* = *Fset* (− (*member A*))

definition *minus-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **where**
[*simp*]: *A* − *B* = *Fset* (*member A* − *member B*)

instance proof
qed *auto*

end

instantiation *fset* :: (*type*) *complete-lattice*
begin

definition *Inf-fset* :: *'a fset set* ⇒ *'a fset* **where**
[*simp*, *code del*]: *Inf-fset As* = *Fset* (*Inf* (*image member As*))

definition *Sup-fset* :: *'a fset set* ⇒ *'a fset* **where**
[*simp*, *code del*]: *Sup-fset As* = *Fset* (*Sup* (*image member As*))

instance proof
qed (*auto simp add*: *le-fun-def le-bool-def*)

end

27.3 Basic operations

definition *is-empty* :: *'a fset* ⇒ *bool* **where**
[*simp*]: *is-empty A* ⟷ *More-Set.is-empty* (*member A*)

lemma *is-empty-Set* [*code*]:
is-empty (*Set xs*) ⟷ *null xs*
by (*simp add*: *is-empty-set*)

lemma *empty-Set* [*code*]:
bot = *Set* []
by *simp*

lemma *UNIV-Set* [*code*]:
top = *Coset* []
by *simp*

definition *insert* :: *'a* ⇒ *'a fset* ⇒ *'a fset* **where**
[*simp*]: *insert x A* = *Fset* (*Set.insert x* (*member A*))

lemma *insert-Set* [*code*]:

insert x (Set xs) = Set (List.insert x xs)
insert x (Coset xs) = Coset (removeAll x xs)
by (*simp-all add*: *Set-def Coset-def*)

definition *remove* :: *'a ⇒ 'a fset ⇒ 'a fset* **where**
[*simp*]: *remove x A = Fset (More-Set.remove x (member A))*

lemma *remove-Set* [*code*]:
remove x (Set xs) = Set (removeAll x xs)
remove x (Coset xs) = Coset (List.insert x xs)
by (*simp-all add*: *Set-def Coset-def remove-set-compl*)
(*simp add*: *More-Set.remove-def*)

definition *map* :: *('a ⇒ 'b) ⇒ 'a fset ⇒ 'b fset* **where**
[*simp*]: *map f A = Fset (image f (member A))*

lemma *map-Set* [*code*]:
map f (Set xs) = Set (remdups (List.map f xs))
by (*simp add*: *Set-def*)

definition *filter* :: *('a ⇒ bool) ⇒ 'a fset ⇒ 'a fset* **where**
[*simp*]: *filter P A = Fset (More-Set.project P (member A))*

lemma *filter-Set* [*code*]:
filter P (Set xs) = Set (List.filter P xs)
by (*simp add*: *Set-def project-set*)

definition *forall* :: *('a ⇒ bool) ⇒ 'a fset ⇒ bool* **where**
[*simp*]: *forall P A ⟷ Ball (member A) P*

lemma *forall-Set* [*code*]:
forall P (Set xs) ⟷ list-all P xs
by (*simp add*: *Set-def ball-set*)

definition *exists* :: *('a ⇒ bool) ⇒ 'a fset ⇒ bool* **where**
[*simp*]: *exists P A ⟷ Bex (member A) P*

lemma *exists-Set* [*code*]:
exists P (Set xs) ⟷ list-ex P xs
by (*simp add*: *Set-def bex-set*)

definition *card* :: *'a fset ⇒ nat* **where**
[*simp*]: *card A = Finite-Set.card (member A)*

lemma *card-Set* [*code*]:
card (Set xs) = length (remdups xs)
proof −
have *Finite-Set.card (set (remdups xs)) = length (remdups xs)*
by (*rule distinct-card*) *simp*

then show *?thesis* **by** (*simp add*: *Set-def*)
qed

lemma *compl-Set* [*simp*, *code*]:
– *Set xs* = *Coset xs*
by (*simp add*: *Set-def Coset-def*)

lemma *compl-Coset* [*simp*, *code*]:
– *Coset xs* = *Set xs*
by (*simp add*: *Set-def Coset-def*)

27.4 Derived operations

lemma *subfset-eq-forall* [*code*]:
A ≤ *B* ⟷ *forall* (*member B*) *A*
by (*simp add*: *subset-eq*)

lemma *subfset-subfset-eq* [*code*]:
A < *B* ⟷ *A* ≤ *B* ∧ ¬ *B* ≤ (*A* :: *'a fset*)
by (*fact less-le-not-le*)

lemma *eq-fset-subfset-eq* [*code*]:
eq-class.eq A B ⟷ *A* ≤ *B* ∧ *B* ≤ (*A* :: *'a fset*)
by (*cases A*, *cases B*) (*simp add*: *eq set-eq*)

27.5 Functorial operations

lemma *inter-project* [*code*]:
inf A (*Set xs*) = *Set* (*List.filter* (*member A*) *xs*)
inf A (*Coset xs*) = *foldr remove xs A*
proof –
show *inf A* (*Set xs*) = *Set* (*List.filter* (*member A*) *xs*)
by (*simp add*: *inter project-def Set-def*)
have ∗: ⋀*x*::*'a*. *remove* = (λ*x*. *Fset* ∘ *More-Set.remove x* ∘ *member*)
by (*simp add*: *expand-fun-eq*)
have *member* ∘ *fold* (λ*x*. *Fset* ∘ *More-Set.remove x* ∘ *member*) *xs* =
fold More-Set.remove xs ∘ *member*
by (*rule fold-apply*) (*simp add*: *expand-fun-eq*)
then have *fold More-Set.remove xs* (*member A*) =
member (*fold* (λ*x*. *Fset* ∘ *More-Set.remove x* ∘ *member*) *xs A*)
by (*simp add*: *expand-fun-eq*)
then have *inf A* (*Coset xs*) = *fold remove xs A*
by (*simp add*: *Diff-eq* [*symmetric*] *minus-set* ∗)
moreover have ⋀*x y* :: *'a*. *Fset.remove y* ∘ *Fset.remove x* = *Fset.remove x* ∘ *Fset.remove y*
by (*auto simp add*: *More-Set.remove-def* ∗ *intro*: *ext*)
ultimately show *inf A* (*Coset xs*) = *foldr remove xs A*
by (*simp add*: *foldr-fold*)
qed

lemma *subtract-remove* [*code*]:
A − Set xs = foldr remove xs A
A − Coset xs = Set (List.filter (member A) xs)
by (*simp-all only*: *diff-eq compl-Set compl-Coset inter-project*)

lemma *union-insert* [*code*]:
sup (Set xs) A = foldr insert xs A
sup (Coset xs) A = Coset (List.filter (Not ∘ member A) xs)
proof −
have ∗: $\bigwedge x$::'*a*. *insert = (λx. Fset ∘ Set.insert x ∘ member)*
by (*simp add*: *expand-fun-eq*)
have *member ∘ fold (λx. Fset ∘ Set.insert x ∘ member) xs =*
fold Set.insert xs ∘ member
by (*rule fold-apply*) (*simp add*: *expand-fun-eq*)
then have *fold Set.insert xs (member A) =*
member (fold (λx. Fset ∘ Set.insert x ∘ member) xs A)
by (*simp add*: *expand-fun-eq*)
then have *sup (Set xs) A = fold insert xs A*
by (*simp add*: *union-set* ∗)
moreover have $\bigwedge x\ y$ :: '*a*. *Fset.insert y ∘ Fset.insert x = Fset.insert x ∘ Fset.insert y*
by (*auto simp add*: ∗ *intro*: *ext*)
ultimately show *sup (Set xs) A = foldr insert xs A*
by (*simp add*: *foldr-fold*)
show *sup (Coset xs) A = Coset (List.filter (Not ∘ member A) xs)*
by (*auto simp add*: *Coset-def*)
qed

context *complete-lattice*
begin

definition *Infimum* :: '*a fset* ⇒ '*a* **where**
[*simp*]: *Infimum A = Inf (member A)*

lemma *Infimum-inf* [*code*]:
Infimum (Set As) = foldr inf As top
Infimum (Coset []*) = bot*
by (*simp-all add*: *Inf-set-foldr Inf-UNIV*)

definition *Supremum* :: '*a fset* ⇒ '*a* **where**
[*simp*]: *Supremum A = Sup (member A)*

lemma *Supremum-sup* [*code*]:
Supremum (Set As) = foldr sup As bot
Supremum (Coset []*) = top*
by (*simp-all add*: *Sup-set-foldr Sup-UNIV*)

end

27.6 Misc operations

lemma *size-fset* [*code*]:
fset-size f A = 0
size A = 0
by (*cases A*, *simp*) (*cases A*, *simp*)

lemma *fset-case-code* [*code*]:
fset-case f A = f (*member A*)
by (*cases A*) *simp*

lemma *fset-rec-code* [*code*]:
fset-rec f A = f (*member A*)
by (*cases A*) *simp*

27.7 Simplified simprules

lemma *is-empty-simp* [*simp*]:
is-empty A $\longleftrightarrow$ *member A* = {}
by (*simp add*: *More-Set.is-empty-def*)
declare *is-empty-def* [*simp del*]

lemma *remove-simp* [*simp*]:
remove x A = Fset (*member A* − {*x*})
by (*simp add*: *More-Set.remove-def*)
declare *remove-def* [*simp del*]

lemma *filter-simp* [*simp*]:
filter P A = Fset {*x* ∈ *member A*. *P x*}
by (*simp add*: *More-Set.project-def*)
declare *filter-def* [*simp del*]

declare *mem-def* [*simp del*]

hide-const (**open**) *is-empty insert remove map filter forall exists card Inter Union*

end

28 Dlist: Lists with elements distinct as canonical example for datatype invariants

theory *Dlist*
imports *Main More-List Fset*
begin

29 The type of distinct lists

typedef (**open**) *'a dlist* = {*xs*::*'a list*. *distinct xs*}
  morphisms *list-of-dlist Abs-dlist*
proof
  show [] ∈ *?dlist* **by** *simp*
qed

lemma *dlist-ext*:
  assumes *list-of-dlist xs* = *list-of-dlist ys*
  shows *xs* = *ys*
  using *assms* **by** (*simp add*: *list-of-dlist-inject*)

Formal, totalized constructor for *'a dlist*:

definition *Dlist* :: *'a list* ⇒ *'a dlist* **where**
  [*code del*]: *Dlist xs* = *Abs-dlist* (*remdups xs*)

lemma *distinct-list-of-dlist* [*simp*]:
  distinct (*list-of-dlist dxs*)
  using *list-of-dlist* [*of dxs*] **by** *simp*

lemma *list-of-dlist-Dlist* [*simp*]:
  list-of-dlist (*Dlist xs*) = *remdups xs*
  by (*simp add*: *Dlist-def Abs-dlist-inverse*)

lemma *Dlist-list-of-dlist* [*simp*, *code abstype*]:
  Dlist (*list-of-dlist dxs*) = *dxs*
  by (*simp add*: *Dlist-def list-of-dlist-inverse distinct-remdups-id*)

Fundamental operations:

definition *empty* :: *'a dlist* **where**
  empty = *Dlist* []

definition *insert* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
  insert x dxs = *Dlist* (*List.insert x* (*list-of-dlist dxs*))

definition *remove* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
  remove x dxs = *Dlist* (*remove1 x* (*list-of-dlist dxs*))

definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b dlist* **where**
  map f dxs = *Dlist* (*remdups* (*List.map f* (*list-of-dlist dxs*)))

definition *filter* :: (*'a* ⇒ *bool*) ⇒ *'a dlist* ⇒ *'a dlist* **where**
  filter P dxs = *Dlist* (*List.filter P* (*list-of-dlist dxs*))

Derived operations:

definition *null* :: *'a dlist* ⇒ *bool* **where**
  null dxs = *List.null* (*list-of-dlist dxs*)

definition *member* :: *'a dlist* ⇒ *'a* ⇒ *bool* **where**
member dxs = *List.member* (*list-of-dlist dxs*)

definition *length* :: *'a dlist* ⇒ *nat* **where**
length dxs = *List.length* (*list-of-dlist dxs*)

definition *fold* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b* ⇒ *'b* **where**
fold f dxs = *More-List.fold f* (*list-of-dlist dxs*)

definition *foldr* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b* ⇒ *'b* **where**
foldr f dxs = *List.foldr f* (*list-of-dlist dxs*)

30 Executable version obeying invariant

lemma *list-of-dlist-empty* [*simp*, *code abstract*]:
list-of-dlist empty = []
by (*simp add*: *empty-def*)

lemma *list-of-dlist-insert* [*simp*, *code abstract*]:
list-of-dlist (*insert x dxs*) = *List.insert x* (*list-of-dlist dxs*)
by (*simp add*: *insert-def*)

lemma *list-of-dlist-remove* [*simp*, *code abstract*]:
list-of-dlist (*remove x dxs*) = *remove1 x* (*list-of-dlist dxs*)
by (*simp add*: *remove-def*)

lemma *list-of-dlist-map* [*simp*, *code abstract*]:
list-of-dlist (*map f dxs*) = *remdups* (*List.map f* (*list-of-dlist dxs*))
by (*simp add*: *map-def*)

lemma *list-of-dlist-filter* [*simp*, *code abstract*]:
list-of-dlist (*filter P dxs*) = *List.filter P* (*list-of-dlist dxs*)
by (*simp add*: *filter-def*)

Explicit executable conversion

definition *dlist-of-list* [*simp*]:
dlist-of-list = *Dlist*

lemma [*code abstract*]:
list-of-dlist (*dlist-of-list xs*) = *remdups xs*
by *simp*

31 Induction principle and case distinction

lemma *dlist-induct* [*case-names empty insert*, *induct type*: *dlist*]:
assumes *empty*: *P empty*
assumes *insrt*: ⋀*x dxs*. ¬ *member dxs x* ⟹ *P dxs* ⟹ *P* (*insert x dxs*)
shows *P dxs*

proof (*cases dxs*)
case (*Abs-dlist xs*)
then have *distinct xs* **and** *dxs*: *dxs* = *Dlist xs* **by** (*simp-all add*: *Dlist-def distinct-remdups-id*)
from ‹*distinct xs*› **have** *P* (*Dlist xs*)
proof (*induct xs rule*: *distinct-induct*)
case *Nil* **from** *empty* **show** *?case* **by** (*simp add*: *empty-def*)
next
case (*insert x xs*)
then have ¬ *member* (*Dlist xs*) *x* **and** *P* (*Dlist xs*)
by (*simp-all add*: *member-def mem-iff*)
with *insrt* **have** *P* (*insert x* (*Dlist xs*)) **.**
with *insert* **show** *?case* **by** (*simp add*: *insert-def distinct-remdups-id*)
qed
with *dxs* **show** *P dxs* **by** *simp*
qed

lemma *dlist-case* [*case-names empty insert*, *cases type*: *dlist*]:
assumes *empty*: *dxs* = *empty* ⟹ *P*
assumes *insert*: ⋀*x dys*. ¬ *member dys x* ⟹ *dxs* = *insert x dys* ⟹ *P*
shows *P*
proof (*cases dxs*)
case (*Abs-dlist xs*)
then have *dxs*: *dxs* = *Dlist xs* **and** *distinct*: *distinct xs*
by (*simp-all add*: *Dlist-def distinct-remdups-id*)
show *P* **proof** (*cases xs*)
case *Nil* **with** *dxs* **have** *dxs* = *empty* **by** (*simp add*: *empty-def*)
with *empty* **show** *P* **.**
next
case (*Cons x xs*)
with *dxs distinct* **have** ¬ *member* (*Dlist xs*) *x*
and *dxs* = *insert x* (*Dlist xs*)
by (*simp-all add*: *member-def mem-iff insert-def distinct-remdups-id*)
with *insert* **show** *P* **.**
qed
qed

32 Implementation of sets by distinct lists – canonical!

definition *Set* :: *'a dlist* ⇒ *'a fset* **where**
Set dxs = *Fset.Set* (*list-of-dlist dxs*)

definition *Coset* :: *'a dlist* ⇒ *'a fset* **where**
Coset dxs = *Fset.Coset* (*list-of-dlist dxs*)

code-datatype *Set Coset*

declare *member-code* [*code del*]
declare *is-empty-Set* [*code del*]
declare *empty-Set* [*code del*]
declare *UNIV-Set* [*code del*]
declare *insert-Set* [*code del*]
declare *remove-Set* [*code del*]
declare *compl-Set* [*code del*]
declare *compl-Coset* [*code del*]
declare *map-Set* [*code del*]
declare *filter-Set* [*code del*]
declare *forall-Set* [*code del*]
declare *exists-Set* [*code del*]
declare *card-Set* [*code del*]
declare *inter-project* [*code del*]
declare *subtract-remove* [*code del*]
declare *union-insert* [*code del*]
declare *Infimum-inf* [*code del*]
declare *Supremum-sup* [*code del*]

lemma *Set-Dlist* [*simp*]:
Set (*Dlist xs*) = *Fset* (*set xs*)
by (*simp add*: *Set-def Fset.Set-def*)

lemma *Coset-Dlist* [*simp*]:
Coset (*Dlist xs*) = *Fset* (− *set xs*)
by (*simp add*: *Coset-def Fset.Coset-def*)

lemma *member-Set* [*simp*]:
Fset.member (*Set dxs*) = *List.member* (*list-of-dlist dxs*)
by (*simp add*: *Set-def member-set*)

lemma *member-Coset* [*simp*]:
Fset.member (*Coset dxs*) = *Not* ∘ *List.member* (*list-of-dlist dxs*)
by (*simp add*: *Coset-def member-set not-set-compl*)

lemma *Set-dlist-of-list* [*code*]:
Fset.Set xs = *Set* (*dlist-of-list xs*)
by *simp*

lemma *Coset-dlist-of-list* [*code*]:
Fset.Coset xs = *Coset* (*dlist-of-list xs*)
by *simp*

lemma *is-empty-Set* [*code*]:
Fset.is-empty (*Set dxs*) ⟷ *null dxs*
by (*simp add*: *null-def null-empty member-set*)

lemma *bot-code* [*code*]:
bot = *Set empty*

by (*simp add*: *empty-def*)

lemma *top-code* [*code*]:
top = *Coset empty*
by (*simp add*: *empty-def*)

lemma *insert-code* [*code*]:
Fset.insert x (*Set dxs*) = *Set* (*insert x dxs*)
Fset.insert x (*Coset dxs*) = *Coset* (*remove x dxs*)
by (*simp-all add*: *insert-def remove-def member-set not-set-compl*)

lemma *remove-code* [*code*]:
Fset.remove x (*Set dxs*) = *Set* (*remove x dxs*)
Fset.remove x (*Coset dxs*) = *Coset* (*insert x dxs*)
by (*auto simp add*: *insert-def remove-def member-set not-set-compl*)

lemma *member-code* [*code*]:
Fset.member (*Set dxs*) = *member dxs*
Fset.member (*Coset dxs*) = *Not* ∘ *member dxs*
by (*simp-all add*: *member-def*)

lemma *compl-code* [*code*]:
− *Set dxs* = *Coset dxs*
− *Coset dxs* = *Set dxs*
by (*simp-all add*: *not-set-compl member-set*)

lemma *map-code* [*code*]:
Fset.map f (*Set dxs*) = *Set* (*map f dxs*)
by (*simp add*: *member-set*)

lemma *filter-code* [*code*]:
Fset.filter f (*Set dxs*) = *Set* (*filter f dxs*)
by (*simp add*: *member-set*)

lemma *forall-Set* [*code*]:
Fset.forall P (*Set xs*) ⟷ *list-all P* (*list-of-dlist xs*)
by (*simp add*: *member-set list-all-iff*)

lemma *exists-Set* [*code*]:
Fset.exists P (*Set xs*) ⟷ *list-ex P* (*list-of-dlist xs*)
by (*simp add*: *member-set list-ex-iff*)

lemma *card-code* [*code*]:
Fset.card (*Set dxs*) = *length dxs*
by (*simp add*: *length-def member-set distinct-card*)

lemma *inter-code* [*code*]:
inf A (*Set xs*) = *Set* (*filter* (*Fset.member A*) *xs*)
inf A (*Coset xs*) = *foldr Fset.remove xs A*

by (*simp-all only*: *Set-def Coset-def foldr-def inter-project list-of-dlist-filter*)

lemma *subtract-code* [*code*]:
A − Set xs = foldr Fset.remove xs A
A − Coset xs = Set (*filter* (*Fset.member A*) *xs*)
by (*simp-all only*: *Set-def Coset-def foldr-def subtract-remove list-of-dlist-filter*)

lemma *union-code* [*code*]:
sup (*Set xs*) *A = foldr Fset.insert xs A*
sup (*Coset xs*) *A = Coset* (*filter* (*Not* ∘ *Fset.member A*) *xs*)
by (*simp-all only*: *Set-def Coset-def foldr-def union-insert list-of-dlist-filter*)

context *complete-lattice*
begin

lemma *Infimum-code* [*code*]:
Infimum (*Set As*) = *foldr inf As top*
by (*simp only*: *Set-def Infimum-inf foldr-def inf.commute*)

lemma *Supremum-code* [*code*]:
Supremum (*Set As*) = *foldr sup As bot*
by (*simp only*: *Set-def Supremum-sup foldr-def sup.commute*)

end

hide-const (**open**) *member fold foldr empty insert remove map filter null member length fold*

end

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

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

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

33.1 Basic arithmetic

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

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

lemma *zero-nat-code* [*code*, *code-unfold-post*]:
0 = (*Numeral0* :: *nat*)
by *simp*

lemma *one-nat-code* [*code*, *code-unfold-post*]:
1 = (*Numeral1* :: *nat*)
by *simp*

lemma *Suc-code* [*code*]:
Suc n = *n* + *1*
by *simp*

lemma *plus-nat-code* [*code*]:
n + *m* = *nat* (*of-nat n* + *of-nat m*)
by *simp*

lemma *minus-nat-code* [*code*]:
n − *m* = *nat* (*of-nat n* − *of-nat m*)
by *simp*

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

Specialized *op div* and *op mod* operations.

definition *divmod-aux* :: *nat* ⇒ *nat* ⇒ *nat* × *nat* **where**
[*code del*]: *divmod-aux* = *divmod-nat*

lemma [*code*]:
divmod-nat n m = (*if m* = *0 then* (*0*, *n*) *else divmod-aux n m*)
unfolding *divmod-aux-def divmod-nat-div-mod* **by** *simp*

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

lemma *eq-nat-code* [*code*]:
eq-class.eq n m ⟷ *eq-class.eq* (*of-nat n* :: *int*) (*of-nat m*)
by (*simp add*: *eq*)

lemma *eq-nat-refl* [*code nbe*]:
eq-class.eq (*n*::*nat*) *n* ⟷ *True*
by (*rule HOL.eq-refl*)

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

by *simp*

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

33.2 Case analysis

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

lemma [*code*, *code-unfold*]:
$\mathit{nat\text{-}case} = (\lambda f\ g\ n.\ \mathit{if}\ n = 0\ \mathit{then}\ f\ \mathit{else}\ g\ (n - 1))$
by (*auto simp add*: *expand-fun-eq dest*!: *gr0-implies-Suc*)

33.3 Preprocessors

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

lemma *Suc-if-eq*: $(\bigwedge n.\ f\ (\mathit{Suc}\ n) \equiv h\ n) \Longrightarrow f\ 0 \equiv g \Longrightarrow$
$f\ n \equiv \mathit{if}\ n = 0\ \mathit{then}\ g\ \mathit{else}\ h\ (n - 1)$
by (*rule eq-reflection*) (*cases n*, *simp-all*)

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

33.4 Target language setup

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

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

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

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

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

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

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

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

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

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

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

code-reserved *Haskell Nat*

code-include *Scala Nat* ⟨⟨
```
import scala.Math

object Nat {

  def apply(numeral: BigInt): Nat = new Nat(numeral max 0)
```

```
  def apply(numeral: Int): Nat = Nat(BigInt(numeral))
  def apply(numeral: String): Nat = Nat(BigInt(numeral))

}

class Nat private(private val value: BigInt) {

  override def hashCode(): Int = this.value.hashCode()

  override def equals(that: Any): Boolean = that match {
    case that: Nat => this equals that
    case - => false
  }

  override def toString(): String = this.value.toString

  def equals(that: Nat): Boolean = this.value == that.value

  def as-BigInt: BigInt = this.value
  def as-Int: Int = if (this.value >= Math.MAX-INT && this.value <= Math.MAX-INT)
      this.value.intValue
    else error(Int value too big: + this.value.toString)

  def +(that: Nat): Nat = new Nat(this.value + that.value)
  def -(that: Nat): Nat = Nat(this.value - that.value)
  def *(that: Nat): Nat = new Nat(this.value * that.value)

  def /%(that: Nat): (Nat, Nat) = if (that.value == 0) (new Nat(0), this)
    else {
      val (k, l) = this.value /% that.value
      (new Nat(k), new Nat(l))
    }

  def <=(that: Nat): Boolean = this.value <= that.value

  def <(that: Nat): Boolean = this.value < that.value

}
```

code-reserved *Scala Nat*

code-type *nat*
(*Haskell Nat.Nat*)
(*Scala Nat.Nat*)

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

Natural numerals.

lemma [*code-unfold-post*]:
nat (*number-of i*) = *number-nat-inst.number-of-nat i*
— this interacts as desired with *number-of ?v* = *nat* (*number-of ?v*)
by (*simp add*: *number-nat-inst.number-of-nat*)

setup ⟪
fold (*Numeral.add-code* @{*const-name number-nat-inst.number-of-nat*}
false Code-Printer.literal-positive-numeral) [*SML*, *OCaml*, *Haskell*]
#> *Numeral.add-code* @{*const-name number-nat-inst.number-of-nat*}
false Code-Printer.literal-positive-numeral Scala
⟫

Since natural numbers are implemented using integers in ML, the coercion function *of-nat* of type *nat* ⇒ *int* is simply implemented by the identity function. For the *nat* function for converting an integer to a natural number, we give a specific implementation using an ML function that returns its input value, provided that it is non-negative, and otherwise returns *0*.

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

lemma *int-code'* [*code*]:
int (*number-of l*) = (*if neg* (*number-of l* :: *int*) *then 0 else number-of l*)
unfolding *int-nat-number-of* [*folded int-def*] **..**

lemma *nat-code'* [*code*]:
nat (*number-of l*) = (*if neg* (*number-of l* :: *int*) *then 0 else number-of l*)
unfolding *nat-number-of-def number-of-is-id neg-def* **by** *simp*

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

lemma *of-nat-aux-int* [*code-unfold*]:
of-nat-aux (λ*i*. *i* + *1*) *k 0* = *int k*
by (*simp add*: *int-def Nat.of-nat-code*)

code-const *int*
(*SML* -)
(*OCaml* -)

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

code-const *nat*
(*SML IntInf.max*/ (/*0*,/ -))
(*OCaml Big'-int.max'-big'-int*/ *Big'-int.zero'-big'-int*)

For Haskell and Scala, things are slightly different again.

code-const *int* **and** *nat*
(*Haskell toInteger* **and** *fromInteger*)
(*Scala* !*-.as′-BigInt* **and** !*Nat.Nat*((-)))

Conversion from and to indices.

code-const *Code-Numeral.of-nat*
(*SML IntInf.toInt*)
(*OCaml* -)
(*Haskell fromEnum*)
(*Scala* !*-.as′-Int*)

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

Using target language arithmetic operations whenever appropriate

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

code-const *op* − :: *nat* ⇒ *nat* ⇒ *nat*
(*Haskell* **infixl** *6* −)
(*Scala* **infixl** *7* −)

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

code-const *divmod-aux*
(*SML IntInf.divMod*/ ((-),/ (-)))
(*OCaml Big′-int.quomod′-big′-int*)
(*Haskell divMod*)
(*Scala* **infixl** *8* /%)

code-const *divmod-nat*
(*Haskell divMod*)
(*Scala* **infixl** *8* /%)

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

(*Scala* **infixl** *5* ==)

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

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

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

Evaluation

lemma [*code*, *code del*]:
(*Code-Evaluation.term-of* :: *nat* ⇒ *term*) = *Code-Evaluation.term-of* **..**

code-const *Code-Evaluation.term-of* :: *nat* ⇒ *term*
(*SML HOLogic.mk'-number*/ *HOLogic.natT*)

Module names

code-modulename *SML*
Efficient-Nat Arith

code-modulename *OCaml*
Efficient-Nat Arith

code-modulename *Haskell*
Efficient-Nat Arith

hide-const *int*

end

34 Enum: Finite types as explicit enumerations

theory *Enum*
imports *Map Main*

begin

34.1 Class *enum*

class *enum* =
  fixes *enum* :: *'a list*
  assumes *UNIV-enum*: *UNIV* = *set enum*
    and *enum-distinct*: *distinct enum*
begin

subclass *finite* **proof**
qed (*simp add*: *UNIV-enum*)

lemma *enum-all*: *set enum* = *UNIV* **unfolding** *UNIV-enum* **..**

lemma *in-enum* [*intro*]: *x* ∈ *set enum*
  unfolding *enum-all* **by** *auto*

lemma *enum-eq-I*:
  assumes ⋀*x*. *x* ∈ *set xs*
  shows *set enum* = *set xs*
proof −
  from *assms UNIV-eq-I* **have** *UNIV* = *set xs* **by** *auto*
  with *enum-all* **show** *?thesis* **by** *simp*
qed

end

34.2 Equality and order on functions

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

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

instance proof
qed (*simp-all add*: *eq-fun-def enum-all expand-fun-eq*)

end

lemma *order-fun* [*code*]:
  fixes *f g* :: *'a*::*enum* ⇒ *'b*::*order*
  shows *f* ≤ *g* ⟷ *list-all* (λ*x*. *f x* ≤ *g x*) *enum*
    and *f* < *g* ⟷ *f* ≤ *g* ∧ ¬ *list-all* (λ*x*. *f x* = *g x*) *enum*
  by (*simp-all add*: *list-all-iff enum-all expand-fun-eq le-fun-def order-less-le*)

34.3 Quantifiers

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

by (*simp add*: *list-all-iff enum-all*)

lemma *exists-code* [*code*]: $(\exists x.\ P\ x) \longleftrightarrow \neg$ *list-all* (*Not o P*) *enum*
by (*simp add*: *list-all-iff enum-all*)

34.4 Default instances

primrec *n-lists* :: *nat* ⇒ *'a list* ⇒ *'a list list* **where**
n-lists 0 xs = [[]]
| *n-lists* (*Suc n*) *xs* = *concat* (*map* (λ*ys*. *map* (λ*y*. *y* # *ys*) *xs*) (*n-lists n xs*))

lemma *n-lists-Nil* [*simp*]: *n-lists n* [] = (*if n* = *0 then* [[]] *else* [])
by (*induct n*) *simp-all*

lemma *length-n-lists*: *length* (*n-lists n xs*) = *length xs* ^ *n*
by (*induct n*) (*auto simp add*: *length-concat o-def listsum-triv*)

lemma *length-n-lists-elem*: *ys* ∈ *set* (*n-lists n xs*) ⟹ *length ys* = *n*
by (*induct n arbitrary*: *ys*) *auto*

lemma *set-n-lists*: *set* (*n-lists n xs*) = {*ys*. *length ys* = *n* ∧ *set ys* ⊆ *set xs*}
proof (*rule set-ext*)
fix *ys* :: *'a list*
show *ys* ∈ *set* (*n-lists n xs*) ⟷ *ys* ∈ {*ys*. *length ys* = *n* ∧ *set ys* ⊆ *set xs*}
proof −
have *ys* ∈ *set* (*n-lists n xs*) ⟹ *length ys* = *n*
by (*induct n arbitrary*: *ys*) *auto*
moreover have ⋀*x*. *ys* ∈ *set* (*n-lists n xs*) ⟹ *x* ∈ *set ys* ⟹ *x* ∈ *set xs*
by (*induct n arbitrary*: *ys*) *auto*
moreover have *set ys* ⊆ *set xs* ⟹ *ys* ∈ *set* (*n-lists* (*length ys*) *xs*)
by (*induct ys*) *auto*
ultimately show *?thesis* **by** *auto*
qed
qed

lemma *distinct-n-lists*:
assumes *distinct xs*
shows *distinct* (*n-lists n xs*)
proof (*rule card-distinct*)
from *assms* **have** *card-length*: *card* (*set xs*) = *length xs* **by** (*rule distinct-card*)
have *card* (*set* (*n-lists n xs*)) = *card* (*set xs*) ^ *n*
proof (*induct n*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc n*)
moreover have *card* (⋃ *ys*∈*set* (*n-lists n xs*). (λ*y*. *y* # *ys*) ‘ *set xs*)
= (∑ *ys*∈*set* (*n-lists n xs*). *card* ((λ*y*. *y* # *ys*) ‘ *set xs*))
by (*rule card-UN-disjoint*) *auto*
moreover have ⋀*ys*. *card* ((λ*y*. *y* # *ys*) ‘ *set xs*) = *card* (*set xs*)

```
    by (rule card-image) (simp add: inj-on-def)
   ultimately show ?case by auto
 qed
 also have ... = length xs ^ n by (simp add: card-length)
 finally show card (set (n-lists n xs)) = length (n-lists n xs)
   by (simp add: length-n-lists)
qed

lemma map-of-zip-enum-is-Some:
 assumes length ys = length (enum :: 'a::enum list)
 shows ∃ y. map-of (zip (enum :: 'a::enum list) ys) x = Some y
proof −
 from assms have x ∈ set (enum :: 'a::enum list) ⟷
   (∃ y. map-of (zip (enum :: 'a::enum list) ys) x = Some y)
   by (auto intro!: map-of-zip-is-Some)
 then show ?thesis using enum-all by auto
qed

lemma map-of-zip-enum-inject:
 fixes xs ys :: 'b::enum list
 assumes length: length xs = length (enum :: 'a::enum list)
     length ys = length (enum :: 'a::enum list)
   and map-of: the ∘ map-of (zip (enum :: 'a::enum list) xs) = the ∘ map-of (zip
(enum :: 'a::enum list) ys)
 shows xs = ys
proof −
 have map-of (zip (enum :: 'a list) xs) = map-of (zip (enum :: 'a list) ys)
 proof
   fix x :: 'a
   from length map-of-zip-enum-is-Some obtain y1 y2
     where map-of (zip (enum :: 'a list) xs) x = Some y1
       and map-of (zip (enum :: 'a list) ys) x = Some y2 by blast
   moreover from map-of have the (map-of (zip (enum :: 'a::enum list) xs) x)
= the (map-of (zip (enum :: 'a::enum list) ys) x)
     by (auto dest: fun-cong)
    ultimately show map-of (zip (enum :: 'a::enum list) xs) x = map-of (zip
(enum :: 'a::enum list) ys) x
     by simp
 qed
 with length enum-distinct show xs = ys by (rule map-of-zip-inject)
qed

instantiation fun :: (enum, enum) enum
begin

definition
  [code del]: enum = map (λys. the o map-of (zip (enum::'a list) ys)) (n-lists
(length (enum::'a::enum list)) enum)
```

```
instance proof
  show UNIV = set (enum :: ('a ⇒ 'b) list)
  proof (rule UNIV-eq-I)
    fix f :: 'a ⇒ 'b
    have f = the ∘ map-of (zip (enum :: 'a::enum list) (map f enum))
      by (auto simp add: map-of-zip-map expand-fun-eq)
    then show f ∈ set enum
      by (auto simp add: enum-fun-def set-n-lists)
  qed
next
  from map-of-zip-enum-inject
  show distinct (enum :: ('a ⇒ 'b) list)
    by (auto intro!: inj-onI simp add: enum-fun-def
      distinct-map distinct-n-lists enum-distinct set-n-lists enum-all)
qed

end

lemma enum-fun-code [code]: enum = (let enum-a = (enum :: 'a::{enum, eq} list)
  in map (λys. the o map-of (zip enum-a ys)) (n-lists (length enum-a) enum))
  by (simp add: enum-fun-def Let-def)

instantiation unit :: enum
begin

definition
  enum = [()]

instance proof
qed (simp-all add: enum-unit-def UNIV-unit)

end

instantiation bool :: enum
begin

definition
  enum = [False, True]

instance proof
qed (simp-all add: enum-bool-def UNIV-bool)

end

primrec product :: 'a list ⇒ 'b list ⇒ ('a × 'b) list where
  product [] - = []
  | product (x#xs) ys = map (Pair x) ys @ product xs ys

lemma product-list-set:
```

set (product xs ys) = set xs × set ys
by (*induct xs*) *auto*

lemma *distinct-product*:
assumes *distinct xs* **and** *distinct ys*
shows *distinct* (*product xs ys*)
using *assms* **by** (*induct xs*)
(*auto intro*: *inj-onI simp add*: *product-list-set distinct-map*)

instantiation ∗ :: (*enum*, *enum*) *enum*
begin

definition
enum = product enum enum

instance by *default*
(*simp-all add*: *enum-prod-def product-list-set distinct-product enum-all enum-distinct*)

end

instantiation + :: (*enum*, *enum*) *enum*
begin

definition
enum = map Inl enum @ map Inr enum

instance by *default*
(*auto simp add*: *enum-all enum-sum-def*, *case-tac x*, *auto intro*: *inj-onI simp add*: *distinct-map enum-distinct*)

end

primrec *sublists* :: *'a list ⇒ 'a list list* **where**
sublists [] = [[]]
| *sublists* (*x*#*xs*) = (*let xss = sublists xs in map* (*Cons x*) *xss @ xss*)

lemma *length-sublists*:
length (*sublists xs*) = *Suc* (*Suc* (*0*::*nat*)) ^ *length xs*
by (*induct xs*) (*simp-all add*: *Let-def*)

lemma *sublists-powset*:
set ' set (*sublists xs*) = *Pow* (*set xs*)
proof −
have *aux*: ⋀*x A*. *set ' Cons x ' A = insert x ' set ' A*
by (*auto simp add*: *image-def*)
have *set* (*map set* (*sublists xs*)) = *Pow* (*set xs*)
by (*induct xs*)
(*simp-all add*: *aux Let-def Pow-insert Un-commute comp-def del*: *map-map*)
then show *?thesis* **by** *simp*

qed

lemma *distinct-set-sublists*:
assumes *distinct xs*
shows *distinct* (*map set* (*sublists xs*))
proof (*rule card-distinct*)
have *finite* (*set xs*) **by** *rule*
then have *card* (*Pow* (*set xs*)) = *Suc* (*Suc 0*) ^ *card* (*set xs*) **by** (*rule card-Pow*)
with *assms distinct-card* [*of xs*]
have *card* (*Pow* (*set xs*)) = *Suc* (*Suc 0*) ^ *length xs* **by** *simp*
then show *card* (*set* (*map set* (*sublists xs*))) = *length* (*map set* (*sublists xs*))
by (*simp add*: *sublists-powset length-sublists*)
qed

instantiation *nibble* :: *enum*
begin

definition
enum = [*Nibble0*, *Nibble1*, *Nibble2*, *Nibble3*, *Nibble4*, *Nibble5*, *Nibble6*, *Nibble7*,
Nibble8, *Nibble9*, *NibbleA*, *NibbleB*, *NibbleC*, *NibbleD*, *NibbleE*, *NibbleF*]

instance proof
qed (*simp-all add*: *enum-nibble-def UNIV-nibble*)

end

instantiation *char* :: *enum*
begin

definition
[*code del*]: *enum* = *map* (*split Char*) (*product enum enum*)

lemma *enum-chars* [*code*]:
enum = *chars*
unfolding *enum-char-def chars-def enum-nibble-def* **by** *simp*

instance proof
qed (*auto intro*: *char.exhaust injI simp add*: *enum-char-def product-list-set enum-all full-SetCompr-eq* [*symmetric*]
distinct-map distinct-product enum-distinct)

end

instantiation *option* :: (*enum*) *enum*
begin

definition
enum = *None* # *map Some enum*

instance proof
qed (*auto simp add*: *enum-all enum-option-def*, *rule option.exhaust*, *auto intro*: *simp add*: *distinct-map enum-distinct*)

end

end

35 Eval-Witness: Evaluation Oracle with ML witnesses

theory *Eval-Witness*
imports *List Main*
begin

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

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

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

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

class *ml-equiv*

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

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

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

ML ⟪
structure Eval-Witness-Method =
struct

val eval-ref : (*unit* −> *bool*) *option Unsynchronized.ref* = *Unsynchronized.ref NONE*;

```
end;
⟩⟩

oracle eval-witness-oracle = ⟨⟨ fn (cgoal, ws) =>
let
  val thy = Thm.theory-of-cterm cgoal;
  val goal = Thm.term-of cgoal;
  fun check-type T =
    if Sorts.of-sort (Sign.classes-of thy) (T, [Eval-Witness.ml-equiv])
    then T
    else error (Type ^ quote (Syntax.string-of-typ-global 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 Code-Eval.eval NONE (Eval-Witness-Method.eval-ref, Eval-Witness-Method.eval-ref)
(K I) thy t ws
  then Thm.cterm-of thy goal
  else @{cprop True} (*dummy*)
end
⟩⟩


method-setup eval-witness = ⟨⟨
  Scan.lift (Scan.repeat Args.name) >>
  (fn ws => K (SIMPLE-METHOD'
    (CSUBGOAL (fn (ct, i) => rtac (eval-witness-oracle (ct, ws)) i))))
⟩⟩ evaluation with ML witnesses
```

35.1 Toy Examples

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

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

```
lemma ∃l. length (l::bool list) = 3
apply (eval-witness [true,true,true])
done
```

Multiple witnesses

```
lemma ∃k l. length (k::bool list) = length (l::bool list)
apply (eval-witness [] [])
done
```

35.2 Discussion

35.2.1 Conflicts

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

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

35.2.2 Haskell

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

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

end

36 Executable-Set: A crude implementation of finite sets by lists – avoid using this at any cost!

theory *Executable-Set*
imports *More-Set*
begin

declare *mem-def* [*code del*]
declare *Collect-def* [*code del*]
declare *insert-code* [*code del*]
declare *vimage-code* [*code del*]

36.1 Set representation

setup ⟪
  Code.add-type-cmd set
⟫

definition *Set* :: *'a list* ⇒ *'a set* **where**
  [*simp*]: *Set* = *set*

definition *Coset* :: *'a list* ⇒ *'a set* **where**
  [*simp*]: *Coset xs* = − *set xs*

setup ⟪

Code.add-signature-cmd (*Set*, *'a list* ⇒ *'a set*)
#> *Code.add-signature-cmd* (*Coset*, *'a list* ⇒ *'a set*)
#> *Code.add-signature-cmd* (*set*, *'a list* ⇒ *'a set*)
#> *Code.add-signature-cmd* (*op* ∈, *'a* ⇒ *'a set* ⇒ *bool*)
⟩⟩

code-datatype *Set Coset*

consts-code
Coset (⟨**module**⟩*Coset*)
Set (⟨**module**⟩*Set*)
attach ⟨⟨
datatype 'a set = Set of 'a list | Coset of 'a list;
⟩⟩ — This assumes that there won't be a *Coset* without a *Set*

36.2 Basic operations

lemma [*code*]:
set xs = *Set* (*remdups xs*)
by *simp*

lemma [*code*]:
x ∈ *Set xs* ⟷ *member xs x*
x ∈ *Coset xs* ⟷ ¬ *member xs x*
by (*simp-all add*: *mem-iff*)

definition *is-empty* :: *'a set* ⇒ *bool* **where**
[*simp*]: *is-empty A* ⟷ *A* = {}

lemma [*code-unfold*]:
A = {} ⟷ *is-empty A*
by *simp*

definition *empty* :: *'a set* **where**
[*simp*]: *empty* = {}

lemma [*code-unfold*]:
{} = *empty*
by *simp*

lemma [*code-unfold*, *code-inline del*]:
empty = *Set* []
by *simp* — Otherwise η-expansion produces funny things.

setup ⟨⟨
Code.add-signature-cmd (*is-empty*, *'a set* ⇒ *bool*)
#> *Code.add-signature-cmd* (*empty*, *'a set*)
#> *Code.add-signature-cmd* (*insert*, *'a* ⇒ *'a set* ⇒ *'a set*)
#> *Code.add-signature-cmd* (*More-Set.remove*, *'a* ⇒ *'a set* ⇒ *'a set*)

```
  #> Code.add-signature-cmd (image, ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set)
  #> Code.add-signature-cmd (More-Set.project, ('a ⇒ bool) ⇒ 'a set ⇒ 'a set)
  #> Code.add-signature-cmd (Ball, 'a set ⇒ ('a ⇒ bool) ⇒ bool)
  #> Code.add-signature-cmd (Bex, 'a set ⇒ ('a ⇒ bool) ⇒ bool)
  #> Code.add-signature-cmd (card, 'a set ⇒ nat)
⟫

lemma is-empty-Set [code]:
  is-empty (Set xs) ⟷ null xs
  by (simp add: empty-null)

lemma empty-Set [code]:
  empty = Set []
  by simp

lemma insert-Set [code]:
  insert x (Set xs) = Set (List.insert x xs)
  insert x (Coset xs) = Coset (removeAll x xs)
  by (simp-all add: set-insert)

lemma remove-Set [code]:
  remove x (Set xs) = Set (removeAll x xs)
  remove x (Coset xs) = Coset (List.insert x xs)
  by (auto simp add: set-insert remove-def)

lemma image-Set [code]:
  image f (Set xs) = Set (remdups (map f xs))
  by simp

lemma project-Set [code]:
  project P (Set xs) = Set (filter P xs)
  by (simp add: project-set)

lemma Ball-Set [code]:
  Ball (Set xs) P ⟷ list-all P xs
  by (simp add: ball-set)

lemma Bex-Set [code]:
  Bex (Set xs) P ⟷ list-ex P xs
  by (simp add: bex-set)

lemma card-Set [code]:
  card (Set xs) = length (remdups xs)
proof −
  have card (set (remdups xs)) = length (remdups xs)
    by (rule distinct-card) simp
  then show ?thesis by simp
qed
```

36.3 Derived operations

definition *set-eq* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
  [*simp*]: *set-eq* = *op* =

lemma [*code-unfold*]:
  op = = *set-eq*
  by *simp*

definition *subset-eq* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
  [*simp*]: *subset-eq* = *op* ⊆

lemma [*code-unfold*]:
  op ⊆ = *subset-eq*
  by *simp*

definition *subset* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
  [*simp*]: *subset* = *op* ⊂

lemma [*code-unfold*]:
  op ⊂ = *subset*
  by *simp*

setup ⟪
  Code.add-signature-cmd (*set-eq*, *'a set* ⇒ *'a set* ⇒ *bool*)
  #> *Code.add-signature-cmd* (*subset-eq*, *'a set* ⇒ *'a set* ⇒ *bool*)
  #> *Code.add-signature-cmd* (*subset*, *'a set* ⇒ *'a set* ⇒ *bool*)
⟫

lemma *set-eq-subset-eq* [*code*]:
  set-eq A B ⟷ *subset-eq A B* ∧ *subset-eq B A*
  by *auto*

lemma *subset-eq-forall* [*code*]:
  subset-eq A B ⟷ (∀ *x*∈*A*. *x* ∈ *B*)
  by (*simp add*: *subset-eq*)

lemma *subset-subset-eq* [*code*]:
  subset A B ⟷ *subset-eq A B* ∧ ¬ *subset-eq B A*
  by (*simp add*: *subset*)

36.4 Functorial operations

definition *inter* :: *'a set* ⇒ *'a set* ⇒ *'a set* **where**
  [*simp*]: *inter* = *op* ∩

lemma [*code-unfold*]:
  op ∩ = *inter*
  by *simp*

definition *subtract* :: *'a set* ⇒ *'a set* ⇒ *'a set* **where**
[*simp*]: *subtract A B* = *B* − *A*

lemma [*code-unfold*]:
B − *A* = *subtract A B*
by *simp*

definition *union* :: *'a set* ⇒ *'a set* ⇒ *'a set* **where**
[*simp*]: *union* = *op* ∪

lemma [*code-unfold*]:
op ∪ = *union*
by *simp*

definition *Inf* :: *'a*::*complete-lattice set* ⇒ *'a* **where**
[*simp*]: *Inf* = *Complete-Lattice.Inf*

lemma [*code-unfold*]:
Complete-Lattice.Inf = *Inf*
by *simp*

definition *Sup* :: *'a*::*complete-lattice set* ⇒ *'a* **where**
[*simp*]: *Sup* = *Complete-Lattice.Sup*

lemma [*code-unfold*]:
Complete-Lattice.Sup = *Sup*
by *simp*

definition *Inter* :: *'a set set* ⇒ *'a set* **where**
[*simp*]: *Inter* = *Inf*

lemma [*code-unfold*]:
Inf = *Inter*
by *simp*

definition *Union* :: *'a set set* ⇒ *'a set* **where**
[*simp*]: *Union* = *Sup*

lemma [*code-unfold*]:
Sup = *Union*
by *simp*

setup ⟪
Code.add-signature-cmd (*inter*, *'a set* ⇒ *'a set* ⇒ *'a set*)
#> *Code.add-signature-cmd* (*subtract*, *'a set* ⇒ *'a set* ⇒ *'a set*)
#> *Code.add-signature-cmd* (*union*, *'a set* ⇒ *'a set* ⇒ *'a set*)
#> *Code.add-signature-cmd* (*Inf*, *'a set* ⇒ *'a*)
#> *Code.add-signature-cmd* (*Sup*, *'a set* ⇒ *'a*)
#> *Code.add-signature-cmd* (*Inter*, *'a set set* ⇒ *'a set*)

```
  #> Code.add-signature-cmd (Union, 'a set set ⇒ 'a set)
⟫

lemma inter-project [code]:
  inter A (Set xs) = Set (List.filter (λx. x ∈ A) xs)
  inter A (Coset xs) = foldr remove xs A
  by (simp add: inter project-def) (simp add: Diff-eq [symmetric] minus-set-foldr)

lemma subtract-remove [code]:
  subtract (Set xs) A = foldr remove xs A
  subtract (Coset xs) A = Set (List.filter (λx. x ∈ A) xs)
  by (auto simp add: minus-set-foldr)

lemma union-insert [code]:
  union (Set xs) A = foldr insert xs A
  union (Coset xs) A = Coset (List.filter (λx. x ∉ A) xs)
  by (auto simp add: union-set-foldr)

lemma Inf-inf [code]:
  Inf (Set xs) = foldr inf xs (top :: 'a::complete-lattice)
  Inf (Coset []) = (bot :: 'a::complete-lattice)
  by (simp-all add: Inf-UNIV Inf-set-foldr)

lemma Sup-sup [code]:
  Sup (Set xs) = foldr sup xs (bot :: 'a::complete-lattice)
  Sup (Coset []) = (top :: 'a::complete-lattice)
  by (simp-all add: Sup-UNIV Sup-set-foldr)

lemma Inter-inter [code]:
  Inter (Set xs) = foldr inter xs (Coset [])
  Inter (Coset []) = empty
  unfolding Inter-def Inf-inf by simp-all

lemma Union-union [code]:
  Union (Set xs) = foldr union xs empty
  Union (Coset []) = Coset []
  unfolding Union-def Sup-sup by simp-all

hide-const (open) is-empty empty remove
  set-eq subset-eq subset inter union subtract Inf Sup Inter Union

end
```

37 Lattice-Algebras: Various algebraic structures combined with a lattice

theory *Lattice-Algebras*

imports *Complex-Main*
begin

class *semilattice-inf-ab-group-add* = *ordered-ab-group-add* + *semilattice-inf*
begin

lemma *add-inf-distrib-left*:
  a + *inf b c* = *inf* (*a* + *b*) (*a* + *c*)
apply (*rule antisym*)
apply (*simp-all add*: *le-infI*)
apply (*rule add-le-imp-le-left* [*of uminus a*])
apply (*simp only*: *add-assoc* [*symmetric*], *simp*)
apply *rule*
apply (*rule add-le-imp-le-left*[*of a*], *simp only*: *add-assoc*[*symmetric*], *simp*)+
done

lemma *add-inf-distrib-right*:
  inf a b + *c* = *inf* (*a* + *c*) (*b* + *c*)
proof −
  have *c* + *inf a b* = *inf* (*c*+*a*) (*c*+*b*) **by** (*simp add*: *add-inf-distrib-left*)
  thus *?thesis* **by** (*simp add*: *add-commute*)
qed

end

class *semilattice-sup-ab-group-add* = *ordered-ab-group-add* + *semilattice-sup*
begin

lemma *add-sup-distrib-left*:
  a + *sup b c* = *sup* (*a* + *b*) (*a* + *c*)
apply (*rule antisym*)
apply (*rule add-le-imp-le-left* [*of uminus a*])
apply (*simp only*: *add-assoc*[*symmetric*], *simp*)
apply *rule*
apply (*rule add-le-imp-le-left* [*of a*], *simp only*: *add-assoc*[*symmetric*], *simp*)+
apply (*rule le-supI*)
apply (*simp-all*)
done

lemma *add-sup-distrib-right*:
  sup a b + *c* = *sup* (*a*+*c*) (*b*+*c*)
proof −
  have *c* + *sup a b* = *sup* (*c*+*a*) (*c*+*b*) **by** (*simp add*: *add-sup-distrib-left*)
  thus *?thesis* **by** (*simp add*: *add-commute*)
qed

end

class *lattice-ab-group-add* = *ordered-ab-group-add* + *lattice*

begin

subclass *semilattice-inf-ab-group-add* **..**
subclass *semilattice-sup-ab-group-add* **..**

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

lemma *inf-eq-neg-sup*: *inf a b* = − *sup* (−*a*) (−*b*)
proof (*rule inf-unique*)
fix *a b* :: *'a*
show − *sup* (−*a*) (−*b*) ≤ *a*
by (*rule add-le-imp-le-right* [*of* - *sup* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-sup-distrib-left*)
next
fix *a b* :: *'a*
show − *sup* (−*a*) (−*b*) ≤ *b*
by (*rule add-le-imp-le-right* [*of* - *sup* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-sup-distrib-left*)
next
fix *a b c* :: *'a*
assume *a* ≤ *b a* ≤ *c*
then show *a* ≤ − *sup* (−*b*) (−*c*) **by** (*subst neg-le-iff-le* [*symmetric*])
(*simp add*: *le-supI*)
qed

lemma *sup-eq-neg-inf*: *sup a b* = − *inf* (−*a*) (−*b*)
proof (*rule sup-unique*)
fix *a b* :: *'a*
show *a* ≤ − *inf* (−*a*) (−*b*)
by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-inf-distrib-left*)
next
fix *a b* :: *'a*
show *b* ≤ − *inf* (−*a*) (−*b*)
by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-inf-distrib-left*)
next
fix *a b c* :: *'a*
assume *a* ≤ *c b* ≤ *c*
then show − *inf* (−*a*) (−*b*) ≤ *c* **by** (*subst neg-le-iff-le* [*symmetric*])
(*simp add*: *le-infI*)
qed

lemma *neg-inf-eq-sup*: − *inf a b* = *sup* (−*a*) (−*b*)
by (*simp add*: *inf-eq-neg-sup*)

lemma *neg-sup-eq-inf*: − *sup a b* = *inf* (−*a*) (−*b*)
by (*simp add*: *sup-eq-neg-inf*)

lemma *add-eq-inf-sup*: $a + b = sup\ a\ b + inf\ a\ b$
proof −
  have $0 = -\ inf\ 0\ (a-b) + inf\ (a-b)\ 0$ **by** (*simp add*: *inf-commute*)
  hence $0 = sup\ 0\ (b-a) + inf\ (a-b)\ 0$ **by** (*simp add*: *inf-eq-neg-sup*)
  hence $0 = (-a + sup\ a\ b) + (inf\ a\ b + (-b))$
    by (*simp add*: *add-sup-distrib-left add-inf-distrib-right*)
      (*simp add*: *algebra-simps*)
  thus *?thesis* **by** (*simp add*: *algebra-simps*)
qed

37.1 Positive Part, Negative Part, Absolute Value

definition
  $nprt :: 'a \Rightarrow 'a$ **where**
  $nprt\ x = inf\ x\ 0$

definition
  $pprt :: 'a \Rightarrow 'a$ **where**
  $pprt\ x = sup\ x\ 0$

lemma *pprt-neg*: $pprt\ (-\ x) = -\ nprt\ x$
proof −
  have $sup\ (-\ x)\ 0 = sup\ (-\ x)\ (-\ 0)$ **unfolding** *minus-zero* **..**
  also have $\ldots = -\ inf\ x\ 0$ **unfolding** *neg-inf-eq-sup* **..**
  finally have $sup\ (-\ x)\ 0 = -\ inf\ x\ 0$ **.**
  then show *?thesis* **unfolding** *pprt-def nprt-def* **.**
qed

lemma *nprt-neg*: $nprt\ (-\ x) = -\ pprt\ x$
proof −
  from *pprt-neg* **have** $pprt\ (-\ (-\ x)) = -\ nprt\ (-\ x)$ **.**
  then have $pprt\ x = -\ nprt\ (-\ x)$ **by** *simp*
  then show *?thesis* **by** *simp*
qed

lemma *prts*: $a = pprt\ a + nprt\ a$
by (*simp add*: *pprt-def nprt-def add-eq-inf-sup*[*symmetric*])

lemma *zero-le-pprt*[*simp*]: $0 \leq pprt\ a$
by (*simp add*: *pprt-def*)

lemma *nprt-le-zero*[*simp*]: $nprt\ a \leq 0$
by (*simp add*: *nprt-def*)

lemma *le-eq-neg*: $a \leq -\ b \longleftrightarrow a + b \leq 0$ (**is** $?l = ?r$)
proof −
  have *a*: $?l \longrightarrow ?r$
    apply (*auto*)

```
    apply (rule add-le-imp-le-right[of - uminus b -])
    apply (simp add: add-assoc)
    done
  have b: ?r ⟶ ?l
    apply (auto)
    apply (rule add-le-imp-le-right[of - b -])
    apply (simp)
    done
  from a b show ?thesis by blast
qed

lemma pprt-0[simp]: pprt 0 = 0 by (simp add: pprt-def)
lemma nprt-0[simp]: nprt 0 = 0 by (simp add: nprt-def)

lemma pprt-eq-id [simp, no-atp]: 0 ≤ x ⟹ pprt x = x
  by (simp add: pprt-def sup-aci sup-absorb1)

lemma nprt-eq-id [simp, no-atp]: x ≤ 0 ⟹ nprt x = x
  by (simp add: nprt-def inf-aci inf-absorb1)

lemma pprt-eq-0 [simp, no-atp]: x ≤ 0 ⟹ pprt x = 0
  by (simp add: pprt-def sup-aci sup-absorb2)

lemma nprt-eq-0 [simp, no-atp]: 0 ≤ x ⟹ nprt x = 0
  by (simp add: nprt-def inf-aci inf-absorb2)

lemma sup-0-imp-0: sup a (− a) = 0 ⟹ a = 0
proof −
  {
    fix a::'a
    assume hyp: sup a (−a) = 0
    hence sup a (−a) + a = a by (simp)
    hence sup (a+a) 0 = a by (simp add: add-sup-distrib-right)
    hence sup (a+a) 0 <= a by (simp)
    hence 0 <= a by (blast intro: order-trans inf-sup-ord)
  }
  note p = this
  assume hyp:sup a (−a) = 0
  hence hyp2:sup (−a) (−(−a)) = 0 by (simp add: sup-commute)
  from p[OF hyp] p[OF hyp2] show a = 0 by simp
qed

lemma inf-0-imp-0: inf a (−a) = 0 ⟹ a = 0
apply (simp add: inf-eq-neg-sup)
apply (simp add: sup-commute)
apply (erule sup-0-imp-0)
done

lemma inf-0-eq-0 [simp, no-atp]: inf a (− a) = 0 ⟷ a = 0
```

by (*rule*, *erule inf-0-imp-0*) *simp*

lemma *sup-0-eq-0* [*simp*, *no-atp*]: *sup a* (− *a*) = *0* ⟷ *a* = *0*
by (*rule*, *erule sup-0-imp-0*) *simp*

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]:
0 ≤ *a* + *a* ⟷ *0* ≤ *a*
proof
assume *0* <= *a* + *a*
hence *a*:*inf* (*a*+*a*) *0* = *0* **by** (*simp add*: *inf-commute inf-absorb1*)
have (*inf a 0*)+(*inf a 0*) = *inf* (*inf* (*a*+*a*) *0*) *a* (**is** *?l*=-)
by (*simp add*: *add-sup-inf-distribs inf-aci*)
hence *?l* = *0* + *inf a 0* **by** (*simp add*: *a*, *simp add*: *inf-commute*)
hence *inf a 0* = *0* **by** (*simp only*: *add-right-cancel*)
then show *0* <= *a* **unfolding** *le-iff-inf* **by** (*simp add*: *inf-commute*)
next
assume *a*: *0* <= *a*
show *0* <= *a* + *a* **by** (*simp add*: *add-mono*[*OF a a*, *simplified*])
qed

lemma *double-zero* [*simp*]:
a + *a* = *0* ⟷ *a* = *0*
proof
assume *assm*: *a* + *a* = *0*
then have *a* + *a* + − *a* = − *a* **by** *simp*
then have *a* + (*a* + − *a*) = − *a* **by** (*simp only*: *add-assoc*)
then have *a*: − *a* = *a* **by** *simp*
show *a* = *0* **apply** (*rule antisym*)
apply (*unfold neg-le-iff-le* [*symmetric*, *of a*])
unfolding *a* **apply** *simp*
unfolding *zero-le-double-add-iff-zero-le-single-add* [*symmetric*, *of a*]
unfolding *assm* **unfolding** *le-less* **apply** *simp-all* **done**
next
assume *a* = *0* **then show** *a* + *a* = *0* **by** *simp*
qed

lemma *zero-less-double-add-iff-zero-less-single-add* [*simp*]:
0 < *a* + *a* ⟷ *0* < *a*
proof (*cases a* = *0*)
case *True* **then show** *?thesis* **by** *auto*
next
case *False* **then show** *?thesis*
unfolding *less-le* **apply** *simp* **apply** *rule*
apply *clarify*
apply *rule*
apply *assumption*
apply (*rule notI*)
unfolding *double-zero* [*symmetric*, *of a*] **apply** *simp*
done

qed

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]:
$a + a \leq 0 \longleftrightarrow a \leq 0$
proof −
have $a + a \leq 0 \longleftrightarrow 0 \leq - (a + a)$ **by** (*subst le-minus-iff*, *simp*)
moreover have $\ldots \longleftrightarrow a \leq 0$ **by** (*simp add*: *zero-le-double-add-iff-zero-le-single-add*)
ultimately show *?thesis* **by** *blast*
qed

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]:
$a + a < 0 \longleftrightarrow a < 0$
proof −
have $a + a < 0 \longleftrightarrow 0 < - (a + a)$ **by** (*subst less-minus-iff*, *simp*)
moreover have $\ldots \longleftrightarrow a < 0$ **by** (*simp add*: *zero-less-double-add-iff-zero-less-single-add*)
ultimately show *?thesis* **by** *blast*
qed

declare *neg-inf-eq-sup* [*simp*] *neg-sup-eq-inf* [*simp*]

lemma *le-minus-self-iff*: $a \leq - a \longleftrightarrow a \leq 0$
proof −
from *add-le-cancel-left* [*of uminus a plus a a zero*]
have $(a <= -a) = (a+a <= 0)$
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *minus-le-self-iff*: $- a \leq a \longleftrightarrow 0 \leq a$
proof −
from *add-le-cancel-left* [*of uminus a zero plus a a*]
have $(-a <= a) = (0 <= a+a)$
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow nprt\ a = 0$
unfolding *le-iff-inf* **by** (*simp add*: *nprt-def inf-commute*)

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow pprt\ a = 0$
unfolding *le-iff-sup* **by** (*simp add*: *pprt-def sup-commute*)

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow pprt\ a = a$
unfolding *le-iff-sup* **by** (*simp add*: *pprt-def sup-commute*)

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow nprt\ a = a$
unfolding *le-iff-inf* **by** (*simp add*: *nprt-def inf-commute*)

lemma *pprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow pprt\ a \leq pprt\ b$

unfolding *le-iff-sup* **by** (*simp add*: *pprt-def sup-aci sup-assoc* [*symmetric*, *of a*])

lemma *nprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow nprt\ a \leq nprt\ b$
unfolding *le-iff-inf* **by** (*simp add*: *nprt-def inf-aci inf-assoc* [*symmetric*, *of a*])

end

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

class *lattice-ab-group-add-abs* = *lattice-ab-group-add* + *abs* +
  assumes *abs-lattice*: $|a| = sup\ a\ (-\ a)$
begin

lemma *abs-prts*: $|a| = pprt\ a - nprt\ a$
proof −
  have $0 \leq |a|$
  proof −
    have *a*: $a \leq |a|$ **and** *b*: $-\ a \leq |a|$ **by** (*auto simp add*: *abs-lattice*)
    show *?thesis* **by** (*rule add-mono* [*OF a b*, *simplified*])
  qed
  then have $0 \leq sup\ a\ (-\ a)$ **unfolding** *abs-lattice* **.**
  then have $sup\ (sup\ a\ (-\ a))\ 0 = sup\ a\ (-\ a)$ **by** (*rule sup-absorb1*)
  then show *?thesis*
    by (*simp add*: *add-sup-inf-distribs sup-aci*
      pprt-def nprt-def diff-minus abs-lattice)
qed

subclass *ordered-ab-group-add-abs*
proof
  have *abs-ge-zero* [*simp*]: $\bigwedge a.\ 0 \leq |a|$
  proof −
    fix *a b*
    have *a*: $a \leq |a|$ **and** *b*: $-\ a \leq |a|$ **by** (*auto simp add*: *abs-lattice*)
    show $0 \leq |a|$ **by** (*rule add-mono* [*OF a b*, *simplified*])
  qed
  have *abs-leI*: $\bigwedge a\ b.\ a \leq b \Longrightarrow -\ a \leq b \Longrightarrow |a| \leq b$
    by (*simp add*: *abs-lattice le-supI*)
  fix *a b*
  show $0 \leq |a|$ **by** *simp*
  show $a \leq |a|$
    by (*auto simp add*: *abs-lattice*)
  show $|-a| = |a|$
    by (*simp add*: *abs-lattice sup-commute*)
  show $a \leq b \Longrightarrow -\ a \leq b \Longrightarrow |a| \leq b$ **by** (*fact abs-leI*)
  show $|a + b| \leq |a| + |b|$
  proof −
    have *g*:$abs\ a\ +\ abs\ b = sup\ (a+b)\ (sup\ (-a-b)\ (sup\ (-a+b)\ (a\ +\ (-b))))$

```
(is -=sup ?m ?n)
     by (simp add: abs-lattice add-sup-inf-distribs sup-aci diff-minus)
   have a:a+b <= sup ?m ?n by (simp)
   have b:-a-b <= ?n by (simp)
   have c:?n <= sup ?m ?n by (simp)
   from b c have d: -a-b <= sup ?m ?n by(rule order-trans)
   have e:-a-b = -(a+b) by (simp add: diff-minus)
   from a d e have abs(a+b) <= sup ?m ?n
    by (drule-tac abs-leI, auto)
   with g[symmetric] show ?thesis by simp
  qed
qed

end

lemma sup-eq-if:
  fixes a :: 'a::{lattice-ab-group-add, linorder}
  shows sup a (- a) = (if a < 0 then - a else a)
proof -
  note add-le-cancel-right [of a a - a, symmetric, simplified]
  moreover note add-le-cancel-right [of -a a a, symmetric, simplified]
 then show ?thesis by (auto simp: sup-max min-max.sup-absorb1 min-max.sup-absorb2)
qed

lemma abs-if-lattice:
  fixes a :: 'a::{lattice-ab-group-add-abs, linorder}
  shows |a| = (if a < 0 then - a else a)
by auto

lemma estimate-by-abs:
  a + b <= (c::'a::lattice-ab-group-add-abs) ==> a <= c + abs b
proof -
  assume a+b <= c
  hence 2: a <= c+(-b) by (simp add: algebra-simps)
  have 3: (-b) <= abs b by (rule abs-ge-minus-self)
  show ?thesis by (rule le-add-right-mono[OF 2 3])
qed

class lattice-ring = ordered-ring + lattice-ab-group-add-abs
begin

subclass semilattice-inf-ab-group-add ..
subclass semilattice-sup-ab-group-add ..

end

lemma abs-le-mult: abs (a * b) ≤ (abs a) * (abs (b::'a::lattice-ring))
proof -
  let ?x = pprt a * pprt b - pprt a * nprt b - nprt a * pprt b + nprt a * nprt b
```

```
  let ?y = pprt a * pprt b + pprt a * nprt b + nprt a * pprt b + nprt a * nprt b
  have a: (abs a) * (abs b) = ?x
    by (simp only: abs-prts[of a] abs-prts[of b] algebra-simps)
  {
    fix u v :: 'a
    have bh: [[u = a; v = b]] ==>
              u * v = pprt a * pprt b + pprt a * nprt b +
                      nprt a * pprt b + nprt a * nprt b
      apply (subst prts[of u], subst prts[of v])
      apply (simp add: algebra-simps)
      done
  }
  note b = this[OF refl[of a] refl[of b]]
  have xy: - ?x <= ?y
    apply (simp)
    apply (rule order-trans [OF add-nonpos-nonpos add-nonneg-nonneg])
    apply (simp-all add: mult-nonneg-nonneg mult-nonpos-nonpos)
    done
  have yx: ?y <= ?x
    apply (simp add:diff-def)
    apply (rule order-trans [OF add-nonpos-nonpos add-nonneg-nonneg])
    apply (simp-all add: mult-nonneg-nonpos mult-nonpos-nonneg)
    done
  have i1: a*b <= abs a * abs b by (simp only: a b yx)
  have i2: - (abs a * abs b) <= a*b by (simp only: a b xy)
  show ?thesis
    apply (rule abs-leI)
    apply (simp add: i1)
    apply (simp add: i2[simplified minus-le-iff])
    done
qed

instance lattice-ring ⊆ ordered-ring-abs
proof
  fix a b :: 'a:: lattice-ring
  assume (0 ≤ a ∨ a ≤ 0) ∧ (0 ≤ b ∨ b ≤ 0)
  show abs (a*b) = abs a * abs b
  proof -
    have s: (0 <= a*b) | (a*b <= 0)
      apply (auto)
      apply (rule-tac split-mult-pos-le)
      apply (rule-tac contrapos-np[of a*b <= 0])
      apply (simp)
      apply (rule-tac split-mult-neg-le)
      apply (insert prems)
      apply (blast)
      done
    have mulprts: a * b = (pprt a + nprt a) * (pprt b + nprt b)
      by (simp add: prts[symmetric])
```

```
    show ?thesis
    proof cases
      assume 0 <= a * b
      then show ?thesis
        apply (simp-all add: mulprts abs-prts)
        apply (insert prems)
        apply (auto simp add:
          algebra-simps
          iffD1[OF zero-le-iff-zero-nprt] iffD1[OF le-zero-iff-zero-pprt]
          iffD1[OF le-zero-iff-pprt-id] iffD1[OF zero-le-iff-nprt-id])
          apply(drule (1) mult-nonneg-nonpos[of a b], simp)
          apply(drule (1) mult-nonneg-nonpos2[of b a], simp)
        done
    next
      assume ~(0 <= a*b)
      with s have a*b <= 0 by simp
      then show ?thesis
        apply (simp-all add: mulprts abs-prts)
        apply (insert prems)
        apply (auto simp add: algebra-simps)
        apply(drule (1) mult-nonneg-nonneg[of a b],simp)
        apply(drule (1) mult-nonpos-nonpos[of a b],simp)
        done
    qed
  qed
qed

lemma mult-le-prts:
  assumes
  a1 <= (a::'a::lattice-ring)
  a <= a2
  b1 <= b
  b <= b2
  shows
  a * b <= pprt a2 * pprt b2 + pprt a1 * nprt b2 + nprt a2 * pprt b1 + nprt a1
* nprt b1
proof -
  have a * b = (pprt a + nprt a) * (pprt b + nprt b)
    apply (subst prts[symmetric])+
    apply simp
    done
  then have a * b = pprt a * pprt b + pprt a * nprt b + nprt a * pprt b + nprt
a * nprt b
    by (simp add: algebra-simps)
  moreover have pprt a * pprt b <= pprt a2 * pprt b2
    by (simp-all add: prems mult-mono)
  moreover have pprt a * nprt b <= pprt a1 * nprt b2
  proof -
    have pprt a * nprt b <= pprt a * nprt b2
```

```
    by (simp add: mult-left-mono prems)
   moreover have pprt a * nprt b2 <= pprt a1 * nprt b2
    by (simp add: mult-right-mono-neg prems)
   ultimately show ?thesis
    by simp
  qed
  moreover have nprt a * pprt b <= nprt a2 * pprt b1
  proof -
   have nprt a * pprt b <= nprt a2 * pprt b
    by (simp add: mult-right-mono prems)
   moreover have nprt a2 * pprt b <= nprt a2 * pprt b1
    by (simp add: mult-left-mono-neg prems)
   ultimately show ?thesis
    by simp
  qed
  moreover have nprt a * nprt b <= nprt a1 * nprt b1
  proof -
   have nprt a * nprt b <= nprt a * nprt b1
    by (simp add: mult-left-mono-neg prems)
   moreover have nprt a * nprt b1 <= nprt a1 * nprt b1
    by (simp add: mult-right-mono-neg prems)
   ultimately show ?thesis
    by simp
  qed
  ultimately show ?thesis
   by - (rule add-mono | simp)+
qed

lemma mult-ge-prts:
  assumes
  a1 <= (a::'a::lattice-ring)
  a <= a2
  b1 <= b
  b <= b2
  shows
  a * b >= nprt a1 * pprt b2 + nprt a2 * nprt b2 + pprt a1 * pprt b1 + pprt a2
* nprt b1
proof -
  from prems have a1:- a2 <= -a by auto
  from prems have a2: -a <= -a1 by auto
  from mult-le-prts[of -a2 -a -a1 b1 b b2, OF a1 a2 prems(3) prems(4), sim-
plified nprt-neg pprt-neg]
  have le: - (a * b) <= - nprt a1 * pprt b2 + - nprt a2 * nprt b2 + - pprt a1
* pprt b1 + - pprt a2 * nprt b1 by simp
  then have -(- nprt a1 * pprt b2 + - nprt a2 * nprt b2 + - pprt a1 * pprt
b1 + - pprt a2 * nprt b1) <= a * b
   by (simp only: minus-le-iff)
  then show ?thesis by simp
qed
```

instance *int* :: *lattice-ring*
proof
  fix *k* :: *int*
  show *abs k* = *sup k* (− *k*)
    by (*auto simp add*: *sup-int-def*)
qed

instance *real* :: *lattice-ring*
proof
  fix *a* :: *real*
  show *abs a* = *sup a* (− *a*)
    by (*auto simp add*: *sup-real-def*)
qed

end

38 Float: Floating-Point Numbers

theory *Float*
imports *Complex-Main Lattice-Algebras*
begin

definition
  pow2 :: *int* ⇒ *real* **where**
  [*simp*]: *pow2 a* = (*if* (*0* <= *a*) *then* (*2*^(*nat a*)) *else* (*inverse* (*2*^(*nat* (−*a*)))))

datatype *float* = *Float int int*

primrec *of-float* :: *float* ⇒ *real* **where**
  of-float (*Float a b*) = *real a* ∗ *pow2 b*

defs (**overloaded**)
  real-of-float-def [*code-unfold*]: *real* == *of-float*

primrec *mantissa* :: *float* ⇒ *int* **where**
  mantissa (*Float a b*) = *a*

primrec *scale* :: *float* ⇒ *int* **where**
  scale (*Float a b*) = *b*

instantiation *float* :: *zero* **begin**
definition *zero-float* **where** *0* = *Float 0 0*
instance ..
end

instantiation *float* :: *one* **begin**
definition *one-float* **where** *1* = *Float 1 0*

instance ..
end

instantiation *float* :: *number* **begin**
definition *number-of-float* **where** *number-of n = Float n 0*
instance ..
end

lemma *number-of-float-Float* [*code-unfold-post*]:
number-of k = Float (number-of k) 0
by (*simp add*: *number-of-float-def number-of-is-id*)

lemma *real-of-float-simp*[*simp*]: *real (Float a b) = real a * pow2 b*
unfolding *real-of-float-def* **using** *of-float.simps* **.**

lemma *real-of-float-neg-exp*: *e < 0 ⟹ real (Float m e) = real m * inverse (2^nat (−e))* **by** *auto*
lemma *real-of-float-nge0-exp*: *¬ 0 ≤ e ⟹ real (Float m e) = real m * inverse (2^nat (−e))* **by** *auto*
lemma *real-of-float-ge0-exp*: *0 ≤ e ⟹ real (Float m e) = real m * (2^nat e)* **by** *auto*

lemma *Float-num*[*simp*]: **shows**
real (Float 1 0) = 1 **and** *real (Float 1 1) = 2* **and** *real (Float 1 2) = 4* **and**
real (Float 1 −1) = 1/2 **and** *real (Float 1 −2) = 1/4* **and** *real (Float 1 −3) = 1/8* **and**
real (Float −1 0) = −1 **and** *real (Float (number-of n) 0) = number-of n*
by *auto*

lemma *float-number-of*[*simp*]: *real (number-of x :: float) = number-of x*
by (*simp only*:*number-of-float-def Float-num*[*unfolded number-of-is-id*])

lemma *float-number-of-int*[*simp*]: *real (Float n 0) = real n*
by (*simp add*: *Float-num*[*unfolded number-of-is-id*] *real-of-float-simp pow2-def*)

lemma *pow2-0*[*simp*]: *pow2 0 = 1* **by** *simp*
lemma *pow2-1*[*simp*]: *pow2 1 = 2* **by** *simp*
lemma *pow2-neg*: *pow2 x = inverse (pow2 (−x))* **by** *simp*

declare *pow2-def*[*simp del*]

lemma *pow2-add1*: *pow2 (1 + a) = 2 * (pow2 a)*
proof −
have *h*: ! *n*. *nat (2 + int n) − Suc 0 = nat (1 + int n)* **by** *arith*
have *g*: ! *a b*. *a − −1 = a + (1*::*int)* **by** *arith*
have *pos*: ! *n*. *pow2 (int n + 1) = 2 * pow2 (int n)*
apply (*auto*, *induct-tac n*)
apply (*simp-all add*: *pow2-def*)
apply (*rule-tac m1=2* **and** *n1=nat (2 + int na)* **in** *ssubst*[*OF realpow-num-eq-if*])

```
    by (auto simp add: h)
  show ?thesis
  proof (induct a)
    case (1 n)
    from pos show ?case by (simp add: algebra-simps)
  next
    case (2 n)
    show ?case
      apply (auto)
      apply (subst pow2-neg[of - int n])
      apply (subst pow2-neg[of -1 - int n])
      apply (auto simp add: g pos)
      done
  qed
qed

lemma pow2-add: pow2 (a+b) = (pow2 a) * (pow2 b)
proof (induct b)
  case (1 n)
  show ?case
  proof (induct n)
    case 0
    show ?case by simp
  next
    case (Suc m)
    show ?case by (auto simp add: algebra-simps pow2-add1 prems)
  qed
next
  case (2 n)
  show ?case
  proof (induct n)
    case 0
    show ?case
      apply (auto)
      apply (subst pow2-neg[of a + -1])
      apply (subst pow2-neg[of -1])
      apply (simp)
      apply (insert pow2-add1[of -a])
      apply (simp add: algebra-simps)
      apply (subst pow2-neg[of -a])
      apply (simp)
      done
    case (Suc m)
    have a: int m - (a + -2) =  1 + (int m - a + 1) by arith
    have b: int m - -2 = 1 + (int m + 1) by arith
    show ?case
      apply (auto)
      apply (subst pow2-neg[of a + (-2 - int m)])
      apply (subst pow2-neg[of -2 - int m])
```

```
    apply (auto simp add: algebra-simps)
    apply (subst a)
    apply (subst b)
    apply (simp only: pow2-add1)
    apply (subst pow2-neg[of int m - a + 1])
    apply (subst pow2-neg[of int m + 1])
    apply auto
    apply (insert prems)
    apply (auto simp add: algebra-simps)
    done
  qed
qed

lemma float-components[simp]: Float (mantissa f) (scale f) = f by (cases f, auto)

lemma float-split: ∃ a b. x = Float a b by (cases x, auto)

lemma float-split2: (∀ a b. x ≠ Float a b) = False by (auto simp add: float-split)

lemma float-zero[simp]: real (Float 0 e) = 0 by simp

lemma abs-div-2-less: a ≠ 0 ⟹ a ≠ −1 ⟹ abs((a::int) div 2) < abs a
by arith

function normfloat :: float ⇒ float where
normfloat (Float a b) = (if a ≠ 0 ∧ even a then normfloat (Float (a div 2) (b+1))
else if a=0 then Float 0 0 else Float a b)
by pat-completeness auto
termination by (relation measure (nat o abs o mantissa)) (auto intro: abs-div-2-less)
declare normfloat.simps[simp del]

theorem normfloat[symmetric, simp]: real f = real (normfloat f)
proof (induct f rule: normfloat.induct)
  case (1 a b)
  have real2: 2 = real (2::int)
    by auto
  show ?case
    apply (subst normfloat.simps)
    apply (auto simp add: float-zero)
    apply (subst 1[symmetric])
    apply (auto simp add: pow2-add even-def)
    done
qed

lemma pow2-neq-zero[simp]: pow2 x ≠ 0
  by (auto simp add: pow2-def)

lemma pow2-int: pow2 (int c) = 2^c
by (simp add: pow2-def)
```

lemma *zero-less-pow2*[*simp*]:
0 < pow2 x
proof −
{
fix *y*
have *0 <= y ⟹ 0 < pow2 y*
by (*induct y*, *induct-tac n*, *simp-all add*: *pow2-add*)
}
note *helper=this*
show *?thesis*
apply (*case-tac 0 <= x*)
apply (*simp add*: *helper*)
apply (*subst pow2-neg*)
apply (*simp add*: *helper*)
done
qed

lemma *normfloat-imp-odd-or-zero*: *normfloat f = Float a b ⟹ odd a ∨ (a = 0 ∧ b = 0)*
proof (*induct f rule*: *normfloat.induct*)
case (*1 u v*)
from *1* **have** *ab*: *normfloat* (*Float u v*) = *Float a b* **by** *auto*
{
assume *eu*: *even u*
assume *z*: *u ≠ 0*
have *normfloat* (*Float u v*) = *normfloat* (*Float* (*u div 2*) (*v + 1*))
apply (*subst normfloat.simps*)
by (*simp add*: *eu z*)
with *ab* **have** *normfloat* (*Float* (*u div 2*) (*v + 1*)) = *Float a b* **by** *simp*
with *1 eu z* **have** *?case* **by** *auto*
}
note *case1 = this*
{
assume *odd u ∨ u = 0*
then have *ou*: *¬ (u ≠ 0 ∧ even u)* **by** *auto*
have *normfloat* (*Float u v*) = (*if u = 0 then Float 0 0 else Float u v*)
apply (*subst normfloat.simps*)
apply (*simp add*: *ou*)
done
with *ab* **have** *Float a b* = (*if u = 0 then Float 0 0 else Float u v*) **by** *auto*
then have *?case*
apply (*case-tac u=0*)
apply (*auto*)
by (*insert ou*, *auto*)
}
note *case2 = this*
show *?case*
apply (*case-tac odd u ∨ u = 0*)

```
    apply (rule case2)
    apply simp
    apply (rule case1)
    apply auto
    done
qed

lemma float-eq-odd-helper:
  assumes odd: odd a'
  and floateq: real (Float a b) = real (Float a' b')
  shows b ≤ b'
proof −
  {
    assume bcmp: b > b'
    from floateq have eq: real a * pow2 b = real a' * pow2 b' by simp
    {
      fix x y z :: real
      assume y ≠ 0
      then have (x * inverse y = z) = (x = z * y)
        by auto
    }
    note inverse = this
    have eq': real a * (pow2 (b − b')) = real a'
      apply (subst diff-int-def)
      apply (subst pow2-add)
      apply (subst pow2-neg[where x = −b'])
      apply simp
      apply (subst mult-assoc[symmetric])
      apply (subst inverse)
      apply (simp-all add: eq)
      done
    have ∃ z > 0. pow2 (b−b') = 2^z
      apply (rule exI[where x=nat (b − b')])
      apply (auto)
      apply (insert bcmp)
      apply simp
      apply (subst pow2-int[symmetric])
      apply auto
      done
    then obtain z where z: z > 0 ∧ pow2 (b−b') = 2^z by auto
    with eq' have real a * 2^z = real a'
      by auto
    then have real a * real ((2::int)^z) = real a'
      by auto
    then have real (a * 2^z) = real a'
      apply (subst real-of-int-mult)
      apply simp
      done
    then have a'-rep: a * 2^z = a' by arith
```

```
    then have a' = a*2^z by simp
    with z have even a' by simp
    with odd have False by auto
  }
  then show ?thesis by arith
qed

lemma float-eq-odd:
  assumes odd1: odd a
  and odd2: odd a'
  and floateq: real (Float a b) = real (Float a' b')
  shows a = a' ∧ b = b'
proof −
  from
    float-eq-odd-helper[OF odd2 floateq]
    float-eq-odd-helper[OF odd1 floateq[symmetric]]
  have beq: b = b' by arith
  with floateq show ?thesis by auto
qed

theorem normfloat-unique:
  assumes real-of-float-eq: real f = real g
  shows normfloat f = normfloat g
proof −
  from float-split[of normfloat f] obtain a b where normf:normfloat f = Float a
b by auto
  from float-split[of normfloat g] obtain a' b' where normg:normfloat g = Float
a' b' by auto
  have real (normfloat f) = real (normfloat g)
    by (simp add: real-of-float-eq)
  then have float-eq: real (Float a b) = real (Float a' b')
    by (simp add: normf normg)
   have ab: odd a ∨ (a = 0 ∧ b = 0) by (rule normfloat-imp-odd-or-zero[OF
normf])
  have ab': odd a' ∨ (a' = 0 ∧ b' = 0) by (rule normfloat-imp-odd-or-zero[OF
normg])
  {
    assume odd: odd a
    then have a ≠ 0 by (simp add: even-def, arith)
    with float-eq have a' ≠ 0 by auto
    with ab' have odd a' by simp
    from odd this float-eq have a = a' ∧ b = b' by (rule float-eq-odd)
  }
  note odd-case = this
  {
    assume even: even a
    with ab have a0: a = 0 by simp
    with float-eq have a0': a' = 0 by auto
    from a0 a0' ab ab' have a = a' ∧ b = b' by auto
```

```
  }
  note even-case = this
  from odd-case even-case show ?thesis
    apply (simp add: normf normg)
    apply (case-tac even a)
    apply auto
    done
qed

instantiation float :: plus begin
fun plus-float where
[simp del]: (Float a-m a-e) + (Float b-m b-e) =
     (if a-e ≤ b-e then Float (a-m + b-m * 2^(nat(b-e - a-e))) a-e
                   else Float (a-m * 2^(nat (a-e - b-e)) + b-m) b-e)
instance ..
end

instantiation float :: uminus begin
primrec uminus-float where [simp del]: uminus-float (Float m e) = Float (−m)
e
instance ..
end

instantiation float :: minus begin
definition minus-float where [simp del]: (z::float) − w = z + (− w)
instance ..
end

instantiation float :: times begin
fun times-float where [simp del]: (Float a-m a-e) * (Float b-m b-e) = Float (a-m
* b-m) (a-e + b-e)
instance ..
end

primrec float-pprt :: float ⇒ float where
  float-pprt (Float a e) = (if 0 <= a then (Float a e) else 0)

primrec float-nprt :: float ⇒ float where
  float-nprt (Float a e) = (if 0 <= a then 0 else (Float a e))

instantiation float :: ord begin
definition le-float-def: z ≤ (w :: float) ≡ real z ≤ real w
definition less-float-def: z < (w :: float) ≡ real z < real w
instance ..
end

lemma real-of-float-add[simp]: real (a + b) = real a + real (b :: float)
  by (cases a, cases b, simp add: algebra-simps plus-float.simps,
      auto simp add: pow2-int[symmetric] pow2-add[symmetric])
```

lemma *real-of-float-minus*[*simp*]: *real* (− *a*) = − *real* (*a* :: *float*)
by (*cases a*, *simp add*: *uminus-float.simps*)

lemma *real-of-float-sub*[*simp*]: *real* (*a* − *b*) = *real a* − *real* (*b* :: *float*)
by (*cases a*, *cases b*, *simp add*: *minus-float-def*)

lemma *real-of-float-mult*[*simp*]: *real* (*a*∗*b*) = *real a* ∗ *real* (*b* :: *float*)
by (*cases a*, *cases b*, *simp add*: *times-float.simps pow2-add*)

lemma *real-of-float-0*[*simp*]: *real* (*0* :: *float*) = *0*
by (*auto simp add*: *zero-float-def float-zero*)

lemma *real-of-float-1*[*simp*]: *real* (*1* :: *float*) = *1*
by (*auto simp add*: *one-float-def*)

lemma *zero-le-float*:
(*0* <= *real* (*Float a b*)) = (*0* <= *a*)
apply *auto*
apply (*auto simp add*: *zero-le-mult-iff*)
apply (*insert zero-less-pow2*[*of b*])
apply (*simp-all*)
done

lemma *float-le-zero*:
(*real* (*Float a b*) <= *0*) = (*a* <= *0*)
apply *auto*
apply (*auto simp add*: *mult-le-0-iff*)
apply (*insert zero-less-pow2*[*of b*])
apply *auto*
done

declare *real-of-float-simp*[*simp del*]

lemma *real-of-float-pprt*[*simp*]: *real* (*float-pprt a*) = *pprt* (*real a*)
by (*cases a*, *auto simp add*: *float-pprt.simps zero-le-float float-le-zero float-zero*)

lemma *real-of-float-nprt*[*simp*]: *real* (*float-nprt a*) = *nprt* (*real a*)
by (*cases a*, *auto simp add*: *float-nprt.simps zero-le-float float-le-zero float-zero*)

instance *float* :: *ab-semigroup-add*
proof (*intro-classes*)
fix *a b c* :: *float*
show *a* + *b* + *c* = *a* + (*b* + *c*)
by (*cases a*, *cases b*, *cases c*, *auto simp add*: *algebra-simps power-add*[*symmetric*]
plus-float.simps)
next
fix *a b* :: *float*
show *a* + *b* = *b* + *a*

by (*cases a*, *cases b*, *simp add*: *plus-float.simps*)
qed

instance *float* :: *comm-monoid-mult*
proof (*intro-classes*)
fix *a b c* :: *float*
show $a * b * c = a * (b * c)$
by (*cases a*, *cases b*, *cases c*, *simp add*: *times-float.simps*)
next
fix *a b* :: *float*
show $a * b = b * a$
by (*cases a*, *cases b*, *simp add*: *times-float.simps*)
next
fix *a* :: *float*
show $1 * a = a$
by (*cases a*, *simp add*: *times-float.simps one-float-def*)
qed

lemma $0 + Float\ 0\ 1 = 0 + Float\ 0\ 2$
by (*simp add*: *plus-float.simps zero-float-def*)

instance *float* :: *comm-semiring*
proof (*intro-classes*)
fix *a b c* :: *float*
show $(a + b) * c = a * c + b * c$
by (*cases a*, *cases b*, *cases c*, *simp*, *simp add*: *plus-float.simps times-float.simps algebra-simps*)
qed

instance *float* :: *zero-neq-one*
proof (*intro-classes*)
show $(0::float) \neq 1$
by (*simp add*: *zero-float-def one-float-def*)
qed

lemma *float-le-simp*: $((x::float) \leq y) = (0 \leq y - x)$
by (*auto simp add*: *le-float-def*)

lemma *float-less-simp*: $((x::float) < y) = (0 < y - x)$
by (*auto simp add*: *less-float-def*)

lemma *real-of-float-min*: *real* (*min x y* :: *float*) = *min* (*real x*) (*real y*) **unfolding** *min-def le-float-def* **by** *auto*
lemma *real-of-float-max*: *real* (*max a b* :: *float*) = *max* (*real a*) (*real b*) **unfolding** *max-def le-float-def* **by** *auto*

lemma *float-power*: *real* (*x* ^ *n* :: *float*) = *real x* ^ *n*
by (*induct n*) *simp-all*

lemma *zero-le-pow2*[*simp*]: *0* ≤ *pow2 s*
apply (*subgoal-tac 0* < *pow2 s*)
apply (*auto simp only*:)
apply *auto*
done

lemma *pow2-less-0-eq-False*[*simp*]: (*pow2 s* < *0*) = *False*
apply *auto*
apply (*subgoal-tac 0* ≤ *pow2 s*)
apply *simp*
apply *simp*
done

lemma *pow2-le-0-eq-False*[*simp*]: (*pow2 s* ≤ *0*) = *False*
apply *auto*
apply (*subgoal-tac 0* < *pow2 s*)
apply *simp*
apply *simp*
done

lemma *float-pos-m-pos*: *0* < *Float m e* ⟹ *0* < *m*
unfolding *less-float-def real-of-float-simp real-of-float-0 zero-less-mult-iff*
by *auto*

lemma *float-pos-less1-e-neg*: **assumes** *0* < *Float m e* **and** *Float m e* < *1* **shows** *e* < *0*
proof −
have *0* < *m* **using** *float-pos-m-pos* ‹*0* < *Float m e*› **by** *auto*
hence *0* ≤ *real m* **and** *1* ≤ *real m* **by** *auto*

show *e* < *0*
proof (*rule ccontr*)
assume ¬ *e* < *0* **hence** *0* ≤ *e* **by** *auto*
hence *1* ≤ *pow2 e* **unfolding** *pow2-def* **by** *auto*
from *mult-mono*[*OF* ‹*1* ≤ *real m*› *this* ‹*0* ≤ *real m*›]
have *1* ≤ *Float m e* **by** (*simp add*: *le-float-def real-of-float-simp*)
thus *False* **using** ‹*Float m e* < *1*› **unfolding** *less-float-def le-float-def* **by** *auto*
qed
qed

lemma *float-less1-mantissa-bound*: **assumes** *0* < *Float m e Float m e* < *1* **shows** *m* < *2*^(*nat* (−*e*))
proof −
have *e* < *0* **using** *float-pos-less1-e-neg assms* **by** *auto*
have ⋀*x*. (*0*::*real*) < *2*^*x* **by** *auto*
have *real m* < *2*^(*nat* (−*e*)) **using** ‹*Float m e* < *1*›

```
    unfolding less-float-def real-of-float-neg-exp[OF ‹e < 0›] real-of-float-1
        real-mult-less-iff1[of - - 1, OF ‹0 < 2^(nat (−e))›, symmetric]
        mult-assoc by auto
  thus ?thesis unfolding real-of-int-less-iff[symmetric] by auto
qed

function bitlen :: int ⇒ int where
bitlen 0 = 0 |
bitlen −1 = 1 |
0 < x ⟹ bitlen x = 1 + (bitlen (x div 2)) |
x < −1 ⟹ bitlen x = 1 + (bitlen (x div 2))
  apply (case-tac x = 0 ∨ x = −1 ∨ x < −1 ∨ x > 0)
  apply auto
  done
termination by (relation measure (nat o abs), auto)

lemma bitlen-ge0: 0 ≤ bitlen x by (induct x rule: bitlen.induct, auto)
lemma bitlen-ge1: x ≠ 0 ⟹ 1 ≤ bitlen x by (induct x rule: bitlen.induct, auto
simp add: bitlen-ge0)

lemma bitlen-bounds': assumes 0 < x shows 2^nat (bitlen x − 1) ≤ x ∧ x + 1
≤ 2^nat (bitlen x) (is ?P x)
  using ‹0 < x›
proof (induct x rule: bitlen.induct)
  fix x
  assume 0 < x and hyp: 0 < x div 2 ⟹ ?P (x div 2) hence 0 ≤ x and x ≠
0 by auto
  { fix x have 0 ≤ 1 + bitlen x using bitlen-ge0[of x] by auto } note gt0-pls1
= this

  have 0 < (2::int) by auto

  show ?P x
  proof (cases x = 1)
    case True show ?P x unfolding True by auto
  next
    case False hence 2 ≤ x using ‹0 < x› ‹x ≠ 1› by auto
    hence 2 div 2 ≤ x div 2 by (rule zdiv-mono1, auto)
    hence 0 < x div 2 and x div 2 ≠ 0 by auto
    hence bitlen-s1-ge0: 0 ≤ bitlen (x div 2) − 1 using bitlen-ge1[OF ‹x div 2 ≠
0›] by auto

    { from hyp[OF ‹0 < x div 2›]
      have 2 ^ nat (bitlen (x div 2) − 1) ≤ x div 2 by auto
    hence 2 ^ nat (bitlen (x div 2) − 1) * 2 ≤ x div 2 * 2 by (rule mult-right-mono,
auto)
      also have ... ≤ x using ‹0 < x› by auto
    finally have 2^nat (1 + bitlen (x div 2) − 1) ≤ x unfolding power-Suc2[symmetric]
Suc-nat-eq-nat-zadd1[OF bitlen-s1-ge0] by auto
```

```
    } moreover
    { have x + 1 ≤ x - x mod 2 + 2
      proof -
        have x mod 2 < 2 using ‹0 < x› by auto
        hence x < x - x mod 2 +  2 unfolding algebra-simps by auto
        thus ?thesis by auto
      qed
      also have x - x mod 2 + 2 = (x div 2 + 1) * 2 unfolding algebra-simps
using ‹0 < x› zdiv-zmod-equality2[of x 2 0] by auto
      also have ... ≤ 2^nat (bitlen (x div 2)) * 2 using hyp[OF ‹0 < x div 2›,
THEN conjunct2] by (rule mult-right-mono, auto)
    also have ... = 2^(1 + nat (bitlen (x div 2))) unfolding power-Suc2[symmetric]
by auto
      finally have x + 1 ≤ 2^(1 + nat (bitlen (x div 2))) .
    }
    ultimately show ?thesis
    unfolding bitlen.simps(3)[OF ‹0 < x›] nat-add-distrib[OF zero-le-one bitlen-ge0]
      unfolding add-commute nat-add-distrib[OF zero-le-one gt0-pls1]
      by auto
  qed
next
  fix x :: int assume x < -1 and 0 < x hence False by auto
  thus ?P x by auto
qed auto

lemma bitlen-bounds: assumes 0 < x shows 2^nat (bitlen x - 1) ≤ x ∧ x <
2^nat (bitlen x)
  using bitlen-bounds'[OF ‹0<x›] by auto

lemma bitlen-div: assumes 0 < m shows 1 ≤ real m / 2^nat (bitlen m - 1)
and real m / 2^nat (bitlen m - 1) < 2
proof -
  let ?B = 2^nat(bitlen m - 1)

  have ?B ≤ m using bitlen-bounds[OF ‹0 <m›] ..
  hence 1 * ?B ≤ real m unfolding real-of-int-le-iff[symmetric] by auto
  thus 1 ≤ real m / ?B by auto

  have m ≠ 0 using assms by auto
  have 0 ≤ bitlen m - 1 using bitlen-ge1[OF ‹m ≠ 0›] by auto

  have m < 2^nat(bitlen m) using bitlen-bounds[OF ‹0 <m›] ..
  also have ... = 2^nat(bitlen m - 1 + 1) using bitlen-ge1[OF ‹m ≠ 0›] by
auto
  also have ... = ?B * 2 unfolding nat-add-distrib[OF ‹0 ≤ bitlen m - 1›
zero-le-one] by auto
  finally have real m < 2 * ?B unfolding real-of-int-less-iff[symmetric] by auto
  hence real m / ?B < 2 * ?B / ?B by (rule divide-strict-right-mono, auto)
  thus real m / ?B < 2 by auto
```

qed

lemma *float-gt1-scale*: **assumes** $1 \leq Float\ m\ e$
shows $0 \leq e + (bitlen\ m - 1)$
proof (*cases* $0 \leq e$)
have $0 < Float\ m\ e$ **using** *assms* **unfolding** *less-float-def le-float-def* **by** *auto*
hence $0 < m$ **using** *float-pos-m-pos* **by** *auto*
hence $m \neq 0$ **by** *auto*
case *True* **with** *bitlen-ge1*[*OF* ‹$m \neq 0$›] **show** *?thesis* **by** *auto*
next
have $0 < Float\ m\ e$ **using** *assms* **unfolding** *less-float-def le-float-def* **by** *auto*
hence $0 < m$ **using** *float-pos-m-pos* **by** *auto*
hence $m \neq 0$ **and** $1 < (2::int)$ **by** *auto*
case *False* **let** *?S* = $2\hat{}(nat\ (-e))$
have $1 \leq real\ m * inverse\ ?S$ **using** *assms* **unfolding** *le-float-def real-of-float-nge0-exp*[*OF False*] **by** *auto*
hence $1 * ?S \leq real\ m * inverse\ ?S * ?S$ **by** (*rule mult-right-mono*, *auto*)
hence $?S \leq real\ m$ **unfolding** *mult-assoc* **by** *auto*
hence $?S \leq m$ **unfolding** *real-of-int-le-iff*[*symmetric*] **by** *auto*
from *this bitlen-bounds*[*OF* ‹$0 < m$›, *THEN conjunct2*]
have $nat\ (-e) < (nat\ (bitlen\ m))$ **unfolding** *power-strict-increasing-iff*[*OF* ‹$1 < 2$›, *symmetric*] **by** (*rule order-le-less-trans*)
hence $-e < bitlen\ m$ **using** *False bitlen-ge0* **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *normalized-float*: **assumes** $m \neq 0$ **shows** $real\ (Float\ m\ (-\ (bitlen\ m - 1))) = real\ m\ /\ 2\hat{}nat\ (bitlen\ m - 1)$
proof (*cases* $-\ (bitlen\ m - 1) = 0$)
case *True* **show** *?thesis* **unfolding** *real-of-float-simp pow2-def* **using** *True* **by** *auto*
next
case *False* **hence** *P*: $\neg\ 0 \leq -\ (bitlen\ m - 1)$ **using** *bitlen-ge1*[*OF* ‹$m \neq 0$›] **by** *auto*
show *?thesis* **unfolding** *real-of-float-nge0-exp*[*OF P*] *divide-inverse* **by** *auto*
qed

lemma *bitlen-Pls*: $bitlen\ (Int.Pls) = Int.Pls$ **by** (*subst Pls-def*, *subst Pls-def*, *simp*)

lemma *bitlen-Min*: $bitlen\ (Int.Min) = Int.Bit1\ Int.Pls$ **by** (*subst Min-def*, *simp add*: *Bit1-def*)

lemma *bitlen-B0*: $bitlen\ (Int.Bit0\ b) = (if\ iszero\ b\ then\ Int.Pls\ else\ Int.succ\ (bitlen\ b))$
apply (*auto simp add*: *iszero-def succ-def*)
apply (*simp add*: *Bit0-def Pls-def*)
apply (*subst Bit0-def*)
apply *simp*

```
  apply (subgoal-tac 0 < 2 * b ∨ 2 * b < -1)
  apply auto
  done

lemma bitlen-B1: bitlen (Int.Bit1 b) = (if iszero (Int.succ b) then Int.Bit1 Int.Pls
else Int.succ (bitlen b))
proof -
  have h: ! x. (2*x + 1) div 2 = (x::int)
    by arith
  show ?thesis
    apply (auto simp add: iszero-def succ-def)
    apply (subst Bit1-def)+
    apply simp
    apply (subgoal-tac 2 * b + 1 = -1)
    apply (simp only:)
    apply simp-all
    apply (subst Bit1-def)
    apply simp
    apply (subgoal-tac 0 < 2 * b + 1 ∨ 2 * b + 1 < -1)
    apply (auto simp add: h)
    done
qed

lemma bitlen-number-of: bitlen (number-of w) = number-of (bitlen w)
  by (simp add: number-of-is-id)

lemma [code]: bitlen x =
     (if x = 0  then 0
 else if x = -1 then 1
                else (1 + (bitlen (x div 2))))
  by (cases x = 0 ∨ x = -1 ∨ 0 < x) auto

definition lapprox-posrat :: nat ⇒ int ⇒ int ⇒ float
where
  lapprox-posrat prec x y =
   (let
       l = nat (int prec + bitlen y - bitlen x) ;
       d = (x * 2^l) div y
    in normfloat (Float d (- (int l))))

lemma pow2-minus: pow2 (-x) = inverse (pow2 x)
  unfolding pow2-neg[of -x] by auto

lemma lapprox-posrat:
  assumes x: 0 ≤ x
  and y: 0 < y
  shows real (lapprox-posrat prec x y) ≤ real x / real y
proof -
  let ?l = nat (int prec + bitlen y - bitlen x)
```

have *real* $(x * 2\hat{}?l \ div\ y) * inverse\ (2\hat{}?l) \leq (real\ (x * 2\hat{}?l) / real\ y) * inverse\ (2\hat{}?l)$
by (*rule mult-right-mono*, *fact real-of-int-div4*, *simp*)
also have $\ldots \leq (real\ x / real\ y) * 2\hat{}?l * inverse\ (2\hat{}?l)$ **by** *auto*
finally have *real* $(x * 2\hat{}?l \ div\ y) * inverse\ (2\hat{}?l) \leq real\ x / real\ y$ **unfolding** *mult-assoc* **by** *auto*
thus *?thesis* **unfolding** *lapprox-posrat-def Let-def normfloat real-of-float-simp*
unfolding *pow2-minus pow2-int minus-minus* **.**
qed

lemma *real-of-int-div-mult*:
fixes *x y c* :: *int* **assumes** $0 < y$ **and** $0 < c$
shows *real* $(x\ div\ y) \leq real\ (x * c\ div\ y) * inverse\ (real\ c)$
proof −
have $c * (x\ div\ y) + 0 \leq c * x\ div\ y$ **unfolding** *zdiv-zmult1-eq*[*of c x y*]
by (*rule zadd-left-mono*,
auto intro!: *mult-nonneg-nonneg*
simp add: *pos-imp-zdiv-nonneg-iff*[*OF* ‹$0 < y$›] ‹$0 < c$›[*THEN less-imp-le*] *pos-mod-sign*[*OF* ‹$0 < y$›])
hence *real* $(x\ div\ y) * real\ c \leq real\ (x * c\ div\ y)$
unfolding *real-of-int-mult*[*symmetric*] *real-of-int-le-iff zmult-commute* **by** *auto*
hence *real* $(x\ div\ y) * real\ c * inverse\ (real\ c) \leq real\ (x * c\ div\ y) * inverse\ (real\ c)$
using ‹$0 < c$› **by** *auto*
thus *?thesis* **unfolding** *mult-assoc* **using** ‹$0 < c$› **by** *auto*
qed

lemma *lapprox-posrat-bottom*: **assumes** $0 < y$
shows *real* $(x\ div\ y) \leq real\ (lapprox\text{-}posrat\ n\ x\ y)$
proof −
have *pow*: $\bigwedge x.\ (0::int) < 2\hat{}x$ **by** *auto*
show *?thesis*
unfolding *lapprox-posrat-def Let-def real-of-float-add normfloat real-of-float-simp pow2-minus pow2-int*
using *real-of-int-div-mult*[*OF* ‹$0 < y$› *pow*] **by** *auto*
qed

lemma *lapprox-posrat-nonneg*: **assumes** $0 \leq x$ **and** $0 < y$
shows $0 \leq real\ (lapprox\text{-}posrat\ n\ x\ y)$
proof −
show *?thesis*
unfolding *lapprox-posrat-def Let-def real-of-float-add normfloat real-of-float-simp pow2-minus pow2-int*
using *pos-imp-zdiv-nonneg-iff*[*OF* ‹$0 < y$›] *assms* **by** (*auto intro*!: *mult-nonneg-nonneg*)
qed

definition *rapprox-posrat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where

```
  rapprox-posrat prec x y = (let
     l = nat (int prec + bitlen y − bitlen x) ;
     X = x * 2^l ;
     d = X div y ;
     m = X mod y
   in normfloat (Float (d + (if m = 0 then 0 else 1)) (− (int l))))

lemma rapprox-posrat:
  assumes x: 0 ≤ x
  and y: 0 < y
  shows real x / real y ≤ real (rapprox-posrat prec x y)
proof −
  let ?l = nat (int prec + bitlen y − bitlen x) let ?X = x * 2^?l
  show ?thesis
  proof (cases ?X mod y = 0)
    case True hence y ≠ 0 and y dvd ?X using ‹0 < y› by auto
    from real-of-int-div[OF this]
    have real (?X div y) * inverse (2 ^ ?l) = real ?X / real y * inverse (2 ^ ?l)
by auto
    also have ... = real x / real y * (2^?l * inverse (2^?l)) by auto
    finally have real (?X div y) * inverse (2^?l) = real x / real y by auto
    thus ?thesis unfolding rapprox-posrat-def Let-def normfloat if-P[OF True]
      unfolding real-of-float-simp pow2-minus pow2-int minus-minus by auto
  next
    case False
    have 0 ≤ real y and real y ≠ 0 using ‹0 < y› by auto
    have 0 ≤ real y * 2^?l by (rule mult-nonneg-nonneg, rule ‹0 ≤ real y›, auto)

    have ?X = y * (?X div y) + ?X mod y by auto
     also have ... ≤ y * (?X div y) + y by (rule add-mono, auto simp add:
pos-mod-bound[OF ‹0 < y›, THEN less-imp-le])
    also have ... = y * (?X div y + 1) unfolding zadd-zmult-distrib2 by auto
   finally have real ?X ≤ real y * real (?X div y + 1) unfolding real-of-int-le-iff
real-of-int-mult[symmetric] .
    hence real ?X / (real y * 2^?l) ≤ real y * real (?X div y + 1) / (real y * 2^?l)

      by (rule divide-right-mono, simp only: ‹0 ≤ real y * 2^?l›)
    also have ... = real y * real (?X div y + 1) / real y / 2^?l by auto
  also have ... = real (?X div y + 1) * inverse (2^?l) unfolding nonzero-mult-divide-cancel-left[OF
‹real y ≠ 0›]
      unfolding divide-inverse ..
  finally show ?thesis unfolding rapprox-posrat-def Let-def normfloat real-of-float-simp
if-not-P[OF False]
      unfolding pow2-minus pow2-int minus-minus by auto
  qed
qed

lemma rapprox-posrat-le1: assumes 0 ≤ x and 0 < y and x ≤ y
  shows real (rapprox-posrat n x y) ≤ 1
```

proof −
let *?l* = *nat* (*int n* + *bitlen y* − *bitlen x*) **let** *?X* = *x* ∗ *2^?l*
show *?thesis*
proof (*cases ?X mod y* = *0*)
case *True* **hence** *y* ≠ *0* **and** *y dvd ?X* **using** ‹*0* < *y*› **by** *auto*
from *real-of-int-div*[*OF this*]
have *real* (*?X div y*) ∗ *inverse* (*2* ^ *?l*) = *real ?X* / *real y* ∗ *inverse* (*2* ^ *?l*)
by *auto*
also have ... = *real x* / *real y* ∗ (*2^?l* ∗ *inverse* (*2^?l*)) **by** *auto*
finally have *real* (*?X div y*) ∗ *inverse* (*2^?l*) = *real x* / *real y* **by** *auto*
also have *real x* / *real y* ≤ *1* **using** ‹*0* ≤ *x*› **and** ‹*0* < *y*› **and** ‹*x* ≤ *y*› **by** *auto*
finally show *?thesis* **unfolding** *rapprox-posrat-def Let-def normfloat if-P*[*OF True*]
unfolding *real-of-float-simp pow2-minus pow2-int minus-minus* **by** *auto*
next
case *False*
have *x* ≠ *y*
proof (*rule ccontr*)
assume ¬ *x* ≠ *y* **hence** *x* = *y* **by** *auto*
have *?X mod y* = *0* **unfolding** ‹*x* = *y*› **using** *mod-mult-self1-is-0* **by** *auto*
thus *False* **using** *False* **by** *auto*
qed
hence *x* < *y* **using** ‹*x* ≤ *y*› **by** *auto*
hence *real x* / *real y* < *1* **using** ‹*0* < *y*› **and** ‹*0* ≤ *x*› **by** *auto*

from *real-of-int-div4*[*of ?X y*]
have *real* (*?X div y*) ≤ (*real x* / *real y*) ∗ *2^?l* **unfolding** *real-of-int-mult times-divide-eq-left real-of-int-power real-number-of* **.**
also have ... < *1* ∗ *2^?l* **using** ‹*real x* / *real y* < *1*› **by** (*rule mult-strict-right-mono*, *auto*)
finally have *?X div y* < *2^?l* **unfolding** *real-of-int-less-iff*[*of* - *2^?l*, *symmetric*] **by** *auto*
hence *?X div y* + *1* ≤ *2^?l* **by** *auto*
hence *real* (*?X div y* + *1*) ∗ *inverse* (*2^?l*) ≤ *2^?l* ∗ *inverse* (*2^?l*)
unfolding *real-of-int-le-iff*[*of* - *2^?l*, *symmetric*] *real-of-int-power real-number-of*
by (*rule mult-right-mono*, *auto*)
hence *real* (*?X div y* + *1*) ∗ *inverse* (*2^?l*) ≤ *1* **by** *auto*
thus *?thesis* **unfolding** *rapprox-posrat-def Let-def normfloat real-of-float-simp if-not-P*[*OF False*]
unfolding *pow2-minus pow2-int minus-minus* **by** *auto*
qed
qed

lemma *zdiv-greater-zero*: **fixes** *a b* :: *int* **assumes** *0* < *a* **and** *a* ≤ *b*
shows *0* < *b div a*
proof (*rule ccontr*)
have *0* ≤ *b* **using** *assms* **by** *auto*
assume ¬ *0* < *b div a* **hence** *b div a* = *0* **using** ‹*0* ≤ *b*›[*unfolded pos-imp-zdiv-nonneg-iff*[*OF* ‹*0*<*a*›, *of b*, *symmetric*]] **by** *auto*

```
  have b = a * (b div a) + b mod a by auto
  hence b = b mod a unfolding ‹b div a = 0› by auto
  hence b < a using ‹0 < a›[THEN pos-mod-bound, of b] by auto
  thus False using ‹a ≤ b› by auto
qed

lemma rapprox-posrat-less1: assumes 0 ≤ x and 0 < y and 2 * x < y and 0
< n
  shows real (rapprox-posrat n x y) < 1
proof (cases x = 0)
   case True thus ?thesis unfolding rapprox-posrat-def True Let-def normfloat
real-of-float-simp by auto
next
  case False hence 0 < x using ‹0 ≤ x› by auto
  hence x < y using assms by auto

  let ?l = nat (int n + bitlen y − bitlen x) let ?X = x * 2^?l
  show ?thesis
  proof (cases ?X mod y = 0)
    case True hence y ≠ 0 and y dvd ?X using ‹0 < y› by auto
    from real-of-int-div[OF this]
    have real (?X div y) * inverse (2 ^ ?l) = real ?X / real y * inverse (2 ^ ?l)
by auto
    also have ... = real x / real y * (2^?l * inverse (2^?l)) by auto
    finally have real (?X div y) * inverse (2^?l) = real x / real y by auto
   also have real x / real y < 1 using ‹0 ≤ x› and ‹0 < y› and ‹x < y› by auto
   finally show ?thesis unfolding rapprox-posrat-def Let-def normfloat real-of-float-simp
if-P[OF True]
     unfolding pow2-minus pow2-int minus-minus by auto
  next
    case False
    hence (real x / real y) < 1 / 2 using ‹0 < y› and ‹0 ≤ x› ‹2 * x < y› by
auto

    have 0 < ?X div y
    proof −
      have 2^nat (bitlen x − 1) ≤ y and y < 2^nat (bitlen y)
        using bitlen-bounds[OF ‹0 < x›, THEN conjunct1] bitlen-bounds[OF ‹0 <
y›, THEN conjunct2] ‹x < y› by auto
      hence (2::int)^nat (bitlen x − 1) < 2^nat (bitlen y) by (rule order-le-less-trans)
      hence bitlen x ≤ bitlen y by auto
      hence len-less: nat (bitlen x − 1) ≤ nat (int (n − 1) + bitlen y) by auto

      have x ≠ 0 and y ≠ 0 using ‹0 < x› ‹0 < y› by auto

      have exp-eq: nat (int (n − 1) + bitlen y) − nat (bitlen x − 1) = ?l
        using ‹bitlen x ≤ bitlen y› bitlen-ge1[OF ‹x ≠ 0›] bitlen-ge1[OF ‹y ≠ 0›]
‹0 < n› by auto
```

have $y * 2\hat{}nat\ (bitlen\ x - 1) \leq y * x$
using *bitlen-bounds*[*OF* ‹$0 < x$›, *THEN conjunct1*] ‹$0 < y$›[*THEN less-imp-le*]
by (*rule mult-left-mono*)
also have $\ldots \leq 2\hat{}nat\ (bitlen\ y) * x$ **using** *bitlen-bounds*[*OF* ‹$0 < y$›, *THEN conjunct2*, *THEN less-imp-le*] ‹$0 \leq x$› **by** (*rule mult-right-mono*)
also have $\ldots \leq x * 2\hat{}nat\ (int\ (n - 1) + bitlen\ y)$ **unfolding** *mult-commute*[*of x*] **by** (*rule mult-right-mono*, *auto simp add*: ‹$0 \leq x$›)
finally have $real\ y * 2\hat{}nat\ (bitlen\ x - 1) * inverse\ (2\hat{}nat\ (bitlen\ x - 1)) \leq real\ x * 2\hat{}nat\ (int\ (n - 1) + bitlen\ y) * inverse\ (2\hat{}nat\ (bitlen\ x - 1))$
unfolding *real-of-int-le-iff*[*symmetric*] **by** *auto*
hence $real\ y \leq real\ x * (2\hat{}nat\ (int\ (n - 1) + bitlen\ y) / (2\hat{}nat\ (bitlen\ x - 1)))$
unfolding *mult-assoc divide-inverse* **by** *auto*
also have $\ldots = real\ x * (2\hat{}(nat\ (int\ (n - 1) + bitlen\ y) - nat\ (bitlen\ x - 1)))$ **using** *power-diff*[*of 2*::*real*, *OF - len-less*] **by** *auto*
finally have $y \leq x * 2\hat{}?l$ **unfolding** *exp-eq* **unfolding** *real-of-int-le-iff*[*symmetric*] **by** *auto*
thus *?thesis* **using** *zdiv-greater-zero*[*OF* ‹$0 < y$›] **by** *auto*
qed

from *real-of-int-div4*[*of ?X y*]
have $real\ (?X\ div\ y) \leq (real\ x\ /\ real\ y) * 2\hat{}?l$ **unfolding** *real-of-int-mult times-divide-eq-left real-of-int-power real-number-of* **.**
also have $\ldots < 1/2 * 2\hat{}?l$ **using** ‹$real\ x\ /\ real\ y < 1/2$› **by** (*rule mult-strict-right-mono*, *auto*)
finally have $?X\ div\ y * 2 < 2\hat{}?l$ **unfolding** *real-of-int-less-iff*[*of - 2*$\hat{}$*?l*, *symmetric*] **by** *auto*
hence $?X\ div\ y + 1 < 2\hat{}?l$ **using** ‹$0 < ?X\ div\ y$› **by** *auto*
hence $real\ (?X\ div\ y + 1) * inverse\ (2\hat{}?l) < 2\hat{}?l * inverse\ (2\hat{}?l)$
unfolding *real-of-int-less-iff*[*of - 2*$\hat{}$*?l*, *symmetric*] *real-of-int-power real-number-of*
by (*rule mult-strict-right-mono*, *auto*)
hence $real\ (?X\ div\ y + 1) * inverse\ (2\hat{}?l) < 1$ **by** *auto*
thus *?thesis* **unfolding** *rapprox-posrat-def Let-def normfloat real-of-float-simp if-not-P*[*OF False*]
unfolding *pow2-minus pow2-int minus-minus* **by** *auto*
qed
qed

lemma *approx-rat-pattern*: **fixes** P **and** $ps :: nat * int * int$
assumes Y: $\bigwedge y\ prec\ x.\ [\![y = 0;\ ps = (prec,\ x,\ 0)]\!] \Longrightarrow P$
and A: $\bigwedge x\ y\ prec.\ [\![0 \leq x;\ 0 < y;\ ps = (prec,\ x,\ y)]\!] \Longrightarrow P$
and B: $\bigwedge x\ y\ prec.\ [\![x < 0;\ 0 < y;\ ps = (prec,\ x,\ y)]\!] \Longrightarrow P$
and C: $\bigwedge x\ y\ prec.\ [\![x < 0;\ y < 0;\ ps = (prec,\ x,\ y)]\!] \Longrightarrow P$
and D: $\bigwedge x\ y\ prec.\ [\![0 \leq x;\ y < 0;\ ps = (prec,\ x,\ y)]\!] \Longrightarrow P$
shows P
proof −
obtain *prec x y* **where** [*simp*]: $ps = (prec,\ x,\ y)$ **by** (*cases ps*, *auto*)
from Y **have** $y = 0 \Longrightarrow P$ **by** *auto*
moreover { **assume** $0 < y$ **have** P **proof** (*cases* $0 \leq x$) **case** *True* **with** A

and ‹$0 < y$› **show** P **by** *auto* **next case** *False* **with** B **and** ‹$0 < y$› **show** P **by** *auto* **qed** **}**
moreover **{** **assume** $y < 0$ **have** P **proof** (*cases* $0 \leq x$) **case** *True* **with** D **and** ‹$y < 0$› **show** P **by** *auto* **next case** *False* **with** C **and** ‹$y < 0$› **show** P **by** *auto* **qed** **}**
ultimately show P **by** (*cases* $y = 0 \vee 0 < y \vee y < 0$, *auto*)
qed

function *lapprox-rat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
$y = 0 \Longrightarrow$ *lapprox-rat prec x y* = *0*
| $0 \leq x \Longrightarrow 0 < y \Longrightarrow$ *lapprox-rat prec x y* = *lapprox-posrat prec x y*
| $x < 0 \Longrightarrow 0 < y \Longrightarrow$ *lapprox-rat prec x y* = − (*rapprox-posrat prec* (−*x*) *y*)
| $x < 0 \Longrightarrow y < 0 \Longrightarrow$ *lapprox-rat prec x y* = *lapprox-posrat prec* (−*x*) (−*y*)
| $0 \leq x \Longrightarrow y < 0 \Longrightarrow$ *lapprox-rat prec x y* = − (*rapprox-posrat prec x* (−*y*))
apply *simp-all* **by** (*rule approx-rat-pattern*)
termination by *lexicographic-order*

lemma *compute-lapprox-rat*[*code*]:
lapprox-rat prec x y = (*if y* = *0 then 0 else if* $0 \leq x$ *then* (*if* $0 < y$ *then lapprox-posrat prec x y else* − (*rapprox-posrat prec x* (−*y*)))
else (*if* $0 < y$ *then* − (*rapprox-posrat prec* (−*x*) *y*) *else lapprox-posrat prec* (−*x*) (−*y*)))
by *auto*

lemma *lapprox-rat*: *real* (*lapprox-rat prec x y*) ≤ *real x* / *real y*
proof −
have *h*[*rule-format*]: ! *a b b'*. $b' \leq b \longrightarrow a \leq b' \longrightarrow a \leq$ (*b*::*real*) **by** *auto*
show *?thesis*
apply (*case-tac y* = *0*)
apply *simp*
apply (*case-tac* $0 \leq x \wedge 0 < y$)
apply (*simp add*: *lapprox-posrat*)
apply (*case-tac* $x < 0 \wedge 0 < y$)
apply *simp*
apply (*subst minus-le-iff*)
apply (*rule h*[*OF rapprox-posrat*])
apply (*simp-all*)
apply (*case-tac* $x < 0 \wedge y < 0$)
apply *simp*
apply (*rule h*[*OF - lapprox-posrat*])
apply (*simp-all*)
apply (*case-tac* $0 \leq x \wedge y < 0$)
apply (*simp*)
apply (*subst minus-le-iff*)
apply (*rule h*[*OF rapprox-posrat*])
apply *simp-all*
apply *arith*
done

qed

lemma *lapprox-rat-bottom*: **assumes** $0 \leq x$ **and** $0 < y$
shows *real* $(x \ div \ y) \leq$ *real* $(lapprox\text{-}rat \ n \ x \ y)$
unfolding *lapprox-rat.simps*(2)[*OF assms*] **using** *lapprox-posrat-bottom*[*OF* ‹0<y›]
.

function *rapprox-rat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float*
where
$y = 0 \Longrightarrow rapprox\text{-}rat \ prec \ x \ y = 0$
| $0 \leq x \Longrightarrow 0 < y \Longrightarrow rapprox\text{-}rat \ prec \ x \ y = rapprox\text{-}posrat \ prec \ x \ y$
| $x < 0 \Longrightarrow 0 < y \Longrightarrow rapprox\text{-}rat \ prec \ x \ y = -\ (lapprox\text{-}posrat \ prec \ (-x) \ y)$
| $x < 0 \Longrightarrow y < 0 \Longrightarrow rapprox\text{-}rat \ prec \ x \ y = rapprox\text{-}posrat \ prec \ (-x) \ (-y)$
| $0 \leq x \Longrightarrow y < 0 \Longrightarrow rapprox\text{-}rat \ prec \ x \ y = -\ (lapprox\text{-}posrat \ prec \ x \ (-y))$
apply *simp-all* **by** (*rule approx-rat-pattern*)
termination by *lexicographic-order*

lemma *compute-rapprox-rat*[*code*]:
rapprox-rat prec x y = (*if y* = *0 then 0 else if* 0 ≤ *x then* (*if* 0 < *y then rapprox-posrat prec x y else* − (*lapprox-posrat prec x* (−*y*))) *else* (*if* 0 < *y then* − (*lapprox-posrat prec* (−*x*) *y*) *else rapprox-posrat prec* (−*x*) (−*y*)))
by *auto*

lemma *rapprox-rat*: *real x* / *real y* ≤ *real* (*rapprox-rat prec x y*)
proof −
have *h*[*rule-format*]: ! *a b b'*. *b'* ≤ *b* ⟶ *a* ≤ *b'* ⟶ *a* ≤ (*b*::*real*) **by** *auto*
show *?thesis*
apply (*case-tac y* = *0*)
apply *simp*
apply (*case-tac* 0 ≤ *x* ∧ 0 < *y*)
apply (*simp add*: *rapprox-posrat*)
apply (*case-tac x* < *0* ∧ 0 < *y*)
apply *simp*
apply (*subst le-minus-iff*)
apply (*rule h*[*OF - lapprox-posrat*])
apply (*simp-all*)
apply (*case-tac x* < *0* ∧ *y* < *0*)
apply *simp*
apply (*rule h*[*OF rapprox-posrat*])
apply (*simp-all*)
apply (*case-tac* 0 ≤ *x* ∧ *y* < *0*)
apply (*simp*)
apply (*subst le-minus-iff*)
apply (*rule h*[*OF - lapprox-posrat*])
apply *simp-all*
apply *arith*
done
qed

lemma *rapprox-rat-le1*: **assumes** $0 \leq x$ **and** $0 < y$ **and** $x \leq y$
shows *real* (*rapprox-rat n x y*) $\leq$ *1*
unfolding *rapprox-rat.simps*(*2*)[*OF* ‹$0 \leq x$› ‹$0 < y$›] **using** *rapprox-posrat-le1*[*OF assms*] **.**

lemma *rapprox-rat-neg*: **assumes** $x < 0$ **and** $0 < y$
shows *real* (*rapprox-rat n x y*) ≤ 0
unfolding *rapprox-rat.simps*(*3*)[*OF assms*] **using** *lapprox-posrat-nonneg*[*of* $-x$ *y n*] *assms* **by** *auto*

lemma *rapprox-rat-nonneg-neg*: **assumes** $0 \leq x$ **and** $y < 0$
shows *real* (*rapprox-rat n x y*) ≤ 0
unfolding *rapprox-rat.simps*(*5*)[*OF assms*] **using** *lapprox-posrat-nonneg*[*of x* $-y$ *n*] *assms* **by** *auto*

lemma *rapprox-rat-nonpos-pos*: **assumes** $x \leq 0$ **and** $0 < y$
shows *real* (*rapprox-rat n x y*) ≤ 0
proof (*cases* $x = 0$)
case *True* **hence** $0 \leq x$ **by** *auto* **show** *?thesis* **unfolding** *rapprox-rat.simps*(*2*)[*OF* ‹$0 \leq x$› ‹$0 < y$›]
unfolding *True rapprox-posrat-def Let-def* **by** *auto*
next
case *False* **hence** $x < 0$ **using** *assms* **by** *auto*
show *?thesis* **using** *rapprox-rat-neg*[*OF* ‹$x < 0$› ‹$0 < y$›] **.**
qed

fun *float-divl* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float*
where
float-divl prec (*Float m1 s1*) (*Float m2 s2*) =
(*let*
l = *lapprox-rat prec m1 m2*;
f = *Float 1* (*s1* − *s2*)
in
f ∗ *l*)

lemma *float-divl*: *real* (*float-divl prec x y*) $\leq$ *real x* / *real y*
proof −
from *float-split*[*of x*] **obtain** *mx sx* **where** *x*: *x* = *Float mx sx* **by** *auto*
from *float-split*[*of y*] **obtain** *my sy* **where** *y*: *y* = *Float my sy* **by** *auto*
have *real mx* / *real my* ≤ (*real mx* ∗ *pow2 sx* / (*real my* ∗ *pow2 sy*)) / (*pow2* (*sx* − *sy*))
apply (*case-tac my* = *0*)
apply *simp*
apply (*case-tac my* > *0*)
apply (*subst pos-le-divide-eq*)
apply *simp*
apply (*subst pos-le-divide-eq*)
apply (*simp add*: *mult-pos-pos*)

apply *simp*
apply (*subst pow2-add*[*symmetric*])
apply *simp*
apply (*subgoal-tac my* < *0*)
apply *auto*
apply (*simp add*: *field-simps*)
apply (*subst pow2-add*[*symmetric*])
apply (*simp add*: *field-simps*)
done
then have *real* (*lapprox-rat prec mx my*) ≤ (*real mx* ∗ *pow2 sx* / (*real my* ∗ *pow2 sy*)) / (*pow2* (*sx* − *sy*))
by (*rule order-trans*[*OF lapprox-rat*])
then have *real* (*lapprox-rat prec mx my*) ∗ *pow2* (*sx* − *sy*) ≤ *real mx* ∗ *pow2 sx* / (*real my* ∗ *pow2 sy*)
apply (*subst pos-le-divide-eq*[*symmetric*])
apply *simp-all*
done
then have *pow2* (*sx* − *sy*) ∗ *real* (*lapprox-rat prec mx my*) ≤ *real mx* ∗ *pow2 sx* / (*real my* ∗ *pow2 sy*)
by (*simp add*: *algebra-simps*)
then show *?thesis*
by (*simp add*: *x y Let-def real-of-float-simp*)
qed

lemma *float-divl-lower-bound*: **assumes** *0* ≤ *x* **and** *0* < *y* **shows** *0* ≤ *float-divl prec x y*
proof (*cases x*, *cases y*)
fix *xm xe ym ye* :: *int*
assume *x-eq*: *x* = *Float xm xe* **and** *y-eq*: *y* = *Float ym ye*
have *0* ≤ *xm* **using** ‹*0* ≤ *x*›[*unfolded x-eq le-float-def real-of-float-simp real-of-float-0 zero-le-mult-iff*] **by** *auto*
have *0* < *ym* **using** ‹*0* < *y*›[*unfolded y-eq less-float-def real-of-float-simp real-of-float-0 zero-less-mult-iff*] **by** *auto*

have ⋀*n*. *0* ≤ *real* (*Float 1 n*) **unfolding** *real-of-float-simp* **using** *zero-le-pow2* **by** *auto*
moreover have *0* ≤ *real* (*lapprox-rat prec xm ym*) **by** (*rule order-trans*[*OF - lapprox-rat-bottom*[*OF* ‹*0* ≤ *xm*› ‹*0* < *ym*›]], *auto simp add*: ‹*0* ≤ *xm*› *pos-imp-zdiv-nonneg-iff*[*OF* ‹*0* < *ym*›])
ultimately show *0* ≤ *float-divl prec x y*
unfolding *x-eq y-eq float-divl.simps Let-def le-float-def real-of-float-0* **by** (*auto intro*!: *mult-nonneg-nonneg*)
qed

lemma *float-divl-pos-less1-bound*: **assumes** *0* < *x* **and** *x* < *1* **and** *0* < *prec* **shows** *1* ≤ *float-divl prec 1 x*
proof (*cases x*)
case (*Float m e*)
from ‹*0* < *x*› ‹*x* < *1*› **have** *0* < *m e* < *0* **using** *float-pos-m-pos float-pos-less1-e-neg*

unfolding *Float* **by** *auto*
let *?b* = *nat* (*bitlen m*) **and** *?e* = *nat* (−*e*)
have *1* ≤ *m* **and** *m* ≠ *0* **using** ‹*0* < *m*› **by** *auto*
with *bitlen-bounds*[*OF* ‹*0* < *m*›] **have** *m* < *2*^*?b* **and** (*2*::*int*) ≤ *2*^*?b* **by** *auto*
hence *1* ≤ *bitlen m* **using** *power-le-imp-le-exp*[*of 2*::*int 1 ?b*] **by** *auto*
hence *pow-split*: *nat* (*int prec* + *bitlen m* − *1*) = (*prec* − *1*) + *?b* **using** ‹*0* < *prec*› **by** *auto*

have *pow-not0*: ⋀*x*. (*2*::*real*) ^ *x* ≠ *0* **by** *auto*

from *float-less1-mantissa-bound* ‹*0* < *x*› ‹*x* < *1*› *Float*
have *m* < *2*^*?e* **by** *auto*
with *bitlen-bounds*[*OF* ‹*0* < *m*›, *THEN conjunct1*]
have (*2*::*int*) ^ *nat* (*bitlen m* − *1*) < *2*^*?e* **by** (*rule order-le-less-trans*)
from *power-less-imp-less-exp*[*OF - this*]
have *bitlen m* ≤ − *e* **by** *auto*
hence (*2*::*real*) ^ *?b* ≤ *2*^*?e* **by** *auto*
hence (*2*::*real*) ^ *?b* * *inverse* (*2*^*?b*) ≤ *2*^*?e* * *inverse* (*2*^*?b*) **by** (*rule mult-right-mono*, *auto*)
hence (*1*::*real*) ≤ *2*^*?e* * *inverse* (*2*^*?b*) **by** *auto*
also
let *?d* = *real* (*2* ^ *nat* (*int prec* + *bitlen m* − *1*) *div m*) * *inverse* (*2* ^ *nat* (*int prec* + *bitlen m* − *1*))
{ have *2*^(*prec* − *1*) * *m* ≤ *2*^(*prec* − *1*) * *2*^*?b* **using** ‹*m* < *2*^*?b*›[*THEN less-imp-le*] **by** (*rule mult-left-mono*, *auto*)
also have ... = *2* ^ *nat* (*int prec* + *bitlen m* − *1*) **unfolding** *pow-split zpower-zadd-distrib* **by** *auto*
finally have *2*^(*prec* − *1*) * *m div m* ≤ *2* ^ *nat* (*int prec* + *bitlen m* − *1*) *div m* **using** ‹*0* < *m*› **by** (*rule zdiv-mono1*)
hence *2*^(*prec* − *1*) ≤ *2* ^ *nat* (*int prec* + *bitlen m* − *1*) *div m* **unfolding** *div-mult-self2-is-id*[*OF* ‹*m* ≠ *0*›] **.**
hence *2*^(*prec* − *1*) * *inverse* (*2* ^ *nat* (*int prec* + *bitlen m* − *1*)) ≤ *?d*
unfolding *real-of-int-le-iff*[*of 2*^(*prec* − *1*), *symmetric*] **by** *auto* **}**
from *mult-left-mono*[*OF this*[*unfolded pow-split power-add inverse-mult-distrib mult-assoc*[*symmetric*] *right-inverse*[*OF pow-not0*] *mult-1-left*], *of 2*^*?e*]
have *2*^*?e* * *inverse* (*2*^*?b*) ≤ *2*^*?e* * *?d* **unfolding** *pow-split power-add* **by** *auto*
finally have *1* ≤ *2*^*?e* * *?d* **.**

have *e-nat*: *0* − *e* = *int* (*nat* (−*e*)) **using** ‹*e* < *0*› **by** *auto*
have *bitlen 1* = *1* **using** *bitlen.simps* **by** *auto*

show *?thesis*
unfolding *one-float-def Float float-divl.simps Let-def lapprox-rat.simps*(*2*)[*OF zero-le-one* ‹*0* < *m*›] *lapprox-posrat-def* ‹*bitlen 1* = *1*›
unfolding *le-float-def real-of-float-mult normfloat real-of-float-simp pow2-minus pow2-int e-nat*
using ‹*1* ≤ *2*^*?e* * *?d*› **by** (*auto simp add*: *pow2-def*)
qed

fun *float-divr* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float*
where
float-divr prec (*Float m1 s1*) (*Float m2 s2*) =
(*let*
r = *rapprox-rat prec m1 m2*;
f = *Float 1* (*s1* − *s2*)
in
f ∗ *r*)

lemma *float-divr*: *real x* / *real y* ≤ *real* (*float-divr prec x y*)
proof −
from *float-split*[*of x*] **obtain** *mx sx* **where** *x*: *x* = *Float mx sx* **by** *auto*
from *float-split*[*of y*] **obtain** *my sy* **where** *y*: *y* = *Float my sy* **by** *auto*
have *real mx* / *real my* ≥ (*real mx* ∗ *pow2 sx* / (*real my* ∗ *pow2 sy*)) / (*pow2* (*sx* − *sy*))
apply (*case-tac my* = *0*)
apply *simp*
apply (*case-tac my* > *0*)
apply *auto*
apply (*subst pos-divide-le-eq*)
apply (*rule mult-pos-pos*)+
apply *simp-all*
apply (*subst pow2-add*[*symmetric*])
apply *simp*
apply (*subgoal-tac my* < *0*)
apply *auto*
apply (*simp add*: *field-simps*)
apply (*subst pow2-add*[*symmetric*])
apply (*simp add*: *field-simps*)
done
then have *real* (*rapprox-rat prec mx my*) ≥ (*real mx* ∗ *pow2 sx* / (*real my* ∗ *pow2 sy*)) / (*pow2* (*sx* − *sy*))
by (*rule order-trans*[*OF - rapprox-rat*])
then have *real* (*rapprox-rat prec mx my*) ∗ *pow2* (*sx* − *sy*) ≥ *real mx* ∗ *pow2 sx* / (*real my* ∗ *pow2 sy*)
apply (*subst pos-divide-le-eq*[*symmetric*])
apply *simp-all*
done
then show *?thesis*
by (*simp add*: *x y Let-def algebra-simps real-of-float-simp*)
qed

lemma *float-divr-pos-less1-lower-bound*: **assumes** *0* < *x* **and** *x* < *1* **shows** *1* ≤ *float-divr prec 1 x*
proof −
have *1* ≤ *1* / *real x* **using** ‹*0* < *x*› **and** ‹*x* < *1*› **unfolding** *less-float-def* **by** *auto*
also have ... ≤ *real* (*float-divr prec 1 x*) **using** *float-divr*[**where** *x*=*1* **and** *y*=*x*] **by** *auto*

finally show *?thesis* **unfolding** *le-float-def* **by** *auto*
qed

lemma *float-divr-nonpos-pos-upper-bound*: **assumes** $x \leq 0$ **and** $0 < y$ **shows** *float-divr prec x y* ≤ 0
proof (*cases x*, *cases y*)
fix *xm xe ym ye* :: *int*
assume *x-eq*: $x = Float\ xm\ xe$ **and** *y-eq*: $y = Float\ ym\ ye$
have $xm \leq 0$ **using** ‹$x \leq 0$›[*unfolded x-eq le-float-def real-of-float-simp real-of-float-0 mult-le-0-iff*] **by** *auto*
have $0 < ym$ **using** ‹$0 < y$›[*unfolded y-eq less-float-def real-of-float-simp real-of-float-0 zero-less-mult-iff*] **by** *auto*

have $\bigwedge n.\ 0 \leq real\ (Float\ 1\ n)$ **unfolding** *real-of-float-simp* **using** *zero-le-pow2* **by** *auto*
moreover have *real* (*rapprox-rat prec xm ym*) ≤ 0 **using** *rapprox-rat-nonpos-pos*[*OF* ‹$xm \leq 0$› ‹$0 < ym$›] **.**
ultimately show *float-divr prec x y* ≤ 0
unfolding *x-eq y-eq float-divr.simps Let-def le-float-def real-of-float-0 real-of-float-mult* **by** (*auto intro*!: *mult-nonneg-nonpos*)
qed

lemma *float-divr-nonneg-neg-upper-bound*: **assumes** $0 \leq x$ **and** $y < 0$ **shows** *float-divr prec x y* ≤ 0
proof (*cases x*, *cases y*)
fix *xm xe ym ye* :: *int*
assume *x-eq*: $x = Float\ xm\ xe$ **and** *y-eq*: $y = Float\ ym\ ye$
have $0 \leq xm$ **using** ‹$0 \leq x$›[*unfolded x-eq le-float-def real-of-float-simp real-of-float-0 zero-le-mult-iff*] **by** *auto*
have $ym < 0$ **using** ‹$y < 0$›[*unfolded y-eq less-float-def real-of-float-simp real-of-float-0 mult-less-0-iff*] **by** *auto*
hence $0 < -\ ym$ **by** *auto*

have $\bigwedge n.\ 0 \leq real\ (Float\ 1\ n)$ **unfolding** *real-of-float-simp* **using** *zero-le-pow2* **by** *auto*
moreover have *real* (*rapprox-rat prec xm ym*) ≤ 0 **using** *rapprox-rat-nonneg-neg*[*OF* ‹$0 \leq xm$› ‹$ym < 0$›] **.**
ultimately show *float-divr prec x y* ≤ 0
unfolding *x-eq y-eq float-divr.simps Let-def le-float-def real-of-float-0 real-of-float-mult* **by** (*auto intro*!: *mult-nonneg-nonpos*)
qed

primrec *round-down* :: *nat* ⇒ *float* ⇒ *float* **where**
round-down prec (*Float m e*) = (*let d* = *bitlen m* − *int prec in*
if 0 < *d then let P* = 2^*nat d* ; *n* = *m div P in Float n* (*e* + *d*)
else Float m e)

primrec *round-up* :: *nat* ⇒ *float* ⇒ *float* **where**
round-up prec (*Float m e*) = (*let d* = *bitlen m* − *int prec in*

if 0 < d then let P = 2^nat d ; n = m div P ; r = m mod P in Float (n + (if r = 0 then 0 else 1)) (e + d)
else Float m e)

lemma *round-up*: *real x ≤ real (round-up prec x)*
proof (*cases x*)
case (*Float m e*)
let *?d = bitlen m − int prec*
let *?p = (2::int) ^nat ?d*
have *0 < ?p* **by** *auto*
show *?thesis*
proof (*cases 0 < ?d*)
case *True*
hence *pow-d*: *pow2 ?d = real ?p* **using** *pow2-int*[*symmetric*] **by** *simp*
show *?thesis*
proof (*cases m mod ?p = 0*)
case *True*
have *m*: *m = m div ?p * ?p + 0* **unfolding** *True*[*symmetric*] **using** *zdiv-zmod-equality2*[**where** *k=0*, *unfolded monoid-add-class.add-0-right*, *symmetric*] **.**
have *real (Float m e) = real (Float (m div ?p) (e + ?d))* **unfolding** *real-of-float-simp arg-cong*[*OF m*, *of real*]
by (*auto simp add*: *pow2-add ‹0 < ?d› pow-d*)
thus *?thesis*
unfolding *Float round-up.simps Let-def if-P*[*OF ‹m mod ?p = 0›*] *if-P*[*OF ‹0 < ?d›*]
by *auto*
next
case *False*
have *m = m div ?p * ?p + m mod ?p* **unfolding** *zdiv-zmod-equality2*[**where** *k=0*, *unfolded monoid-add-class.add-0-right*] **..**
also have *... ≤ (m div ?p + 1) * ?p* **unfolding** *left-distrib zmult-1* **by** (*rule add-left-mono*, *rule pos-mod-bound*[*OF ‹0 < ?p›*, *THEN less-imp-le*])
finally have *real (Float m e) ≤ real (Float (m div ?p + 1) (e + ?d))* **unfolding** *real-of-float-simp add-commute*[*of e*]
unfolding *pow2-add mult-assoc*[*symmetric*] *real-of-int-le-iff*[*of m*, *symmetric*]
by (*auto intro*!: *mult-mono simp add*: *pow2-add ‹0 < ?d› pow-d*)
thus *?thesis*
unfolding *Float round-up.simps Let-def if-not-P*[*OF ‹¬ m mod ?p = 0›*] *if-P*[*OF ‹0 < ?d›*] **.**
qed
next
case *False*
show *?thesis*
unfolding *Float round-up.simps Let-def if-not-P*[*OF False*] **..**
qed
qed

lemma *round-down*: *real (round-down prec x) ≤ real x*

proof (*cases x*)
case (*Float m e*)
let *?d* = *bitlen m* − *int prec*
let *?p* = (*2*::*int*) ^ *nat ?d*
have *0* < *?p* **by** *auto*
show *?thesis*
proof (*cases 0* < *?d*)
case *True*
hence *pow-d*: *pow2 ?d* = *real ?p* **using** *pow2-int*[*symmetric*] **by** *simp*
have *m div ?p* ∗ *?p* ≤ *m div ?p* ∗ *?p* + *m mod ?p* **by** (*auto simp add*: *pos-mod-bound*[*OF* ‹*0* < *?p*›, *THEN less-imp-le*])
also have ... ≤ *m* **unfolding** *zdiv-zmod-equality2*[**where** *k*=*0*, *unfolded monoid-add-class.add-0-right*] **..**
finally have *real* (*Float* (*m div ?p*) (*e* + *?d*)) ≤ *real* (*Float m e*) **unfolding** *real-of-float-simp add-commute*[*of e*]
unfolding *pow2-add mult-assoc*[*symmetric*] *real-of-int-le-iff*[*of* - *m*, *symmetric*]
by (*auto intro*!: *mult-mono simp add*: *pow2-add* ‹*0* < *?d*› *pow-d*)
thus *?thesis*
unfolding *Float round-down.simps Let-def if-P*[*OF* ‹*0* < *?d*›] **.**
next
case *False*
show *?thesis*
unfolding *Float round-down.simps Let-def if-not-P*[*OF False*] **..**
qed
qed

definition *lb-mult* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **where**
lb-mult prec x y = (*case normfloat* (*x* ∗ *y*) *of Float m e* ⇒ *let*
l = *bitlen m* − *int prec*
in if l > *0 then Float* (*m div* (*2* ^ *nat l*)) (*e* + *l*)
else Float m e)

definition *ub-mult* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **where**
ub-mult prec x y = (*case normfloat* (*x* ∗ *y*) *of Float m e* ⇒ *let*
l = *bitlen m* − *int prec*
in if l > *0 then Float* (*m div* (*2* ^ *nat l*) + *1*) (*e* + *l*)
else Float m e)

lemma *lb-mult*: *real* (*lb-mult prec x y*) ≤ *real* (*x* ∗ *y*)
proof (*cases normfloat* (*x* ∗ *y*))
case (*Float m e*)
hence *odd m* ∨ (*m* = *0* ∧ *e* = *0*) **by** (*rule normfloat-imp-odd-or-zero*)
let *?l* = *bitlen m* − *int prec*
have *real* (*lb-mult prec x y*) ≤ *real* (*normfloat* (*x* ∗ *y*))
proof (*cases ?l* > *0*)
case *False* **thus** *?thesis* **unfolding** *lb-mult-def Float Let-def float.cases* **by** *auto*
next
case *True*
have *real* (*m div* *2* ^ (*nat ?l*)) ∗ *pow2 ?l* ≤ *real m*

```
    proof −
     have real (m div 2^(nat ?l)) * pow2 ?l = real (2^(nat ?l) * (m div 2^(nat ?l)))
unfolding real-of-int-mult real-of-int-power real-number-of unfolding pow2-int[symmetric]

       using ‹?l > 0› by auto
      also have ... ≤ real (2^(nat ?l) * (m div 2^(nat ?l)) + m mod 2^(nat ?l))
unfolding real-of-int-add by auto
      also have ... = real m unfolding zmod-zdiv-equality[symmetric] ..
      finally show ?thesis by auto
    qed
     thus ?thesis unfolding lb-mult-def Float Let-def float.cases if-P[OF True]
real-of-float-simp pow2-add mult-commute mult-assoc by auto
  qed
  also have ... = real (x * y) unfolding normfloat ..
  finally show ?thesis .
qed

lemma ub-mult: real (x * y) ≤ real (ub-mult prec x y)
proof (cases normfloat (x * y))
  case (Float m e)
  hence odd m ∨ (m = 0 ∧ e = 0) by (rule normfloat-imp-odd-or-zero)
  let ?l = bitlen m − int prec
  have real (x * y) = real (normfloat (x * y)) unfolding normfloat ..
  also have ... ≤ real (ub-mult prec x y)
  proof (cases ?l > 0)
    case False thus ?thesis unfolding ub-mult-def Float Let-def float.cases by
auto
  next
    case True
    have real m ≤ real (m div 2^(nat ?l) + 1) * pow2 ?l
    proof −
      have m mod 2^(nat ?l) < 2^(nat ?l) by (rule pos-mod-bound) auto
     hence mod-uneq: real (m mod 2^(nat ?l)) ≤ 1 * 2^(nat ?l) unfolding zmult-1
real-of-int-less-iff[symmetric] by auto

      have real m = real (2^(nat ?l) * (m div 2^(nat ?l)) + m mod 2^(nat ?l))
unfolding zmod-zdiv-equality[symmetric] ..
      also have ... = real (m div 2^(nat ?l)) * 2^(nat ?l) + real (m mod 2^(nat
?l)) unfolding real-of-int-add by auto
       also have ... ≤ (real (m div 2^(nat ?l)) + 1) * 2^(nat ?l) unfolding
left-distrib using mod-uneq by auto
      finally show ?thesis unfolding pow2-int[symmetric] using True by auto
    qed
     thus ?thesis unfolding ub-mult-def Float Let-def float.cases if-P[OF True]
real-of-float-simp pow2-add mult-commute mult-assoc by auto
  qed
  finally show ?thesis .
qed
```

primrec *float-abs* :: *float* ⇒ *float* **where**
float-abs (*Float m e*) = *Float* |*m*| *e*

instantiation *float* :: *abs* **begin**
definition *abs-float-def*: |*x*| = *float-abs x*
instance ..
end

lemma *real-of-float-abs*: *real* |*x* :: *float*| = |*real x*|
proof (*cases x*)
case (*Float m e*)
have |*real m*| * *pow2 e* = |*real m* * *pow2 e*| **unfolding** *abs-mult* **by** *auto*
thus *?thesis* **unfolding** *Float abs-float-def float-abs.simps real-of-float-simp* **by** *auto*
qed

primrec *floor-fl* :: *float* ⇒ *float* **where**
floor-fl (*Float m e*) = (*if* 0 ≤ *e then Float m e*
else Float (*m div* (2 ^ (*nat* (−*e*)))) 0)

lemma *floor-fl*: *real* (*floor-fl x*) ≤ *real x*
proof (*cases x*)
case (*Float m e*)
show *?thesis*
proof (*cases* 0 ≤ *e*)
case *False*
hence *me-eq*: *pow2* (−*e*) = *pow2* (*int* (*nat* (−*e*))) **by** *auto*
have *real* (*Float* (*m div* (2 ^ (*nat* (−*e*)))) 0) = *real* (*m div* 2 ^ (*nat* (−*e*)))
unfolding *real-of-float-simp* **by** *auto*
also have ... ≤ *real m* / *real* ((2::*int*) ^ (*nat* (−*e*))) **using** *real-of-int-div4* **.**
also have ... = *real m* * *inverse* (2 ^ (*nat* (−*e*))) **unfolding** *real-of-int-power real-number-of divide-inverse* **..**
also have ... = *real* (*Float m e*) **unfolding** *real-of-float-simp me-eq pow2-int pow2-neg*[*of e*] **..**
finally show *?thesis* **unfolding** *Float floor-fl.simps if-not-P*[*OF* ‹¬ 0 ≤ *e*›] **.**
next
case *True* **thus** *?thesis* **unfolding** *Float* **by** *auto*
qed
qed

lemma *floor-pos-exp*: **assumes** *floor*: *Float m e* = *floor-fl x* **shows** 0 ≤ *e*
proof (*cases x*)
case (*Float mx me*)
from *floor*[*unfolded Float floor-fl.simps*] **show** *?thesis* **by** (*cases* 0 ≤ *me*, *auto*)
qed

declare *floor-fl.simps*[*simp del*]

primrec *ceiling-fl* :: *float* ⇒ *float* **where**

ceiling-fl (Float m e) = (if 0 ≤ e then Float m e
else Float (m div (2 ^ (nat (−e))) + 1) 0)

lemma *ceiling-fl*: *real x ≤ real (ceiling-fl x)*
proof (*cases x*)
case (*Float m e*)
show *?thesis*
proof (*cases 0 ≤ e*)
case *False*
hence *me-eq*: *pow2 (−e) = pow2 (int (nat (−e)))* **by** *auto*
have *real (Float m e) = real m * inverse (2 ^ (nat (−e)))* **unfolding** *real-of-float-simp me-eq pow2-int pow2-neg*[*of e*] **..**
also have *... = real m / real ((2*::*int) ^ (nat (−e)))* **unfolding** *real-of-int-power real-number-of divide-inverse* **..**
also have *... ≤ 1 + real (m div 2 ^ (nat (−e)))* **using** *real-of-int-div3*[*unfolded diff-le-eq*] **.**
also have *... = real (Float (m div (2 ^ (nat (−e))) + 1) 0)* **unfolding** *real-of-float-simp* **by** *auto*
finally show *?thesis* **unfolding** *Float ceiling-fl.simps if-not-P*[*OF ‹¬ 0 ≤ e›*]
.
next
case *True* **thus** *?thesis* **unfolding** *Float* **by** *auto*
qed
qed

declare *ceiling-fl.simps*[*simp del*]

definition *lb-mod* :: *nat ⇒ float ⇒ float ⇒ float ⇒ float* **where**
*lb-mod prec x ub lb = x − ceiling-fl (float-divr prec x lb) * ub*

definition *ub-mod* :: *nat ⇒ float ⇒ float ⇒ float ⇒ float* **where**
*ub-mod prec x ub lb = x − floor-fl (float-divl prec x ub) * lb*

lemma *lb-mod*: **fixes** *k* :: *int* **assumes** *0 ≤ real x* **and** *real k * y ≤ real x* (**is** *?k * y ≤ ?x*)
assumes *0 < real lb real lb ≤ y* (**is** *?lb ≤ y*) *y ≤ real ub* (**is** *y ≤ ?ub*)
shows *real (lb-mod prec x ub lb) ≤ ?x − ?k * y*
proof −
have *?lb ≤ ?ub* **using** *assms* **by** *auto*
have *0 ≤ ?lb* **and** *?lb ≠ 0* **using** *assms* **by** *auto*
have *?k * y ≤ ?x* **using** *assms* **by** *auto*
also have *... ≤ ?x / ?lb * ?ub* **by** (*metis mult-left-mono*[*OF ‹?lb ≤ ?ub› ‹0 ≤ ?x›*] *divide-right-mono*[*OF - ‹0 ≤ ?lb›*] *times-divide-eq-left nonzero-mult-divide-cancel-right*[*OF ‹?lb ≠ 0›*])
also have *... ≤ real (ceiling-fl (float-divr prec x lb)) * ?ub* **by** (*metis mult-right-mono order-trans ‹0 ≤ ?lb› ‹?lb ≤ ?ub› float-divr ceiling-fl*)
finally show *?thesis* **unfolding** *lb-mod-def real-of-float-sub real-of-float-mult* **by** *auto*
qed

lemma *ub-mod*: **fixes** $k :: int$ **and** $x :: float$ **assumes** $0 \leq real\ x$ **and** $real\ x \leq real\ k * y$ (**is** $?x \leq ?k * y$)
assumes $0 < real\ lb\ real\ lb \leq y$ (**is** $?lb \leq y$) $y \leq real\ ub$ (**is** $y \leq ?ub$)
shows $?x - ?k * y \leq real\ (ub\text{-}mod\ prec\ x\ ub\ lb)$
proof −
have $?lb \leq ?ub$ **using** *assms* **by** *auto*
hence $0 \leq ?lb$ **and** $0 \leq ?ub$ **and** $?ub \neq 0$ **using** *assms* **by** *auto*
have $real\ (floor\text{-}fl\ (float\text{-}divl\ prec\ x\ ub)) * ?lb \leq ?x\ /\ ?ub * ?lb$ **by** (*metis mult-right-mono order-trans* ‹$0 \leq ?lb$› ‹$?lb \leq ?ub$› *float-divl floor-fl*)
also have $\ldots \leq ?x$ **by** (*metis mult-left-mono*[*OF* ‹$?lb \leq ?ub$› ‹$0 \leq ?x$›] *divide-right-mono*[*OF* - ‹$0 \leq ?ub$›] *times-divide-eq-left nonzero-mult-divide-cancel-right*[*OF* ‹$?ub \neq 0$›])
also have $\ldots \leq ?k * y$ **using** *assms* **by** *auto*
finally show *?thesis* **unfolding** *ub-mod-def real-of-float-sub real-of-float-mult* **by** *auto*
qed

lemma *le-float-def'*: $f \leq g = (case\ f - g\ of\ Float\ a\ b \Rightarrow a \leq 0)$
proof −
have *le-transfer*: $(f \leq g) = (real\ (f - g) \leq 0)$ **by** (*auto simp add*: *le-float-def*)
from *float-split*[*of* $f - g$] **obtain** $a\ b$ **where** *f-diff-g*: $f - g = Float\ a\ b$ **by** *auto*
with *le-transfer* **have** *le-transfer'*: $f \leq g = (real\ (Float\ a\ b) \leq 0)$ **by** *simp*
show *?thesis* **by** (*simp add*: *le-transfer' f-diff-g float-le-zero*)
qed

lemma *float-less-zero*:
$(real\ (Float\ a\ b) < 0) = (a < 0)$
apply (*auto simp add*: *mult-less-0-iff real-of-float-simp*)
done

lemma *less-float-def'*: $f < g = (case\ f - g\ of\ Float\ a\ b \Rightarrow a < 0)$
proof −
have *less-transfer*: $(f < g) = (real\ (f - g) < 0)$ **by** (*auto simp add*: *less-float-def*)
from *float-split*[*of* $f - g$] **obtain** $a\ b$ **where** *f-diff-g*: $f - g = Float\ a\ b$ **by** *auto*
with *less-transfer* **have** *less-transfer'*: $f < g = (real\ (Float\ a\ b) < 0)$ **by** *simp*
show *?thesis* **by** (*simp add*: *less-transfer' f-diff-g float-less-zero*)
qed

end

39 Formal-Power-Series: A formalization of formal power series

theory *Formal-Power-Series*
imports *Complex-Main Binomial*
begin

39.1 The type of formal power series

typedef (**open**) *'a fps* = {*f* :: *nat* ⇒ *'a*. *True*}
  morphisms *fps-nth Abs-fps*
  by *simp*

notation *fps-nth* (**infixl** $ *75*)

lemma *expand-fps-eq*: *p* = *q* ⟷ (∀ *n*. *p* $ *n* = *q* $ *n*)
  by (*simp add*: *fps-nth-inject* [*symmetric*] *expand-fun-eq*)

lemma *fps-ext*: (⋀*n*. *p* $ *n* = *q* $ *n*) ⟹ *p* = *q*
  by (*simp add*: *expand-fps-eq*)

lemma *fps-nth-Abs-fps* [*simp*]: *Abs-fps f* $ *n* = *f n*
  by (*simp add*: *Abs-fps-inverse*)

Definition of the basic elements 0 and 1 and the basic operations of addition, negation and multiplication

instantiation *fps* :: (*zero*) *zero*
begin

definition *fps-zero-def*:
  0 = *Abs-fps* (λ*n*. *0*)

instance ..
end

lemma *fps-zero-nth* [*simp*]: *0* $ *n* = *0*
  unfolding *fps-zero-def* **by** *simp*

instantiation *fps* :: ({*one*, *zero*}) *one*
begin

definition *fps-one-def*:
  1 = *Abs-fps* (λ*n*. *if n* = *0 then 1 else 0*)

instance ..
end

lemma *fps-one-nth* [*simp*]: *1* $ *n* = (*if n* = *0 then 1 else 0*)
  unfolding *fps-one-def* **by** *simp*

instantiation *fps* :: (*plus*) *plus*
begin

definition *fps-plus-def*:
  op + = (λ*f g*. *Abs-fps* (λ*n*. *f* $ *n* + *g* $ *n*))

instance ..

end

lemma *fps-add-nth* [*simp*]: $(f + g)\ \$\ n = f\ \$\ n + g\ \$\ n$
unfolding *fps-plus-def* **by** *simp*

instantiation *fps* :: (*minus*) *minus*
begin

definition *fps-minus-def*:
$op - = (\lambda f\ g.\ Abs\text{-}fps\ (\lambda n.\ f\ \$\ n - g\ \$\ n))$

instance ..
end

lemma *fps-sub-nth* [*simp*]: $(f - g)\ \$\ n = f\ \$\ n - g\ \$\ n$
unfolding *fps-minus-def* **by** *simp*

instantiation *fps* :: (*uminus*) *uminus*
begin

definition *fps-uminus-def*:
$uminus = (\lambda f.\ Abs\text{-}fps\ (\lambda n.\ -\ (f\ \$\ n)))$

instance ..
end

lemma *fps-neg-nth* [*simp*]: $(-\ f)\ \$\ n = -\ (f\ \$\ n)$
unfolding *fps-uminus-def* **by** *simp*

instantiation *fps* :: ({*comm-monoid-add*, *times*}) *times*
begin

definition *fps-times-def*:
$op * = (\lambda f\ g.\ Abs\text{-}fps\ (\lambda n.\ \sum i{=}0..n.\ f\ \$\ i * g\ \$\ (n - i)))$

instance ..
end

lemma *fps-mult-nth*: $(f * g)\ \$\ n = (\sum i{=}0..n.\ f\$i * g\$(n - i))$
unfolding *fps-times-def* **by** *simp*

declare *atLeastAtMost-iff*[*presburger*]
declare *Bex-def*[*presburger*]
declare *Ball-def*[*presburger*]

lemma *mult-delta-left*:
fixes *x y* :: *'a*::*mult-zero*
shows (*if b then x else 0*) * *y* = (*if b then x * y else 0*)
by *simp*

lemma *mult-delta-right*:
fixes $x\ y :: 'a$::*mult-zero*
shows $x * (\mathit{if}\ b\ \mathit{then}\ y\ \mathit{else}\ 0) = (\mathit{if}\ b\ \mathit{then}\ x * y\ \mathit{else}\ 0)$
by *simp*

lemma *cond-value-iff*: $f\ (\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ b\ \mathit{then}\ f\ x\ \mathit{else}\ f\ y)$
by *auto*
lemma *cond-application-beta*: $(\mathit{if}\ b\ \mathit{then}\ f\ \mathit{else}\ g)\ x = (\mathit{if}\ b\ \mathit{then}\ f\ x\ \mathit{else}\ g\ x)$
by *auto*

39.2 Formal power series form a commutative ring with unity, if the range of sequences they represent is a commutative ring with unity

instance *fps* :: (*semigroup-add*) *semigroup-add*
proof
fix $a\ b\ c :: 'a\ fps$ **show** $a + b + c = a + (b + c)$
by (*simp add*: *fps-ext add-assoc*)
qed

instance *fps* :: (*ab-semigroup-add*) *ab-semigroup-add*
proof
fix $a\ b :: 'a\ fps$ **show** $a + b = b + a$
by (*simp add*: *fps-ext add-commute*)
qed

lemma *fps-mult-assoc-lemma*:
fixes $k :: nat$ **and** $f :: nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a$::*comm-monoid-add*
shows $(\sum j{=}0..k.\ \sum i{=}0..j.\ f\ i\ (j - i)\ (n - j)) =$
$(\sum j{=}0..k.\ \sum i{=}0..k - j.\ f\ j\ i\ (n - j - i))$
proof (*induct k*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc k*) **thus** *?case*
by (*simp add*: *Suc-diff-le setsum-addf add-assoc*
cong: *strong-setsum-cong*)
qed

instance *fps* :: (*semiring-0*) *semigroup-mult*
proof
fix $a\ b\ c :: 'a\ fps$
show $(a * b) * c = a * (b * c)$
proof (*rule fps-ext*)
fix $n :: nat$
have $(\sum j{=}0..n.\ \sum i{=}0..j.\ a\$i * b\$(j - i) * c\$(n - j)) =$
$(\sum j{=}0..n.\ \sum i{=}0..n - j.\ a\$j * b\$i * c\$(n - j - i))$
by (*rule fps-mult-assoc-lemma*)
thus $((a * b) * c)\ \$\ n = (a * (b * c))\ \$\ n$

```
    by (simp add: fps-mult-nth setsum-right-distrib
                  setsum-left-distrib mult-assoc)
  qed
qed

lemma fps-mult-commute-lemma:
  fixes n :: nat and f :: nat ⇒ nat ⇒ 'a::comm-monoid-add
  shows (∑ i=0..n. f i (n − i)) = (∑ i=0..n. f (n − i) i)
proof (rule setsum-reindex-cong)
  show inj-on (λi. n − i) {0..n}
    by (rule inj-onI) simp
  show {0..n} = (λi. n − i) ` {0..n}
    by (auto, rule-tac x=n − x in image-eqI, simp-all)
next
  fix i assume i ∈ {0..n}
  hence n − (n − i) = i by simp
  thus f (n − i) i = f (n − i) (n − (n − i)) by simp
qed

instance fps :: (comm-semiring-0) ab-semigroup-mult
proof
  fix a b :: 'a fps
  show a * b = b * a
  proof (rule fps-ext)
    fix n :: nat
    have (∑ i=0..n. a$i * b$(n − i)) = (∑ i=0..n. a$(n − i) * b$i)
      by (rule fps-mult-commute-lemma)
    thus (a * b) $ n = (b * a) $ n
      by (simp add: fps-mult-nth mult-commute)
  qed
qed

instance fps :: (monoid-add) monoid-add
proof
  fix a :: 'a fps show 0 + a = a
    by (simp add: fps-ext)
next
  fix a :: 'a fps show a + 0 = a
    by (simp add: fps-ext)
qed

instance fps :: (comm-monoid-add) comm-monoid-add
proof
  fix a :: 'a fps show 0 + a = a
    by (simp add: fps-ext)
qed

instance fps :: (semiring-1) monoid-mult
proof
```

fix *a* :: *'a fps* **show** *1 * a = a*
by (*simp add*: *fps-ext fps-mult-nth mult-delta-left setsum-delta*)
next
fix *a* :: *'a fps* **show** *a * 1 = a*
by (*simp add*: *fps-ext fps-mult-nth mult-delta-right setsum-delta'*)
qed

instance *fps* :: (*cancel-semigroup-add*) *cancel-semigroup-add*
proof
fix *a b c* :: *'a fps*
assume *a + b = a + c* **then show** *b = c*
by (*simp add*: *expand-fps-eq*)
next
fix *a b c* :: *'a fps*
assume *b + a = c + a* **then show** *b = c*
by (*simp add*: *expand-fps-eq*)
qed

instance *fps* :: (*cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
proof
fix *a b c* :: *'a fps*
assume *a + b = a + c* **then show** *b = c*
by (*simp add*: *expand-fps-eq*)
qed

instance *fps* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add* **..**

instance *fps* :: (*group-add*) *group-add*
proof
fix *a* :: *'a fps* **show** *- a + a = 0*
by (*simp add*: *fps-ext*)
next
fix *a b* :: *'a fps* **show** *a - b = a + - b*
by (*simp add*: *fps-ext diff-minus*)
qed

instance *fps* :: (*ab-group-add*) *ab-group-add*
proof
fix *a* :: *'a fps*
show *- a + a = 0*
by (*simp add*: *fps-ext*)
next
fix *a b* :: *'a fps*
show *a - b = a + - b*
by (*simp add*: *fps-ext*)
qed

instance *fps* :: (*zero-neq-one*) *zero-neq-one*
by *default* (*simp add*: *expand-fps-eq*)

instance *fps* :: (*semiring-0*) *semiring*
proof
fix *a b c* :: *'a fps*
show $(a + b) * c = a * c + b * c$
by (*simp add*: *expand-fps-eq fps-mult-nth left-distrib setsum-addf*)
next
fix *a b c* :: *'a fps*
show $a * (b + c) = a * b + a * c$
by (*simp add*: *expand-fps-eq fps-mult-nth right-distrib setsum-addf*)
qed

instance *fps* :: (*semiring-0*) *semiring-0*
proof
fix *a*:: *'a fps* **show** $0 * a = 0$
by (*simp add*: *fps-ext fps-mult-nth*)
next
fix *a*:: *'a fps* **show** $a * 0 = 0$
by (*simp add*: *fps-ext fps-mult-nth*)
qed

instance *fps* :: (*semiring-0-cancel*) *semiring-0-cancel* **..**

39.3 Selection of the nth power of the implicit variable in the infinite sum

lemma *fps-nonzero-nth*: $f \neq 0 \longleftrightarrow (\exists\ n.\ f\ \$n \neq 0)$
by (*simp add*: *expand-fps-eq*)

lemma *fps-nonzero-nth-minimal*:
$f \neq 0 \longleftrightarrow (\exists n.\ f\ \$\ n \neq 0 \wedge (\forall m<n.\ f\ \$\ m = 0))$
proof
let *?n* = *LEAST n*. $f\ \$\ n \neq 0$
assume $f \neq 0$
then have $\exists n.\ f\ \$\ n \neq 0$
by (*simp add*: *fps-nonzero-nth*)
then have $f\ \$\ ?n \neq 0$
by (*rule LeastI-ex*)
moreover have $\forall m<?n.\ f\ \$\ m = 0$
by (*auto dest*: *not-less-Least*)
ultimately have $f\ \$\ ?n \neq 0 \wedge (\forall m<?n.\ f\ \$\ m = 0)$ **..**
then show $\exists n.\ f\ \$\ n \neq 0 \wedge (\forall m<n.\ f\ \$\ m = 0)$ **..**
next
assume $\exists n.\ f\ \$\ n \neq 0 \wedge (\forall m<n.\ f\ \$\ m = 0)$
then show $f \neq 0$ **by** (*auto simp add*: *expand-fps-eq*)
qed

lemma *fps-eq-iff*: $f = g \longleftrightarrow (\forall n.\ f\ \$\ n = g\ \$n)$
by (*rule expand-fps-eq*)

lemma *fps-setsum-nth*: (*setsum f S*) $ *n* = *setsum* (λ*k*. (*f k*) $ *n*) *S*
proof (*cases finite S*)
assume ¬ *finite S* **then show** *?thesis* **by** *simp*
next
assume *finite S*
then show *?thesis* **by** (*induct set*: *finite*) *auto*
qed

39.4 Injection of the basic ring elements and multiplication by scalars

definition
fps-const c = *Abs-fps* (λ*n*. *if n* = *0 then c else 0*)

lemma *fps-nth-fps-const* [*simp*]: *fps-const c* $ *n* = (*if n* = *0 then c else 0*)
unfolding *fps-const-def* **by** *simp*

lemma *fps-const-0-eq-0* [*simp*]: *fps-const 0* = *0*
by (*simp add*: *fps-ext*)

lemma *fps-const-1-eq-1* [*simp*]: *fps-const 1* = *1*
by (*simp add*: *fps-ext*)

lemma *fps-const-neg* [*simp*]: − (*fps-const* (*c*::*'a*::*ring*)) = *fps-const* (− *c*)
by (*simp add*: *fps-ext*)

lemma *fps-const-add* [*simp*]: *fps-const* (*c*::*'a*::*monoid-add*) + *fps-const d* = *fps-const* (*c* + *d*)
by (*simp add*: *fps-ext*)
lemma *fps-const-sub* [*simp*]: *fps-const* (*c*::*'a*::*group-add*) − *fps-const d* = *fps-const* (*c* − *d*)
by (*simp add*: *fps-ext*)
lemma *fps-const-mult*[*simp*]: *fps-const* (*c*::*'a*::*ring*) ∗ *fps-const d* = *fps-const* (*c* ∗ *d*)
by (*simp add*: *fps-eq-iff fps-mult-nth setsum-0'*)

lemma *fps-const-add-left*: *fps-const* (*c*::*'a*::*monoid-add*) + *f* = *Abs-fps* (λ*n*. *if n* = *0 then c* + *f*$*0 else f*$*n*)
by (*simp add*: *fps-ext*)

lemma *fps-const-add-right*: *f* + *fps-const* (*c*::*'a*::*monoid-add*) = *Abs-fps* (λ*n*. *if n* = *0 then f*$*0* + *c else f*$*n*)
by (*simp add*: *fps-ext*)

lemma *fps-const-mult-left*: *fps-const* (*c*::*'a*::*semiring-0*) ∗ *f* = *Abs-fps* (λ*n*. *c* ∗ *f*$*n*)
unfolding *fps-eq-iff fps-mult-nth*
by (*simp add*: *fps-const-def mult-delta-left setsum-delta*)

lemma *fps-const-mult-right*: $f * fps\text{-}const\ (c::'a::semiring\text{-}0) = Abs\text{-}fps\ (\lambda n.\ f\$n * c)$
unfolding *fps-eq-iff fps-mult-nth*
by (*simp add*: *fps-const-def mult-delta-right setsum-delta'*)

lemma *fps-mult-left-const-nth* [*simp*]: $(fps\text{-}const\ (c::'a::semiring\text{-}1) * f)\$n = c * f\$n$
by (*simp add*: *fps-mult-nth mult-delta-left setsum-delta*)

lemma *fps-mult-right-const-nth* [*simp*]: $(f * fps\text{-}const\ (c::'a::semiring\text{-}1))\$n = f\$n * c$
by (*simp add*: *fps-mult-nth mult-delta-right setsum-delta'*)

39.5 Formal power series form an integral domain

instance *fps* :: (*ring*) *ring* **..**

instance *fps* :: (*ring-1*) *ring-1*
by (*intro-classes*, *auto simp add*: *diff-minus left-distrib*)

instance *fps* :: (*comm-ring-1*) *comm-ring-1*
by (*intro-classes*, *auto simp add*: *diff-minus left-distrib*)

instance *fps* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors*
proof
fix *a b* :: *'a fps*
assume *a0*: $a \neq 0$ **and** *b0*: $b \neq 0$
then obtain *i j* **where** *i*: $a\$i \neq 0\ \forall k<i.\ a\$k=0$
and *j*: $b\$j \neq 0\ \forall k<j.\ b\$k = 0$ **unfolding** *fps-nonzero-nth-minimal*
by *blast+*
have $(a * b)\ \$\ (i+j) = (\sum k=0..i+j.\ a\$k * b\$(i+j-k))$
by (*rule fps-mult-nth*)
also have $\ldots = (a\$i * b\$(i+j-i)) + (\sum k \in \{0..i+j\}-\{i\}.\ a\$k * b\$(i+j-k))$
by (*rule setsum-diff1'*) *simp-all*
also have $(\sum k \in \{0..i+j\}-\{i\}.\ a\$k * b\$(i+j-k)) = 0$
proof (*rule setsum-0' [rule-format]*)
fix *k* **assume** $k \in \{0..i+j\} - \{i\}$
then have $k < i \vee i+j-k < j$ **by** *auto*
then show $a\$k * b\$(i+j-k) = 0$ **using** *i j* **by** *auto*
qed
also have $a\$i * b\$(i+j-i) + 0 = a\$i * b\$j$ **by** *simp*
also have $a\$i * b\$j \neq 0$ **using** *i j* **by** *simp*
finally have $(a*b)\ \$\ (i+j) \neq 0$ **.**
then show $a*b \neq 0$ **unfolding** *fps-nonzero-nth* **by** *blast*
qed

instance *fps* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors* **..**

instance *fps* :: (*idom*) *idom* **..**

instantiation *fps* :: (*comm-ring-1*) *number-ring*
begin
definition *number-of-fps-def*: (*number-of k*::*'a fps*) = *of-int k*

instance proof
qed (*rule number-of-fps-def*)
end

lemma *number-of-fps-const*: (*number-of k*::(*'a*::*comm-ring-1*) *fps*) = *fps-const* (*of-int k*)

proof(*induct k rule*: *int-induct* [**where** *k=0*])
case *base* **thus** *?case* **unfolding** *number-of-fps-def of-int-0* **by** *simp*
next
case (*step1 i*) **thus** *?case* **unfolding** *number-of-fps-def*
by (*simp add*: *fps-const-add*[*symmetric*] *del*: *fps-const-add*)
next
case (*step2 i*) **thus** *?case* **unfolding** *number-of-fps-def*
by (*simp add*: *fps-const-sub*[*symmetric*] *del*: *fps-const-sub*)
qed

39.6 The eXtractor series X

lemma *minus-one-power-iff*: (− (*1*::*'a* :: {*comm-ring-1*})) ^ *n* = (*if even n then 1 else* − *1*)
by (*induct n*, *auto*)

definition *X* = *Abs-fps* (*λn. if n* = *1 then 1 else 0*)
lemma *X-mult-nth*[*simp*]: (*X* ∗ (*f* :: (*'a*::*semiring-1*) *fps*)) $*n* = (*if n* = *0 then 0 else f* $ (*n* − *1*))
proof−
{assume *n*: *n* ≠ *0*
have *fN*: *finite* {*0* .. *n*} **by** *simp*
have (*X* ∗ *f*) $*n* = (∑ *i* = *0*..*n*. *X* $ *i* ∗ *f* $ (*n* − *i*)) **by** (*simp add*: *fps-mult-nth*)
also have ... = *f* $ (*n* − *1*)
using *n* **by** (*simp add*: *X-def mult-delta-left setsum-delta* [*OF fN*])
finally have *?thesis* **using** *n* **by** *simp* **}**
moreover
{assume *n*: *n=0* **hence** *?thesis* **by** (*simp add*: *fps-mult-nth X-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *X-mult-right-nth*[*simp*]: ((*f* :: (*'a*::*comm-semiring-1*) *fps*) ∗ *X*) $*n* = (*if n* = *0 then 0 else f* $ (*n* − *1*))
by (*metis X-mult-nth mult-commute*)

lemma *X-power-iff*: *X*^*k* = *Abs-fps* (*λn. if n* = *k then* (*1*::*'a*::*comm-ring-1*) *else*

0)
proof(*induct k*)
case *0* **thus** *?case* **by** (*simp add*: *X-def fps-eq-iff*)
next
case (*Suc k*)
{fix *m*
have (*X^Suc k*) $ *m* = (*if m* = *0 then* (*0*::*'a*) *else* (*X^k*) $ (*m* − *1*))
by (*simp add*: *power-Suc del*: *One-nat-def*)
then have (*X^Suc k*) $ *m* = (*if m* = *Suc k then* (*1*::*'a*) *else 0*)
using *Suc.hyps* **by** (*auto cong del*: *if-weak-cong*)**}**
then show *?case* **by** (*simp add*: *fps-eq-iff*)
qed

lemma *X-power-mult-nth*: (*X^k* ∗ (*f* :: (*'a*::*comm-ring-1*) *fps*)) $*n* = (*if n* < *k then 0 else f* $ (*n* − *k*))
apply (*induct k arbitrary*: *n*)
apply (*simp*)
unfolding *power-Suc mult-assoc*
by (*case-tac n*, *auto*)

lemma *X-power-mult-right-nth*: ((*f* :: (*'a*::*comm-ring-1*) *fps*) ∗ *X^k*) $*n* = (*if n* < *k then 0 else f* $ (*n* − *k*))
by (*metis X-power-mult-nth mult-commute*)

39.7 Formal Power series form a metric space

definition (**in** *dist*) *ball-def*: *ball x r* = {*y*. *dist y x* < *r*}
instantiation *fps* :: (*comm-ring-1*) *dist*
begin

definition *dist-fps-def*: *dist* (*a*::*'a fps*) *b* = (*if* (∃ *n*. *a*$*n* ≠ *b*$*n*) *then inverse* (*2* ^ *The* (*leastP* (λ*n*. *a*$*n* ≠ *b*$*n*))) *else 0*)

lemma *dist-fps-ge0*: *dist* (*a*::*'a fps*) *b* ≥ *0*
by (*simp add*: *dist-fps-def*)

lemma *dist-fps-sym*: *dist* (*a*::*'a fps*) *b* = *dist b a*
apply (*auto simp add*: *dist-fps-def*)
apply (*rule cong*[*OF refl*, **where** *x*=(λ*n*::*nat*. *a* $ *n* ≠ *b* $ *n*)])
apply (*rule ext*)
by *auto*
instance ..
end

lemma *fps-nonzero-least-unique*: **assumes** *a0*: *a* ≠ *0*
shows ∃! *n*. *leastP* (λ*n*. *a*$*n* ≠ *0*) *n*
proof−
from *fps-nonzero-nth-minimal*[*of a*] *a0*
obtain *n* **where** *n*: *a*$*n* ≠ *0* ∀ *m* < *n*. *a*$*m* = *0* **by** *blast*

```
  from n have ln: leastP (λn. a$n ≠ 0) n
    by (auto simp add: leastP-def setge-def not-le[symmetric])
  moreover
  {fix m assume leastP (λn. a$n ≠ 0) m
    then have m = n using ln
      apply (auto simp add: leastP-def setge-def)
      apply (erule allE[where x=n])
      apply (erule allE[where x=m])
      by simp}
  ultimately show ?thesis by blast
qed

lemma fps-eq-least-unique: assumes ab: (a::('a::ab-group-add) fps) ≠ b
  shows ∃! n. leastP (λn. a$n ≠ b$n) n
using fps-nonzero-least-unique[of a − b] ab
by auto

instantiation fps :: (comm-ring-1) metric-space
begin

definition open-fps-def: open (S :: 'a fps set) = (∀ a ∈ S. ∃ r. r >0 ∧ ball a r ⊆
S)

instance
proof
  fix S :: 'a fps set
  show open S = (∀ x∈S. ∃ e>0. ∀ y. dist y x < e ⟶ y ∈ S)
    by (auto simp add: open-fps-def ball-def subset-eq)
next
{  fix a b :: 'a fps
  {assume ab: a = b
   then have ¬ (∃ n. a$n ≠ b$n) by simp
   then have dist a b = 0 by (simp add: dist-fps-def)}
 moreover
 {assume d: dist a b = 0
   then have ∀ n. a$n = b$n
     by − (rule ccontr, simp add: dist-fps-def)
   then have a = b by (simp add: fps-eq-iff)}
 ultimately show dist a b =0 ⟷ a = b by blast}
note th = this
from th have th'[simp]: ⋀a::'a fps. dist a a = 0 by simp
  fix a b c :: 'a fps
  {assume ab: a = b then have d0: dist a b = 0  unfolding th .
    then have dist a b ≤ dist a c + dist b c
      using dist-fps-ge0[of a c] dist-fps-ge0[of b c] by simp}
  moreover
  {assume c: c = a ∨ c = b then have dist a b ≤ dist a c + dist b c
     by (cases c=a, simp-all add: th dist-fps-sym) }
  moreover
```

```
 {assume ab: a ≠ b and ac: a ≠ c and bc: b ≠ c
  let ?P = λa b n. a$n ≠ b$n
  from fps-eq-least-unique[OF ab] fps-eq-least-unique[OF ac]
    fps-eq-least-unique[OF bc]
  obtain nab nac nbc where nab: leastP (?P a b) nab
    and nac: leastP (?P a c) nac
    and nbc: leastP (?P b c) nbc by blast
  from nab have nab': ⋀m. m < nab ⟹ a$m = b$m a$nab ≠ b$nab
    by (auto simp add: leastP-def setge-def)
  from nac have nac': ⋀m. m < nac ⟹ a$m = c$m a$nac ≠ c$nac
    by (auto simp add: leastP-def setge-def)
  from nbc have nbc': ⋀m. m < nbc ⟹ b$m = c$m b$nbc ≠ c$nbc
    by (auto simp add: leastP-def setge-def)

  have th0: ⋀(a::'a fps) b. a ≠ b ⟷ (∃ n. a$n ≠ b$n)
    by (simp add: fps-eq-iff)
  from ab ac bc nab nac nbc
  have dab: dist a b = inverse (2 ^ nab)
    and dac: dist a c = inverse (2 ^ nac)
    and dbc: dist b c = inverse (2 ^ nbc)
    unfolding th0
    apply (simp-all add: dist-fps-def)
    apply (erule the1-equality[OF fps-eq-least-unique[OF ab]])
    apply (erule the1-equality[OF fps-eq-least-unique[OF ac]])
    by (erule the1-equality[OF fps-eq-least-unique[OF bc]])
  from ab ac bc have nz: dist a b ≠ 0 dist a c ≠ 0 dist b c ≠ 0
    unfolding th by simp-all
  from nz have pos: dist a b > 0 dist a c > 0 dist b c > 0
    using dist-fps-ge0[of a b] dist-fps-ge0[of a c] dist-fps-ge0[of b c]
    by auto
  have th1: ⋀n. (2::real) ^n >0 by auto
  {assume h: dist a b > dist a c + dist b c
    then have gt: dist a b > dist a c dist a b > dist b c
      using pos by auto
    from gt have gtn: nab < nbc nab < nac
      unfolding dab dbc dac by (auto simp add: th1)
    from nac'(1)[OF gtn(2)] nbc'(1)[OF gtn(1)]
    have a$nab = b$nab by simp
    with nab'(2) have False  by simp}
  then have dist a b ≤ dist a c + dist b c
    by (auto simp add: not-le[symmetric]) }
 ultimately show dist a b ≤ dist a c + dist b c by blast
qed

end
```

The infinite sums and justification of the notation in textbooks

```
lemma reals-power-lt-ex: assumes xp: x > 0 and y1: (y::real) > 1
 shows ∃ k>0. (1/y) ^k < x
```

proof−
have *yp*: $y > 0$ **using** *y1* **by** *simp*
from *reals-Archimedean2*[*of max 0* $(-\ log\ y\ x) + 1$]
obtain *k*::*nat* **where** *k*: *real* $k > max\ 0\ (-\ log\ y\ x) + 1$ **by** *blast*
from *k* **have** *kp*: $k > 0$ **by** *simp*
from *k* **have** *real* $k > -\ log\ y\ x$ **by** *simp*
then have *ln* $y * real\ k > -\ ln\ x$ **unfolding** *log-def*
using *ln-gt-zero-iff*[*OF yp*] *y1*
by (*simp add*: *minus-divide-left field-simps del*:*minus-divide-left*[*symmetric*])
then have *ln* $y * real\ k + ln\ x > 0$ **by** *simp*
then have *exp* $(real\ k * ln\ y + ln\ x) > exp\ 0$
by (*simp add*: *mult-ac*)
then have $y\ \hat{}\ k * x > 1$
unfolding *exp-zero exp-add exp-real-of-nat-mult*
exp-ln[*OF xp*] *exp-ln*[*OF yp*] **by** *simp*
then have $x > (1/y)\ \hat{}k$ **using** *yp*
by (*simp add*: *field-simps nonzero-power-divide*)
then show *?thesis* **using** *kp* **by** *blast*
qed
lemma *X-nth*[*simp*]: $X\$n = (if\ n = 1\ then\ 1\ else\ 0)$ **by** (*simp add*: *X-def*)
lemma *X-power-nth*[*simp*]: $(X\ \hat{}k)\ \$n = (if\ n = k\ then\ 1\ else\ (0::'a::comm\text{-}ring\text{-}1))$
by (*simp add*: *X-power-iff*)

lemma *fps-sum-rep-nth*: $(setsum\ (\%i.\ fps\text{-}const(a\$i)*X\ \hat{}\ i)\ \{0..m\})\$n = (if\ n \leq m\ then\ a\$n\ else\ (0::'a::comm\text{-}ring\text{-}1))$
apply (*auto simp add*: *fps-eq-iff fps-setsum-nth X-power-nth cond-application-beta cond-value-iff cong del*: *if-weak-cong*)
by (*simp add*: *setsum-delta'*)

lemma *fps-notation*:
$(\%n.\ setsum\ (\%i.\ fps\text{-}const(a\$i) * X\ \hat{}\ i)\ \{0..n\})$ −−−−> a (**is** *?s* −−−−> a)
proof−
{fix *r*:: *real*
assume *rp*: $r > 0$
have *th0*: $(2::real) > 1$ **by** *simp*
from *reals-power-lt-ex*[*OF rp th0*]
obtain *n0* **where** *n0*: $(1/2)\ \hat{}n0 < r\ n0 > 0$ **by** *blast*
{fix *n*::*nat*
assume *nn0*: $n \geq n0$
then have *thnn0*: $(1/2)\ \hat{}n <= (1/2 :: real)\ \hat{}n0$
by (*auto intro*: *power-decreasing*)
{assume *?s n* = *a* **then have** *dist* (*?s n*) $a < r$
unfolding *dist-eq-0-iff*[*of ?s n a*, *symmetric*]
using *rp* **by** (*simp del*: *dist-eq-0-iff*)**}**
moreover
{assume *neq*: *?s n* ≠ *a*
from *fps-eq-least-unique*[*OF neq*]
obtain *k* **where** *k*: *leastP* $(\lambda i.\ ?s\ n\ \$\ i \neq a\$i)\ k$ **by** *blast*

have *th0*: $\bigwedge(a::'a\ fps)\ b.\ a \neq b \longleftrightarrow (\exists n.\ a\$n \neq b\$n)$
by (*simp add*: *fps-eq-iff*)
from *neq* **have** *dth*: *dist* (*?s n*) *a* = (*1*/*2*) ^*k*
unfolding *th0 dist-fps-def*
unfolding *the1-equality*[*OF fps-eq-least-unique*[*OF neq*], *OF k*]
by (*auto simp add*: *inverse-eq-divide power-divide*)

from *k* **have** *kn*: *k* > *n*
by (*simp add*: *leastP-def setge-def fps-sum-rep-nth split*:*split-if-asm*)
then have *dist* (*?s n*) *a* < (*1*/*2*) ^*n* **unfolding** *dth*
by (*auto intro*: *power-strict-decreasing*)
also have ... <= (*1*/*2*) ^*n0* **using** *nn0*
by (*auto intro*: *power-decreasing*)
also have ... < *r* **using** *n0* **by** *simp*
finally have *dist* (*?s n*) *a* < *r* **.}**
ultimately have *dist* (*?s n*) *a* < *r* **by** *blast***}**
then have $\exists n0.\ \forall\ n \geq n0.$ *dist* (*?s n*) *a* < *r* **by** *blast***}**
then show *?thesis* **unfolding** *LIMSEQ-def* **by** *blast*
qed

39.8 Inverses of formal power series

declare *setsum-cong*[*fundef-cong*]

instantiation *fps* :: ({*comm-monoid-add*, *inverse*, *times*, *uminus*}) *inverse*
begin

fun *natfun-inverse*:: *'a fps* ⇒ *nat* ⇒ *'a* **where**
natfun-inverse f 0 = *inverse* (*f*$*0*)
| *natfun-inverse f n* = − *inverse* (*f*$*0*) ∗ *setsum* (λ*i*. *f*$*i* ∗ *natfun-inverse f* (*n* − *i*)) {*1*..*n*}

definition *fps-inverse-def*:
inverse f = (*if f* $ *0* = *0 then 0 else Abs-fps* (*natfun-inverse f*))

definition *fps-divide-def*: *divide* = (λ(*f*::*'a fps*) *g*. *f* ∗ *inverse g*)

instance ..

end

lemma *fps-inverse-zero*[*simp*]:
inverse (*0* :: *'a*::{*comm-monoid-add*,*inverse*, *times*, *uminus*} *fps*) = *0*
by (*simp add*: *fps-ext fps-inverse-def*)

lemma *fps-inverse-one*[*simp*]: *inverse* (*1* :: *'a*::{*division-ring*,*zero-neq-one*} *fps*) = *1*
apply (*auto simp add*: *expand-fps-eq fps-inverse-def*)
by (*case-tac n*, *auto*)

lemma *inverse-mult-eq-1* [*intro*]: **assumes** *f0*: $f\$0 \neq (0::'a::field)$
shows *inverse f * f = 1*
proof−
have *c*: *inverse f * f = f * inverse f* **by** (*simp add*: *mult-commute*)
from *f0* **have** *ifn*: $\bigwedge n.$ *inverse f $ n = natfun-inverse f n*
by (*simp add*: *fps-inverse-def*)
from *f0* **have** *th0*: *(inverse f * f) $ 0 = 1*
by (*simp add*: *fps-mult-nth fps-inverse-def*)
{fix *n*::*nat* **assume** *np*: *n >0*
from *np* **have** *eq*: *{0..n} = {0} ∪ {1 .. n}* **by** *auto*
have *d*: *{0} ∩ {1 .. n} = {}* **by** *auto*
have *f*: *finite {0::nat} finite {1..n}* **by** *auto*
from *f0 np* **have** *th0*: *− (inverse f$n) =*
(setsum (λi. f$i * natfun-inverse f (n − i)) {1..n}) / (f$0)
by (*cases n*, *simp*, *simp add*: *divide-inverse fps-inverse-def*)
from *th0*[*symmetric*, *unfolded nonzero-divide-eq-eq*[*OF f0*]]
have *th1*: *setsum (λi. f$i * natfun-inverse f (n − i)) {1..n} =*
− (f$0) * (inverse f)$n
by (*simp add*: *field-simps*)
have *(f * inverse f) $ n = (∑ i = 0..n. f $i * natfun-inverse f (n − i))*
unfolding *fps-mult-nth ifn* **..**
also have *... = f$0 * natfun-inverse f n*
*+ (∑ i = 1..n. f$i * natfun-inverse f (n−i))*
unfolding *setsum-Un-disjoint*[*OF f d*, *unfolded eq*[*symmetric*]]
by *simp*
also have *... = 0* **unfolding** *th1 ifn* **by** *simp*
finally have *(inverse f * f)$n = 0* **unfolding** *c* **. }**
with *th0* **show** *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

lemma *fps-inverse-0-iff*[*simp*]: *(inverse f)$0 = (0::'a::division-ring) ⟷ f$0 = 0*
by (*simp add*: *fps-inverse-def nonzero-imp-inverse-nonzero*)

lemma *fps-inverse-eq-0-iff*[*simp*]: *inverse f = (0:: ('a::field) fps) ⟷ f $0 = 0*
proof−
{assume *f$0 = 0* **hence** *inverse f = 0* **by** (*simp add*: *fps-inverse-def*)**}**
moreover
{assume *h*: *inverse f = 0* **and** *c*: *f $0 ≠ 0*
from *inverse-mult-eq-1*[*OF c*] *h* **have** *False* **by** *simp***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *fps-inverse-idempotent*[*intro*]: **assumes** *f0*: $f\$0 \neq (0::'a::field)$
shows *inverse (inverse f) = f*
proof−
from *f0* **have** *if0*: *inverse f $ 0 ≠ 0* **by** *simp*
from *inverse-mult-eq-1*[*OF f0*] *inverse-mult-eq-1*[*OF if0*]

have *th0*: *inverse f * f = inverse f * inverse (inverse f)* **by** (*simp add*: *mult-ac*)
then show *?thesis* **using** *f0* **unfolding** *mult-cancel-left* **by** *simp*
qed

lemma *fps-inverse-unique*: **assumes** *f0*: *f$0 ≠ (0::'a::field)* **and** *fg*: *f*g = 1*
shows *inverse f = g*
proof−
from *inverse-mult-eq-1*[*OF f0*] *fg*
have *th0*: *inverse f * f = g * f* **by** (*simp add*: *mult-ac*)
then show *?thesis* **using** *f0* **unfolding** *mult-cancel-right*
by (*auto simp add*: *expand-fps-eq*)
qed

lemma *fps-inverse-gp*: *inverse (Abs-fps(λn. (1::'a::field)))*
= Abs-fps (λn. if n= 0 then 1 else if n=1 then − 1 else 0)
apply (*rule fps-inverse-unique*)
apply *simp*
apply (*simp add*: *fps-eq-iff fps-mult-nth*)
proof(*clarsimp*)
fix *n*::*nat* **assume** *n*: *n > 0*
let *?f = λi. if n = i then (1::'a) else if n − i = 1 then − 1 else 0*
let *?g = λi. if i = n then 1 else if i=n − 1 then − 1 else 0*
let *?h = λi. if i=n − 1 then − 1 else 0*
have *th1*: *setsum ?f {0..n} = setsum ?g {0..n}*
by (*rule setsum-cong2*) *auto*
have *th2*: *setsum ?g {0..n − 1} = setsum ?h {0..n − 1}*
using *n* **apply** − **by** (*rule setsum-cong2*) *auto*
have *eq*: *{0 .. n} = {0.. n − 1} ∪ {n}* **by** *auto*
from *n* **have** *d*: *{0.. n − 1} ∩ {n} = {}* **by** *auto*
have *f*: *finite {0.. n − 1} finite {n}* **by** *auto*
show *setsum ?f {0..n} = 0*
unfolding *th1*
apply (*simp add*: *setsum-Un-disjoint*[*OF f d*, *unfolded eq*[*symmetric*]] *del*: *One-nat-def*)
unfolding *th2*
by(*simp add*: *setsum-delta*)
qed

39.9 Formal Derivatives, and the MacLaurin theorem around 0

definition *fps-deriv f = Abs-fps (λn. of-nat (n + 1) * f $ (n + 1))*

lemma *fps-deriv-nth*[*simp*]: *fps-deriv f $ n = of-nat (n +1) * f $ (n+1)* **by** (*simp add*: *fps-deriv-def*)

lemma *fps-deriv-linear*[*simp*]: *fps-deriv (fps-const (a::'a::comm-semiring-1) * f + fps-const b * g) = fps-const a * fps-deriv f + fps-const b * fps-deriv g*
unfolding *fps-eq-iff fps-add-nth fps-const-mult-left fps-deriv-nth* **by** (*simp add*:

field-simps)

lemma *fps-deriv-mult*[*simp*]:
fixes $f :: ('a :: comm\text{-}ring\text{-}1)\ fps$
shows $fps\text{-}deriv\ (f * g) = f * fps\text{-}deriv\ g + fps\text{-}deriv\ f * g$
proof−
let *?D* = *fps-deriv*
{fix *n*::*nat*
let *?Zn* = {*0* ..*n*}
let *?Zn1* = {*0* .. *n* + *1*}
let *?f* = λ*i*. *i* + *1*
have *fi*: *inj-on ?f* {*0*..*n*} **by** (*simp add*: *inj-on-def*)
have *eq*: {*1*.. *n*+*1*} = *?f* ' {*0*..*n*} **by** *auto*
let *?g* = λ*i*. *of-nat* (*i*+*1*) ∗ *g* $ (*i*+*1*) ∗ *f* $ (*n* − *i*) +
of-nat (*i*+*1*)∗ *f* $ (*i*+*1*) ∗ *g* $ (*n* − *i*)
let *?h* = λ*i*. *of-nat i* ∗ *g* $ *i* ∗ *f* $ ((*n*+*1*) − *i*) +
of-nat i∗ *f* $ *i* ∗ *g* $ ((*n* + *1*) − *i*)
{fix *k* **assume** *k*: *k* ∈ {*0*..*n*}
have *?h* (*k* + *1*) = *?g k* **using** *k* **by** *auto***}**
note *th0* = *this*
have *eq′*: {*0*..*n* +*1*}− {*1* .. *n*+*1*} = {*0*} **by** *auto*
have *s0*: *setsum* (λ*i*. *of-nat i* ∗ *f* $ *i* ∗ *g* $ (*n* + *1* − *i*)) *?Zn1* = *setsum* (λ*i*. *of-nat* (*n* + *1* − *i*) ∗ *f* $ (*n* + *1* − *i*) ∗ *g* $ *i*) *?Zn1*
apply (*rule setsum-reindex-cong*[**where** *f*=λ*i*. *n* + *1* − *i*])
apply (*simp add*: *inj-on-def Ball-def*)
apply *presburger*
apply (*rule set-ext*)
apply (*presburger add*: *image-iff*)
by *simp*
have *s1*: *setsum* (λ*i*. *f* $ *i* ∗ *g* $ (*n* + *1* − *i*)) *?Zn1* = *setsum* (λ*i*. *f* $ (*n* + *1* − *i*) ∗ *g* $ *i*) *?Zn1*
apply (*rule setsum-reindex-cong*[**where** *f*=λ*i*. *n* + *1* − *i*])
apply (*simp add*: *inj-on-def Ball-def*)
apply *presburger*
apply (*rule set-ext*)
apply (*presburger add*: *image-iff*)
by *simp*
have (*f* ∗ *?D g* + *?D f* ∗ *g*)$*n* = (*?D g* ∗ *f* + *?D f* ∗ *g*)$*n* **by** (*simp only*: *mult-commute*)
also have ... = (∑ *i* = *0*..*n*. *?g i*)
by (*simp add*: *fps-mult-nth setsum-addf*[*symmetric*])
also have ... = *setsum ?h* {*1*..*n*+*1*}
using *th0 setsum-reindex-cong*[*OF fi eq*, *of ?g ?h*] **by** *auto*
also have ... = *setsum ?h* {*0*..*n*+*1*}
apply (*rule setsum-mono-zero-left*)
apply *simp*
apply (*simp add*: *subset-eq*)
unfolding *eq′*
by *simp*

```
    also have ... = (fps-deriv (f * g)) $ n
      apply (simp only: fps-deriv-nth fps-mult-nth setsum-addf)
      unfolding s0 s1
      unfolding setsum-addf[symmetric] setsum-right-distrib
      apply (rule setsum-cong2)
      by (auto simp add: of-nat-diff field-simps)
    finally have (f * ?D g + ?D f * g) $ n = ?D (f*g) $ n .}
  then show ?thesis unfolding fps-eq-iff by auto
qed

lemma fps-deriv-X[simp]: fps-deriv X = 1
  by (simp add: fps-deriv-def X-def fps-eq-iff)

lemma fps-deriv-neg[simp]: fps-deriv (− (f:: ('a:: comm-ring-1) fps)) = − (fps-deriv
f)
  by (simp add: fps-eq-iff fps-deriv-def)
lemma fps-deriv-add[simp]: fps-deriv ((f:: ('a::comm-ring-1) fps) + g) = fps-deriv
f + fps-deriv g
  using fps-deriv-linear[of 1 f 1 g] by simp

lemma fps-deriv-sub[simp]: fps-deriv ((f:: ('a::comm-ring-1) fps) − g) = fps-deriv
f − fps-deriv g
  unfolding diff-minus by simp

lemma fps-deriv-const[simp]: fps-deriv (fps-const c) = 0
  by (simp add: fps-ext fps-deriv-def fps-const-def)

lemma fps-deriv-mult-const-left[simp]: fps-deriv (fps-const (c::'a::comm-ring-1) *
f) = fps-const c * fps-deriv f
  by simp

lemma fps-deriv-0[simp]: fps-deriv 0 = 0
  by (simp add: fps-deriv-def fps-eq-iff)

lemma fps-deriv-1[simp]: fps-deriv 1 = 0
  by (simp add: fps-deriv-def fps-eq-iff )

lemma fps-deriv-mult-const-right[simp]: fps-deriv (f * fps-const (c::'a::comm-ring-1))
= fps-deriv f * fps-const c
  by simp

lemma fps-deriv-setsum: fps-deriv (setsum f S) = setsum (λi. fps-deriv (f i ::
('a::comm-ring-1) fps)) S
proof−
  {assume ¬ finite S hence ?thesis by simp}
  moreover
  {assume fS: finite S
    have ?thesis  by (induct rule: finite-induct[OF fS], simp-all)}
  ultimately show ?thesis by blast
```

qed

lemma *fps-deriv-eq-0-iff*[*simp*]: *fps-deriv f = 0 ⟷ (f = fps-const (f$0 :: 'a::{idom,semiring-char-0}))*
proof−
{assume *f= fps-const (f$0)* **hence** *fps-deriv f = fps-deriv (fps-const (f$0))* **by** *simp*
hence *fps-deriv f = 0* **by** *simp* **}**
moreover
{assume *z*: *fps-deriv f = 0*
hence ∀ *n*. *(fps-deriv f)$n = 0* **by** *simp*
hence ∀ *n*. *f$(n+1) = 0* **by** *(simp del*: *of-nat-Suc of-nat-add One-nat-def)*
hence *f = fps-const (f$0)*
apply *(clarsimp simp add*: *fps-eq-iff fps-const-def)*
apply *(erule-tac x=n − 1* **in** *allE)*
by *simp***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *fps-deriv-eq-iff*:
fixes *f*:: *('a::{idom,semiring-char-0}) fps*
shows *fps-deriv f = fps-deriv g ⟷ (f = fps-const(f$0 − g$0) + g)*
proof−
have *fps-deriv f = fps-deriv g ⟷ fps-deriv (f − g) = 0* **by** *simp*
also have *... ⟷ f − g = fps-const ((f−g)$0)* **unfolding** *fps-deriv-eq-0-iff* **..**
finally show *?thesis* **by** *(simp add*: *field-simps)*
qed

lemma *fps-deriv-eq-iff-ex*: *(fps-deriv f = fps-deriv g) ⟷ (∃ (c::'a::{idom,semiring-char-0}). f = fps-const c + g)*
apply *auto* **unfolding** *fps-deriv-eq-iff* **by** *blast*

fun *fps-nth-deriv* :: *nat* ⇒ *('a::semiring-1) fps* ⇒ *'a fps* **where**
fps-nth-deriv 0 f = f
| *fps-nth-deriv (Suc n) f = fps-nth-deriv n (fps-deriv f)*

lemma *fps-nth-deriv-commute*: *fps-nth-deriv (Suc n) f = fps-deriv (fps-nth-deriv n f)*
by *(induct n arbitrary*: *f*, *auto)*

lemma *fps-nth-deriv-linear*[*simp*]: *fps-nth-deriv n (fps-const (a::'a::comm-semiring-1) * f + fps-const b * g) = fps-const a * fps-nth-deriv n f + fps-const b * fps-nth-deriv n g*
by *(induct n arbitrary*: *f g*, *auto simp add*: *fps-nth-deriv-commute)*

lemma *fps-nth-deriv-neg*[*simp*]: *fps-nth-deriv n (− (f:: ('a:: comm-ring-1) fps)) = − (fps-nth-deriv n f)*
by *(induct n arbitrary*: *f*, *simp-all)*

lemma *fps-nth-deriv-add*[*simp*]: *fps-nth-deriv n* ((*f*:: ('*a*::*comm-ring-1*) *fps*) + *g*) = *fps-nth-deriv n f* + *fps-nth-deriv n g*
using *fps-nth-deriv-linear*[*of n 1 f 1 g*] **by** *simp*

lemma *fps-nth-deriv-sub*[*simp*]: *fps-nth-deriv n* ((*f*:: ('*a*::*comm-ring-1*) *fps*) − *g*) = *fps-nth-deriv n f* − *fps-nth-deriv n g*
unfolding *diff-minus fps-nth-deriv-add* **by** *simp*

lemma *fps-nth-deriv-0*[*simp*]: *fps-nth-deriv n 0* = *0*
by (*induct n*, *simp-all*)

lemma *fps-nth-deriv-1*[*simp*]: *fps-nth-deriv n 1* = (*if n* = *0 then 1 else 0*)
by (*induct n*, *simp-all*)

lemma *fps-nth-deriv-const*[*simp*]: *fps-nth-deriv n* (*fps-const c*) = (*if n* = *0 then fps-const c else 0*)
by (*cases n*, *simp-all*)

lemma *fps-nth-deriv-mult-const-left*[*simp*]: *fps-nth-deriv n* (*fps-const* (*c*::'*a*::*comm-ring-1*) ∗ *f*) = *fps-const c* ∗ *fps-nth-deriv n f*
using *fps-nth-deriv-linear*[*of n c f 0 0*] **by** *simp*

lemma *fps-nth-deriv-mult-const-right*[*simp*]: *fps-nth-deriv n* (*f* ∗ *fps-const* (*c*::'*a*::*comm-ring-1*)) = *fps-nth-deriv n f* ∗ *fps-const c*
using *fps-nth-deriv-linear*[*of n c f 0 0*] **by** (*simp add*: *mult-commute*)

lemma *fps-nth-deriv-setsum*: *fps-nth-deriv n* (*setsum f S*) = *setsum* (λ*i*. *fps-nth-deriv n* (*f i* :: ('*a*::*comm-ring-1*) *fps*)) *S*
proof−
{assume ¬ *finite S* **hence** *?thesis* **by** *simp***}**
moreover
{assume *fS*: *finite S*
have *?thesis* **by** (*induct rule*: *finite-induct*[*OF fS*], *simp-all*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *fps-deriv-maclauren-0*: (*fps-nth-deriv k* (*f*:: ('*a*::*comm-semiring-1*) *fps*)) $ *0* = *of-nat* (*fact k*) ∗ *f*$(*k*)
by (*induct k arbitrary*: *f*) (*auto simp add*: *field-simps of-nat-mult*)

39.10 Powers

lemma *fps-power-zeroth-eq-one*: *a*$*0* =*1* ⟹ *a*^*n* $ *0* = (*1*::'*a*::*semiring-1*)
by (*induct n*, *auto simp add*: *expand-fps-eq fps-mult-nth*)

lemma *fps-power-first-eq*: (*a*:: '*a*::*comm-ring-1 fps*)$*0* =*1* ⟹ *a*^*n* $ *1* = *of-nat n* ∗ *a*$*1*
proof(*induct n*)
case *0* **thus** *?case* **by** *simp*

next
case (*Suc n*)
note *h* = *Suc.hyps*[*OF* ‹*a*$*0* = *1*›]
show *?case* **unfolding** *power-Suc fps-mult-nth*
using *h* ‹*a*$*0* = *1*› *fps-power-zeroth-eq-one*[*OF* ‹*a*$*0*=*1*›] **by** (*simp add*: *field-simps*)
qed

lemma *startsby-one-power*:*a* $ *0* = (*1*::'*a*::*comm-ring-1*) ⟹ *a*^*n* $ *0* = *1*
by (*induct n*, *auto simp add*: *fps-mult-nth*)

lemma *startsby-zero-power*:*a* $*0* = (*0*::'*a*::*comm-ring-1*) ⟹ *n* > *0* ⟹ *a*^*n* $*0* = *0*
by (*induct n*, *auto simp add*: *fps-mult-nth*)

lemma *startsby-power*:*a* $*0* = (*v*::'*a*::{*comm-ring-1*}) ⟹ *a*^*n* $*0* = *v*^*n*
by (*induct n*, *auto simp add*: *fps-mult-nth power-Suc*)

lemma *startsby-zero-power-iff*[*simp*]:
a^*n* $*0* = (*0*::'*a*::{*idom*}) ⟷ (*n* ≠ *0* ∧ *a*$*0* = *0*)
apply (*rule iffI*)
apply (*induct n*, *auto simp add*: *power-Suc fps-mult-nth*)
by (*rule startsby-zero-power*, *simp-all*)

lemma *startsby-zero-power-prefix*:
assumes *a0*: *a* $*0* = (*0*::'*a*::*idom*)
shows ∀ *n* < *k*. *a* ^ *k* $ *n* = *0*
using *a0*
proof(*induct k rule*: *nat-less-induct*)
fix *k* **assume** *H*: ∀ *m*<*k*. *a* $*0* = *0* ⟶ (∀ *n*<*m*. *a* ^ *m* $ *n* = *0*) **and** *a0*: *a* $*0* = (*0*::'*a*)
let *?ths* = ∀ *m*<*k*. *a* ^ *k* $ *m* = *0*
{assume *k* = *0* **then have** *?ths* **by** *simp***}**
moreover
{fix *l* **assume** *k*: *k* = *Suc l*
{fix *m* **assume** *mk*: *m* < *k*
{assume *m*=*0* **hence** *a*^*k* $ *m* = *0* **using** *startsby-zero-power*[*of a k*] *k a0*
by *simp***}**
moreover
{assume *m0*: *m* ≠ *0*
have *a* ^*k* $ *m* = (*a*^*l* ∗ *a*) $*m* **by** (*simp add*: *k power-Suc mult-commute*)
also have ... = (∑ *i* = *0*..*m*. *a* ^ *l* $ *i* ∗ *a* $ (*m* − *i*)) **by** (*simp add*: *fps-mult-nth*)
also have ... = *0* **apply** (*rule setsum-0'*)
apply *auto*
apply (*case-tac aa* = *m*)
using *a0*
apply *simp*
apply (*rule H*[*rule-format*])

```
        using a0 k mk by auto
      finally have a^k $ m = 0 .}
    ultimately have a^k $ m = 0 by blast}
    hence ?ths by blast}
  ultimately show ?ths by (cases k, auto)
qed

lemma startsby-zero-setsum-depends:
  assumes a0: a $0 = (0::'a::idom) and kn: n ≥ k
  shows setsum (λi. (a ^ i)$k) {0 .. n} = setsum (λi. (a ^ i)$k) {0 .. k}
  apply (rule setsum-mono-zero-right)
  using kn apply auto
  apply (rule startsby-zero-power-prefix[rule-format, OF a0])
  by arith

lemma startsby-zero-power-nth-same: assumes a0: a$0 = (0::'a::{idom})
  shows a^n $ n = (a$1) ^ n
proof(induct n)
  case 0 thus ?case by (simp add: power-0)
next
  case (Suc n)
   have a ^ Suc n $ (Suc n) = (a^n * a)$(Suc n) by (simp add: field-simps
power-Suc)
  also have ... = setsum (λi. a^n$i * a $ (Suc n - i)) {0.. Suc n} by (simp
add: fps-mult-nth)
  also have ... = setsum (λi. a^n$i * a $ (Suc n - i)) {n .. Suc n}
    apply (rule setsum-mono-zero-right)
    apply simp
    apply clarsimp
    apply clarsimp
    apply (rule startsby-zero-power-prefix[rule-format, OF a0])
    apply arith
    done
  also have ... = a^n $ n * a$1 using a0 by simp
  finally show ?case using Suc.hyps by (simp add: power-Suc)
qed

lemma fps-inverse-power:
  fixes a :: ('a::{field}) fps
  shows inverse (a^n) = inverse a ^ n
proof−
  {assume a0: a$0 = 0
    hence eq: inverse a = 0 by (simp add: fps-inverse-def)
    {assume n = 0 hence ?thesis by simp}
    moreover
    {assume n: n > 0
      from startsby-zero-power[OF a0 n] eq a0 n have ?thesis
        by (simp add: fps-inverse-def)}
    ultimately have ?thesis by blast}
```

```
  moreover
  {assume a0: a$0 ≠ 0
    have ?thesis
      apply (rule fps-inverse-unique)
      apply (simp add: a0)
      unfolding power-mult-distrib[symmetric]
      apply (rule ssubst[where t = a * inverse a and s= 1])
      apply simp-all
      apply (subst mult-commute)
      by (rule inverse-mult-eq-1[OF a0])}
  ultimately show ?thesis by blast
qed

lemma fps-deriv-power: fps-deriv (a ^ n) = fps-const (of-nat n :: 'a:: comm-ring-1)
* fps-deriv a * a ^ (n - 1)
  apply (induct n, auto simp add: power-Suc field-simps fps-const-add[symmetric]
simp del: fps-const-add)
  by (case-tac n, auto simp add: power-Suc field-simps)

lemma fps-inverse-deriv:
  fixes a:: ('a :: field) fps
  assumes a0: a$0 ≠ 0
  shows fps-deriv (inverse a) = - fps-deriv a * inverse a ^ 2
proof-
  from inverse-mult-eq-1[OF a0]
  have fps-deriv (inverse a * a) = 0 by simp
  hence inverse a * fps-deriv a + fps-deriv (inverse a) * a = 0 by simp
  hence inverse a * (inverse a * fps-deriv a + fps-deriv (inverse a) * a) = 0  by
simp
  with inverse-mult-eq-1[OF a0]
  have inverse a ^ 2 * fps-deriv a + fps-deriv (inverse a) = 0
    unfolding power2-eq-square
    apply (simp add: field-simps)
    by (simp add: mult-assoc[symmetric])
  hence inverse a ^ 2 * fps-deriv a + fps-deriv (inverse a) - fps-deriv a * inverse
a ^ 2 = 0 - fps-deriv a * inverse a ^ 2
    by simp
  then show fps-deriv (inverse a) = - fps-deriv a * inverse a ^ 2 by (simp add:
field-simps)
qed

lemma fps-inverse-mult:
  fixes a::('a :: field) fps
  shows inverse (a * b) = inverse a * inverse b
proof-
  {assume a0: a$0 = 0 hence ab0: (a*b)$0 = 0 by (simp add: fps-mult-nth)
    from a0 ab0 have th: inverse a = 0 inverse (a*b) = 0 by simp-all
    have ?thesis unfolding th by simp}
  moreover
```

{assume *b0*: *b$0 = 0* **hence** *ab0*: *(a∗b)$0 = 0* **by** (*simp add*: *fps-mult-nth*)
from *b0 ab0* **have** *th*: *inverse b = 0 inverse (a∗b) = 0* **by** *simp-all*
have *?thesis* **unfolding** *th* **by** *simp***}**
moreover
{assume *a0*: *a$0 ≠ 0* **and** *b0*: *b$0 ≠ 0*
from *a0 b0* **have** *ab0*:*(a∗b) $ 0 ≠ 0* **by** (*simp add*: *fps-mult-nth*)
from *inverse-mult-eq-1*[*OF ab0*]
have *inverse (a∗b) ∗ (a∗b) ∗ inverse a ∗ inverse b = 1 ∗ inverse a ∗ inverse b* **by** *simp*
then have *inverse (a∗b) ∗ (inverse a ∗ a) ∗ (inverse b ∗ b) = inverse a ∗ inverse b*
by (*simp add*: *field-simps*)
then have *?thesis* **using** *inverse-mult-eq-1*[*OF a0*] *inverse-mult-eq-1*[*OF b0*] **by** *simp***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *fps-inverse-deriv'*:
fixes *a*:: (*'a* :: *field*) *fps*
assumes *a0*: *a$0 ≠ 0*
shows *fps-deriv* (*inverse a*) = − *fps-deriv a* / *a* ^ *2*
using *fps-inverse-deriv*[*OF a0*]
unfolding *power2-eq-square fps-divide-def*
fps-inverse-mult **by** *simp*

lemma *inverse-mult-eq-1'*: **assumes** *f0*: *f$0 ≠* (*0*::*'a*::*field*)
shows *f ∗ inverse f* = *1*
by (*metis mult-commute inverse-mult-eq-1 f0*)

lemma *fps-divide-deriv*: **fixes** *a*:: (*'a* :: *field*) *fps*
assumes *a0*: *b$0 ≠ 0*
shows *fps-deriv* (*a* / *b*) = (*fps-deriv a ∗ b* − *a ∗ fps-deriv b*) / *b* ^ *2*
using *fps-inverse-deriv*[*OF a0*]
by (*simp add*: *fps-divide-def field-simps power2-eq-square fps-inverse-mult inverse-mult-eq-1'*[*OF a0*])

lemma *fps-inverse-gp'*: *inverse* (*Abs-fps*(*λn.* (*1*::*'a*::*field*)))
= *1* − *X*
by (*simp add*: *fps-inverse-gp fps-eq-iff X-def*)

lemma *fps-nth-deriv-X*[*simp*]: *fps-nth-deriv n X* = (*if n* = *0 then X else if n*=*1 then 1 else 0*)
by (*cases n*, *simp-all*)

lemma *fps-inverse-X-plus1*:
inverse (*1* + *X*) = *Abs-fps* (*λn.* (− (*1*::*'a*::{*field*})) ^ *n*) (**is** - = *?r*)
proof−

have *eq*: $(1 + X) * ?r = 1$
unfolding *minus-one-power-iff*
by (*auto simp add*: *field-simps fps-eq-iff*)
show *?thesis* **by** (*auto simp add*: *eq intro*: *fps-inverse-unique*)
qed

39.11 Integration

definition
fps-integral :: *'a*::*field-char-0 fps* ⇒ *'a* ⇒ *'a fps* **where**
fps-integral a a0 = *Abs-fps* (λ*n*. *if n* = *0 then a0 else* (*a*$(*n* − *1*) / *of-nat n*))

lemma *fps-deriv-fps-integral*: *fps-deriv* (*fps-integral a a0*) = *a*
unfolding *fps-integral-def fps-deriv-def*
by (*simp add*: *fps-eq-iff del*: *of-nat-Suc*)

lemma *fps-integral-linear*:
fps-integral (*fps-const a* ∗ *f* + *fps-const b* ∗ *g*) (*a*∗*a0* + *b*∗*b0*) =
fps-const a ∗ *fps-integral f a0* + *fps-const b* ∗ *fps-integral g b0*
(**is** *?l* = *?r*)
proof−
have *fps-deriv ?l* = *fps-deriv ?r* **by** (*simp add*: *fps-deriv-fps-integral*)
moreover have *?l*$*0* = *?r*$*0* **by** (*simp add*: *fps-integral-def*)
ultimately show *?thesis*
unfolding *fps-deriv-eq-iff* **by** *auto*
qed

39.12 Composition of FPSs

definition *fps-compose* :: (*'a*::*semiring-1*) *fps* ⇒ *'a fps* ⇒ *'a fps* (**infixl** *oo 55*)
where
fps-compose-def: *a oo b* = *Abs-fps* (λ*n*. *setsum* (λ*i*. *a*$*i* ∗ (*b* ^*i*$*n*)) {*0*..*n*})

lemma *fps-compose-nth*: (*a oo b*)$*n* = *setsum* (λ*i*. *a*$*i* ∗ (*b* ^*i*$*n*)) {*0*..*n*} **by** (*simp add*: *fps-compose-def*)

lemma *fps-compose-X*[*simp*]: *a oo X* = (*a* :: (*'a* :: *comm-ring-1*) *fps*)
by (*simp add*: *fps-ext fps-compose-def mult-delta-right setsum-delta'*)

lemma *fps-const-compose*[*simp*]:
fps-const (*a*::*'a*::{*comm-ring-1*}) *oo b* = *fps-const* (*a*)
by (*simp add*: *fps-eq-iff fps-compose-nth mult-delta-left setsum-delta*)

lemma *number-of-compose*[*simp*]: (*number-of k*::(*'a*::{*comm-ring-1*}) *fps*) *oo b* = *number-of k*
unfolding *number-of-fps-const* **by** *simp*

lemma *X-fps-compose-startby0*[*simp*]: *a*$*0* = *0* ⟹ *X oo a* = (*a* :: (*'a* :: *comm-ring-1*) *fps*)
by (*simp add*: *fps-eq-iff fps-compose-def mult-delta-left setsum-delta*

power-Suc not-le)

39.13 Rules from Herbert Wilf's Generatingfunctionology

39.13.1 Rule 1

lemma *fps-power-mult-eq-shift*:
*X^Suc k * Abs-fps (λn. a (n + Suc k)) = Abs-fps a − setsum (λi. fps-const (a i :: 'a:: comm-ring-1) * X^i) {0 .. k}* (**is** *?lhs = ?rhs*)
proof−
{fix *n*:: *nat*
have *?lhs $ n = (if n < Suc k then 0 else a n)*
unfolding *X-power-mult-nth* **by** *auto*
also have *... = ?rhs $ n*
proof(*induct k*)
case *0* **thus** *?case* **by** (*simp add*: *fps-setsum-nth power-Suc*)
next
case (*Suc k*)
note *th = Suc.hyps*[*symmetric*]
have (*Abs-fps a − setsum (λi. fps-const (a i :: 'a) * X^i) {0 .. Suc k}*)*$n = (Abs-fps a − setsum (λi. fps-const (a i :: 'a) * X^i) {0 .. k} − fps-const (a (Suc k)) * X^ Suc k) $ n* **by** (*simp add*: *field-simps*)
also have *... = (if n < Suc k then 0 else a n) − (fps-const (a (Suc k)) * X^ Suc k)$n*
using *th*
unfolding *fps-sub-nth* **by** *simp*
also have *... = (if n < Suc (Suc k) then 0 else a n)*
unfolding *X-power-mult-right-nth*
apply (*auto simp add*: *not-less fps-const-def*)
apply (*rule cong*[*of a a*, *OF refl*])
by *arith*
finally show *?case* **by** *simp*
qed
finally have *?lhs $ n = ?rhs $ n* **.}**
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

39.13.2 Rule 2

definition *XD = op * X o fps-deriv*

lemma *XD-add*[*simp*]:*XD (a + b) = XD a + XD (b :: ('a::comm-ring-1) fps)*
by (*simp add*: *XD-def field-simps*)

lemma *XD-mult-const*[*simp*]:*XD (fps-const (c::'a::comm-ring-1) * a) = fps-const c * XD a*
by (*simp add*: *XD-def field-simps*)

lemma *XD-linear*[*simp*]: *XD (fps-const c * a + fps-const d * b) = fps-const c * XD a + fps-const d * XD (b :: ('a::comm-ring-1) fps)*

by *simp*

lemma *XDN-linear*:
(*XD* ^^ *n*) (*fps-const c* * *a* + *fps-const d* * *b*) = *fps-const c* * (*XD* ^^ *n*) *a* + *fps-const d* * (*XD* ^^ *n*) (*b* :: ('*a*::*comm-ring-1*) *fps*)
by (*induct n*, *simp-all*)

lemma *fps-mult-X-deriv-shift*: *X** *fps-deriv a* = *Abs-fps* (λ*n*. *of-nat n** *a*$*n*) **by** (*simp add*: *fps-eq-iff*)

lemma *fps-mult-XD-shift*:
(*XD* ^^ *k*) (*a*:: ('*a*::{*comm-ring-1*}) *fps*) = *Abs-fps* (λ*n*. (*of-nat n* ^ *k*) * *a*$*n*)
by (*induct k arbitrary*: *a*) (*simp-all add*: *power-Suc XD-def fps-eq-iff field-simps del*: *One-nat-def*)

39.13.3 Rule 3 is trivial and is given by *fps-times-def*

39.13.4 Rule 5 — summation and "division" by (1 - X)

lemma *fps-divide-X-minus1-setsum-lemma*:
a = ((*1*::('*a*::*comm-ring-1*) *fps*) − *X*) * *Abs-fps* (λ*n*. *setsum* (λ*i*. *a* $ *i*) {*0*..*n*})
proof−
let *?X* = *X*::('*a*::*comm-ring-1*) *fps*
let *?sa* = *Abs-fps* (λ*n*. *setsum* (λ*i*. *a* $ *i*) {*0*..*n*})
have *th0*: ⋀*i*. (*1* − (*X*::'*a fps*)) $ *i* = (*if i* = *0 then 1 else if i* = *1 then* − *1 else 0*) **by** *simp*
{fix *n*:: *nat*
{assume *n*=*0* **hence** *a*$*n* = ((*1* − *?X*) * *?sa*) $ *n*
by (*simp add*: *fps-mult-nth*)**}**
moreover
{assume *n0*: *n* ≠ *0*
then have *u*: {*0*} ∪ ({*1*} ∪ {*2*..*n*}) = {*0*..*n*} {*1*}∪{*2*..*n*} = {*1*..*n*}
{*0*..*n* − *1*}∪{*n*} = {*0*..*n*}
by (*auto simp*: *expand-set-eq*)
have *d*: {*0*} ∩ ({*1*} ∪ {*2*..*n*}) = {} {*1*} ∩ {*2*..*n*} = {}
{*0*..*n* − *1*}∩{*n*} ={} **using** *n0* **by** *simp-all*
have *f*: *finite* {*0*} *finite* {*1*} *finite* {*2* .. *n*}
finite {*0* .. *n* − *1*} *finite* {*n*} **by** *simp-all*
have ((*1* − *?X*) * *?sa*) $ *n* = *setsum* (λ*i*. (*1* − *?X*)$ *i* * *?sa* $ (*n* − *i*)) {*0* .. *n*}
by (*simp add*: *fps-mult-nth*)
also have ... = *a*$*n* **unfolding** *th0*
unfolding *setsum-Un-disjoint*[*OF f*(*1*) *finite-UnI*[*OF f*(*2*,*3*)] *d*(*1*), *unfolded u*(*1*)]
unfolding *setsum-Un-disjoint*[*OF f*(*2*) *f*(*3*) *d*(*2*)]
apply (*simp*)
unfolding *setsum-Un-disjoint*[*OF f*(*4*,*5*) *d*(*3*), *unfolded u*(*3*)]
by *simp*
finally have *a*$*n* = ((*1* − *?X*) * *?sa*) $ *n* **by** *simp***}**

ultimately have $a\$n = ((1 - ?X) * ?sa)\ \$\ n$ **by** *blast*}
then show *?thesis*
unfolding *fps-eq-iff* **by** *blast*
qed

lemma *fps-divide-X-minus1-setsum*:
$a\ /((1::('a::field)\ fps) - X) = Abs\text{-}fps\ (\lambda n.\ setsum\ (\lambda i.\ a\ \$\ i)\ \{0..n\})$
proof−
let *?X* = $1 - (X::('a::field)\ fps)$
have *th0*: $?X\ \$\ 0 \neq 0$ **by** *simp*
have $a\ /?X = ?X * Abs\text{-}fps\ (\lambda n::nat.\ setsum\ (op\ \$\ a)\ \{0..n\}) * inverse\ ?X$
using *fps-divide-X-minus1-setsum-lemma*[*of a*, *symmetric*] *th0*
by (*simp add*: *fps-divide-def mult-assoc*)
also have ... = $(inverse\ ?X * ?X) * Abs\text{-}fps\ (\lambda n::nat.\ setsum\ (op\ \$\ a)\ \{0..n\})$

by (*simp add*: *mult-ac*)
finally show *?thesis* **by** (*simp add*: *inverse-mult-eq-1*[*OF th0*])
qed

39.13.5 Rule 4 in its more general form: generalizes Rule 3 for an arbitrary finite product of FPS, also the relvant instance of powers of a FPS

definition $natpermute\ n\ k = \{l::\ nat\ list.\ length\ l = k \wedge foldl\ op\ +\ 0\ l = n\}$

lemma *natlist-trivial-1*: $natpermute\ n\ 1 = \{[n]\}$
apply (*auto simp add*: *natpermute-def*)
apply (*case-tac x*, *auto*)
done

lemma *foldl-add-start0*:
$foldl\ op\ +\ x\ xs = x + foldl\ op\ +\ (0::nat)\ xs$
apply (*induct xs arbitrary*: *x*)
apply *simp*
unfolding *foldl.simps*
apply *atomize*
apply (*subgoal-tac* $\forall x::nat.\ foldl\ op\ +\ x\ xs = x + foldl\ op\ +\ (0::nat)\ xs$)
apply (*erule-tac* $x{=}x + a$ **in** *allE*)
apply (*erule-tac* $x{=}a$ **in** *allE*)
apply *simp*
apply *assumption*
done

lemma *foldl-add-append*: $foldl\ op\ +\ (x::nat)\ (xs@ys) = foldl\ op\ +\ x\ xs + foldl\ op\ +\ 0\ ys$
apply (*induct ys arbitrary*: *x xs*)
apply *auto*
apply (*subst* (*2*) *foldl-add-start0*)
apply *simp*

apply (*subst* (*2*) *foldl-add-start0*)
by *simp*

lemma *foldl-add-setsum*: *foldl op +* (*x*::*nat*) *xs* = *x* + *setsum* (*nth xs*) {*0*..<*length xs*}
proof(*induct xs arbitrary*: *x*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a as x*)
have *eq*: *setsum* (*op* ! (*a*#*as*)) {*1*..<*length* (*a*#*as*)} = *setsum* (*op* ! *as*) {*0*..<*length as*}
apply (*rule setsum-reindex-cong* [**where** *f*=*Suc*])
by (*simp-all add*: *inj-on-def*)
have *f*: *finite* {*0*} *finite* {*1* ..< *length* (*a*#*as*)} **by** *simp-all*
have *d*: {*0*} ∩ {*1*..<*length* (*a*#*as*)} = {} **by** *simp*
have *seq*: {*0*} ∪ {*1*..<*length*(*a*#*as*)} = {*0* ..<*length* (*a*#*as*)} **by** *auto*
have *foldl op + x* (*a*#*as*) = *x* + *foldl op + a as*
apply (*subst foldl-add-start0*) **by** *simp*
also have ... = *x* + *a* + *setsum* (*op* ! *as*) {*0*..<*length as*} **unfolding** *Cons.hyps*
by *simp*
also have ... = *x* + *setsum* (*op* ! (*a*#*as*)) {*0*..<*length* (*a*#*as*)}
unfolding *eq*[*symmetric*]
unfolding *setsum-Un-disjoint*[*OF f d*, *unfolded seq*]
by *simp*
finally show *?case* **.**
qed

lemma *append-natpermute-less-eq*:
assumes *h*: *xs*@*ys* ∈ *natpermute n k* **shows** *foldl op + 0 xs* ≤ *n* **and** *foldl op + 0 ys* ≤ *n*
proof−
{from *h* **have** *foldl op + 0* (*xs*@ *ys*) = *n* **by** (*simp add*: *natpermute-def*)
hence *foldl op + 0 xs + foldl op + 0 ys* = *n* **unfolding** *foldl-add-append* **.}**
note *th* = *this*
{from *th* **show** *foldl op + 0 xs* ≤ *n* **by** *simp***}**
{from *th* **show** *foldl op + 0 ys* ≤ *n* **by** *simp***}**
qed

lemma *natpermute-split*:
assumes *mn*: *h* ≤ *k*
shows *natpermute n k* = (⋃*m* ∈{*0*..*n*}. {*l1* @ *l2* |*l1 l2*. *l1* ∈ *natpermute m h* ∧ *l2* ∈ *natpermute* (*n* − *m*) (*k* − *h*)}) (**is** *?L* = *?R* **is** *?L* = (⋃*m* ∈{*0*..*n*}. *?S m*))
proof−
{fix *l* **assume** *l*: *l* ∈ *?R*
from *l* **obtain** *m xs ys* **where** *h*: *m* ∈ {*0*..*n*} **and** *xs*: *xs* ∈ *natpermute m h* **and** *ys*: *ys* ∈ *natpermute* (*n* − *m*) (*k* − *h*) **and** *leq*: *l* = *xs*@*ys* **by** *blast*
from *xs* **have** *xs'*: *foldl op + 0 xs* = *m* **by** (*simp add*: *natpermute-def*)
from *ys* **have** *ys'*: *foldl op + 0 ys* = *n* − *m* **by** (*simp add*: *natpermute-def*)

```
    have l ∈ ?L using leq xs ys h
      apply simp
      apply (clarsimp simp add: natpermute-def simp del: foldl-append)
      apply (simp add: foldl-add-append[unfolded foldl-append])
      unfolding xs' ys'
      using mn xs ys
      unfolding natpermute-def by simp}
  moreover
  {fix l assume l: l ∈ natpermute n k
    let ?xs = take h l
    let ?ys = drop h l
    let ?m = foldl op + 0 ?xs
    from l have ls: foldl op + 0 (?xs @ ?ys) = n by (simp add: natpermute-def)
    have xs: ?xs ∈ natpermute ?m h using l mn by (simp add: natpermute-def)
   have ys: ?ys ∈ natpermute (n − ?m) (k − h) using l mn ls[unfolded foldl-add-append]
      by (simp add: natpermute-def)
    from ls have m: ?m ∈ {0..n}  unfolding foldl-add-append by simp
    from xs ys ls have l ∈ ?R
      apply auto
      apply (rule bexI[where x = ?m])
      apply (rule exI[where x = ?xs])
      apply (rule exI[where x = ?ys])
      using ls l unfolding foldl-add-append
      by (auto simp add: natpermute-def)}
  ultimately show ?thesis by blast
qed

lemma natpermute-0: natpermute n 0 = (if n = 0 then {[]} else {})
  by (auto simp add: natpermute-def)
lemma natpermute-0'[simp]: natpermute 0 k = (if k = 0 then {[]} else {replicate
k 0})
  apply (auto simp add: set-replicate-conv-if natpermute-def)
  apply (rule nth-equalityI)
  by simp-all

lemma natpermute-finite: finite (natpermute n k)
proof(induct k arbitrary: n)
  case 0 thus ?case
    apply (subst natpermute-split[of 0 0, simplified])
    by (simp add: natpermute-0)
next
  case (Suc k)
  then show ?case unfolding natpermute-split[of k Suc k, simplified]
    apply −
    apply (rule finite-UN-I)
    apply simp
    unfolding One-nat-def[symmetric] natlist-trivial-1
    apply simp
    done
```

qed

lemma *natpermute-contain-maximal*:
{*xs* ∈ *natpermute n* (*k*+*1*). *n* ∈ *set xs*} = *UNION* {*0* .. *k*} (λ*i*. {(*replicate* (*k*+*1*) *0*) [*i*:=*n*]})
(**is** *?A* = *?B*)
proof−
{fix *xs* **assume** *H*: *xs* ∈ *natpermute n* (*k*+*1*) **and** *n*: *n* ∈ *set xs*
from *n* **obtain** *i* **where** *i*: *i* ∈ {*0*.. *k*} *xs*!*i* = *n* **using** *H*
unfolding *in-set-conv-nth* **by** (*auto simp add*: *less-Suc-eq-le natpermute-def*)
have *eqs*: ({*0*..*k*} − {*i*}) ∪ {*i*} = {*0*..*k*} **using** *i* **by** *auto*
have *f*: *finite*({*0*..*k*} − {*i*}) *finite* {*i*} **by** *auto*
have *d*: ({*0*..*k*} − {*i*}) ∩ {*i*} = {} **using** *i* **by** *auto*
from *H* **have** *n* = *setsum* (*nth xs*) {*0*..*k*} **apply** (*simp add*: *natpermute-def*)
unfolding *foldl-add-setsum* **by** (*auto simp add*: *atLeastLessThanSuc-atLeastAtMost*)
also have ... = *n* + *setsum* (*nth xs*) ({*0*..*k*} − {*i*})
unfolding *setsum-Un-disjoint*[*OF f d*, *unfolded eqs*] **using** *i* **by** *simp*
finally have *zxs*: ∀ *j*∈ {*0*..*k*} − {*i*}. *xs*!*j* = *0* **by** *auto*
from *H* **have** *xsl*: *length xs* = *k*+*1* **by** (*simp add*: *natpermute-def*)
from *i* **have** *i'*: *i* < *length* (*replicate* (*k*+*1*) *0*) *i* < *k*+*1*
unfolding *length-replicate* **by** *arith*+
have *xs* = *replicate* (*k*+*1*) *0* [*i* := *n*]
apply (*rule nth-equalityI*)
unfolding *xsl length-list-update length-replicate*
apply *simp*
apply *clarify*
unfolding *nth-list-update*[*OF i'*(*1*)]
using *i zxs*
by (*case-tac ia*=*i*, *auto simp del*: *replicate.simps*)
then have *xs* ∈ *?B* **using** *i* **by** *blast*}
moreover
{fix *i* **assume** *i*: *i* ∈ {*0*..*k*}
let *?xs* = *replicate* (*k*+*1*) *0* [*i*:=*n*]
have *nxs*: *n* ∈ *set ?xs*
apply (*rule set-update-memI*) **using** *i* **by** *simp*
have *xsl*: *length ?xs* = *k*+*1* **by** (*simp only*: *length-replicate length-list-update*)
have *foldl op* + *0 ?xs* = *setsum* (*nth ?xs*) {*0*..<*k*+*1*}
unfolding *foldl-add-setsum add-0 length-replicate length-list-update* **..**
also have ... = *setsum* (λ*j*. *if j* = *i then n else 0*) {*0*..< *k*+*1*}
apply (*rule setsum-cong2*) **by** (*simp del*: *replicate.simps*)
also have ... = *n* **using** *i* **by** (*simp add*: *setsum-delta*)
finally
have *?xs* ∈ *natpermute n* (*k*+*1*) **using** *xsl* **unfolding** *natpermute-def Collect-def mem-def*
by *blast*
then have *?xs* ∈ *?A* **using** *nxs* **by** *blast*}
ultimately show *?thesis* **by** *auto*
qed

lemma *fps-setprod-nth*:
fixes $m :: nat$ **and** $a :: nat \Rightarrow ('a::comm\text{-}ring\text{-}1)\ fps$
shows $(setprod\ a\ \{0\ ..\ m\})\$n = setsum\ (\lambda v.\ setprod\ (\lambda j.\ (a\ j)\ \$\ (v!j))\ \{0..m\})$
$(natpermute\ n\ (m+1))$
(**is** *?P m n*)
proof(*induct m arbitrary*: *n rule*: *nat-less-induct*)
fix *m n* **assume** *H*: $\forall m' < m.\ \forall n.\ ?P\ m'\ n$
{assume *m0*: $m = 0$
hence *?P m n* **apply** *simp*
unfolding *natlist-trivial-1*[**where** $n = n$, *unfolded One-nat-def*] **by** *simp***}**
moreover
{fix *k* **assume** *k*: $m = Suc\ k$
have *km*: $k < m$ **using** *k* **by** *arith*
have *u0*: $\{0\ ..\ k\} \cup \{m\} = \{0..m\}$ **using** *k* **apply** (*simp add*: *expand-set-eq*)
by *presburger*
have *f0*: *finite* $\{0\ ..\ k\}$ *finite* $\{m\}$ **by** *auto*
have *d0*: $\{0\ ..\ k\} \cap \{m\} = \{\}$ **using** *k* **by** *auto*
have $(setprod\ a\ \{0\ ..\ m\})\ \$\ n = (setprod\ a\ \{0\ ..\ k\} * a\ m)\ \$\ n$
unfolding *setprod-Un-disjoint*[*OF f0 d0*, *unfolded u0*] **by** *simp*
also have $\ldots = (\sum i = 0..n.\ (\sum v \in natpermute\ i\ (k+1).\ \prod j \in \{0..k\}.\ a\ j\ \$$
$v\ !\ j) * a\ m\ \$\ (n - i))$
unfolding *fps-mult-nth H*[*rule-format*, *OF km*] **..**
also have $\ldots = (\sum v \in natpermute\ n\ (m+1).\ \prod j \in \{0..m\}.\ a\ j\ \$\ v\ !\ j)$
apply (*simp add*: *k*)
unfolding *natpermute-split*[*of m m + 1*, *simplified*, *of n*, *unfolded natlist-trivial-1*[*unfolded*
One-nat-def] *k*]
apply (*subst setsum-UN-disjoint*)
apply *simp*
apply *simp*
unfolding *image-Collect*[*symmetric*]
apply *clarsimp*
apply (*rule finite-imageI*)
apply (*rule natpermute-finite*)
apply (*clarsimp simp add*: *expand-set-eq*)
apply *auto*
apply (*rule setsum-cong2*)
unfolding *setsum-left-distrib*
apply (*rule sym*)
apply (*rule-tac f*=$\lambda xs.\ xs\ @[n - x]$ **in** *setsum-reindex-cong*)
apply (*simp add*: *inj-on-def*)
apply *auto*
unfolding *setprod-Un-disjoint*[*OF f0 d0*, *unfolded u0*, *unfolded k*]
apply (*clarsimp simp add*: *natpermute-def nth-append*)
done
finally have *?P m n* **.}**
ultimately show *?P m n* **by** (*cases m*, *auto*)
qed

The special form for powers

lemma *fps-power-nth-Suc*:
fixes $m :: nat$ **and** $a :: ('a::comm\text{-}ring\text{-}1)\ fps$
shows $(a\ \hat{}\ Suc\ m)\$n = setsum\ (\lambda v.\ setprod\ (\lambda j.\ a\ \$\ (v!j))\ \{0..m\})\ (natpermute\ n\ (m+1))$
proof−
have *f*: *finite* $\{0\ ..m\}$ **by** *simp*
have *th0*: $a\hat{}Suc\ m = setprod\ (\lambda i.\ a)\ \{0..m\}$ **unfolding** *setprod-constant*[*OF f*, *of a*] **by** *simp*
show *?thesis* **unfolding** *th0 fps-setprod-nth* **..**
qed
lemma *fps-power-nth*:
fixes $m :: nat$ **and** $a :: ('a::comm\text{-}ring\text{-}1)\ fps$
shows $(a\ \hat{}m)\$n = (if\ m=0\ then\ 1\$n\ else\ setsum\ (\lambda v.\ setprod\ (\lambda j.\ a\ \$\ (v!j))\ \{0..m - 1\})\ (natpermute\ n\ m))$
by (*cases m*, *simp-all add*: *fps-power-nth-Suc del*: *power-Suc*)

lemma *fps-nth-power-0*:
fixes $m :: nat$ **and** $a :: ('a::\{comm\text{-}ring\text{-}1\})\ fps$
shows $(a\ \hat{}m)\$0 = (a\$0)\ \hat{}\ m$
proof−
{assume *m*=*0* **hence** *?thesis* **by** *simp***}**
moreover
{fix *n* **assume** *m*: $m = Suc\ n$
have *c*: $m = card\ \{0..n\}$ **using** *m* **by** *simp*
have $(a\ \hat{}m)\$0 = setprod\ (\lambda i.\ a\$0)\ \{0..n\}$
by (*simp add*: *m fps-power-nth del*: *replicate.simps power-Suc*)
also have $\ldots = (a\$0)\ \hat{}\ m$
unfolding *c* **by** (*rule setprod-constant*, *simp*)
finally have *?thesis* **.}**
ultimately show *?thesis* **by** (*cases m*, *auto*)
qed

lemma *fps-compose-inj-right*:
assumes *a0*: $a\$0 = (0::'a::\{idom\})$
and *a1*: $a\$1 \neq 0$
shows $(b\ oo\ a = c\ oo\ a) \longleftrightarrow b = c$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof−
{assume *?rhs* **then have** *?lhs* **by** *simp***}**
moreover
{assume *h*: *?lhs*
{fix *n* **have** $b\$n = c\$n$
proof(*induct n rule*: *nat-less-induct*)
fix *n* **assume** *H*: $\forall m<n.\ b\$m = c\$m$
{assume *n0*: *n*=*0*
from *h* **have** $(b\ oo\ a)\$n = (c\ oo\ a)\$n$ **by** *simp*
hence $b\$n = c\$n$ **using** *n0* **by** (*simp add*: *fps-compose-nth*)**}**
moreover
{fix *n1* **assume** *n1*: $n = Suc\ n1$
have *f*: *finite* $\{0\ ..\ n1\}$ *finite* $\{n\}$ **by** *simp-all*

```
        have eq: {0 .. n1} ∪ {n} = {0 .. n} using n1 by auto
        have d: {0 .. n1} ∩ {n} = {} using n1 by auto
        have seq: (∑ i = 0..n1. b $ i * a ^ i $ n) = (∑ i = 0..n1. c $ i * a ^ i
$ n)
          apply (rule setsum-cong2)
          using H n1 by auto
       have th0: (b oo a) $n = (∑ i = 0..n1. c $ i * a ^ i $ n) + b$n * (a$1) ^n
          unfolding fps-compose-nth setsum-Un-disjoint[OF f d, unfolded eq] seq
          using startsby-zero-power-nth-same[OF a0]
          by simp
       have th1: (c oo a) $n = (∑ i = 0..n1. c $ i * a ^ i $ n) + c$n * (a$1) ^n
          unfolding fps-compose-nth setsum-Un-disjoint[OF f d, unfolded eq]
          using startsby-zero-power-nth-same[OF a0]
          by simp
        from h[unfolded fps-eq-iff, rule-format, of n] th0 th1 a1
        have b$n = c$n by auto}
      ultimately show b$n = c$n by (cases n, auto)
    qed}
  then have ?rhs by (simp add: fps-eq-iff)}
 ultimately show ?thesis by blast
qed
```

39.14 Radicals

```
declare setprod-cong[fundef-cong]
function radical :: (nat ⇒ 'a ⇒ 'a) ⇒ nat ⇒ ('a::{field}) fps ⇒ nat ⇒ 'a where
  radical r 0 a 0 = 1
| radical r 0 a (Suc n) = 0
| radical r (Suc k) a 0 = r (Suc k) (a$0)
| radical r (Suc k) a (Suc n) = (a$ Suc n − setsum (λxs. setprod (λj. radical r
(Suc k) a (xs ! j)) {0..k}) {xs. xs ∈ natpermute (Suc n) (Suc k) ∧ Suc n ∉ set
xs}) / (of-nat (Suc k) * (radical r (Suc k) a 0) ^k)
by pat-completeness auto

termination radical
proof
  let ?R = measure (λ(r, k, a, n). n)
  {
    show wf ?R by auto}
  {fix r k a n xs i
    assume xs: xs ∈ {xs ∈ natpermute (Suc n) (Suc k). Suc n ∉ set xs} and i: i
∈ {0..k}
    {assume c: Suc n ≤ xs ! i
    from xs i have xs !i ≠ Suc n by (auto simp add: in-set-conv-nth natpermute-def)
      with c have c': Suc n < xs!i by arith
      have fths: finite {0 ..< i} finite {i} finite {i+1..<Suc k} by simp-all
      have d: {0 ..< i} ∩ ({i} ∪ {i+1 ..< Suc k}) = {} {i} ∩ {i+1..< Suc k} =
{} by auto
       have eqs: {0..<Suc k} = {0 ..< i} ∪ ({i} ∪ {i+1 ..< Suc k}) using i by
```

```
auto
      from xs have Suc n = foldl op + 0 xs by (simp add: natpermute-def)
       also have ... = setsum (nth xs) {0..<Suc k} unfolding foldl-add-setsum
using xs
        by (simp add: natpermute-def)
     also have ... = xs!i + setsum (nth xs) {0..<i} + setsum (nth xs) {i+1..<Suc
k}
          unfolding eqs  setsum-Un-disjoint[OF fths(1) finite-UnI[OF fths(2,3)]
d(1)]
        unfolding setsum-Un-disjoint[OF fths(2) fths(3) d(2)]
        by simp
      finally have False using c' by simp}
    then show ((r,Suc k,a,xs!i), r, Suc k, a, Suc n) ∈ ?R
      apply auto by (metis not-less)}
  {fix r k a n
    show ((r,Suc k, a, 0),r, Suc k, a, Suc n) ∈ ?R by simp}
qed

definition fps-radical r n a = Abs-fps (radical r n a)

lemma fps-radical0[simp]: fps-radical r 0 a = 1
  apply (auto simp add: fps-eq-iff fps-radical-def)  by (case-tac n, auto)

lemma fps-radical-nth-0[simp]: fps-radical r n a $ 0 = (if n=0 then 1 else r n
(a$0))
  by (cases n, simp-all add: fps-radical-def)

lemma fps-radical-power-nth[simp]:
  assumes r: (r k (a$0)) ^ k = a$0
  shows fps-radical r k a ^ k $ 0 = (if k = 0 then 1 else a$0)
proof−
  {assume k=0 hence ?thesis by simp }
  moreover
  {fix h assume h: k = Suc h
    have fh: finite {0..h} by simp
    have eq1: fps-radical r k a ^ k $ 0 = (∏j∈{0..h}. fps-radical r k a $ (replicate
k 0) ! j)
      unfolding fps-power-nth h by simp
    also have ... = (∏j∈{0..h}. r k (a$0))
      apply (rule setprod-cong)
      apply simp
      using h
      apply (subgoal-tac replicate k (0::nat) ! x = 0)
      by (auto intro: nth-replicate simp del: replicate.simps)
    also have ... = a$0
      unfolding setprod-constant[OF fh] using r by (simp add: h)
    finally have ?thesis using h by simp}
  ultimately show ?thesis by (cases k, auto)
qed
```

lemma *natpermute-max-card*: **assumes** *n0*: $n \neq 0$
shows *card* $\{xs \in natpermute\ n\ (k+1).\ n \in set\ xs\} = k+1$
unfolding *natpermute-contain-maximal*
proof−
let *?A*= $\lambda i.\ \{replicate\ (k + 1)\ 0[i := n]\}$
let *?K* = $\{0\ ..k\}$
have *fK*: *finite ?K* **by** *simp*
have *fAK*: $\forall i \in ?K.$ *finite* (*?A i*) **by** *auto*
have *d*: $\forall i \in ?K.\ \forall j \in ?K.\ i \neq j \longrightarrow \{replicate\ (k + 1)\ 0[i := n]\} \cap \{replicate\ (k + 1)\ 0[j := n]\} = \{\}$
proof(*clarify*)
fix *i j* **assume** *i*: $i \in ?K$ **and** *j*: $j \in ?K$ **and** *ij*: $i \neq j$
{assume *eq*: *replicate* $(k+1)$ *0* $[i:=n]$ = *replicate* $(k+1)$ *0* $[j:= n]$
have (*replicate* $(k+1)$ *0* $[i:=n]$! *i*) = *n* **using** *i* **by** (*simp del*: *replicate.simps*)
moreover
have (*replicate* $(k+1)$ *0* $[j:=n]$! *i*) = *0* **using** *i ij* **by** (*simp del*: *replicate.simps*)
ultimately have *False* **using** *eq n0* **by** (*simp del*: *replicate.simps*)**}**
then show $\{replicate\ (k + 1)\ 0[i := n]\} \cap \{replicate\ (k + 1)\ 0[j := n]\} = \{\}$
by *auto*
qed
from *card-UN-disjoint*[*OF fK fAK d*]
show *card* $(\bigcup i \in \{0..k\}.\ \{replicate\ (k + 1)\ 0[i := n]\}) = k+1$ **by** *simp*
qed

lemma *power-radical*:
fixes *a*:: *'a*::*field-char-0 fps*
assumes *a0*: $a\$0 \neq 0$
shows $(r\ (Suc\ k)\ (a\$0))\ \hat{}\ Suc\ k = a\$0 \longleftrightarrow (fps\text{-}radical\ r\ (Suc\ k)\ a)\ \hat{}\ (Suc\ k) = a$
proof−
let *?r* = *fps-radical r* (*Suc k*) *a*
{assume *r0*: $(r\ (Suc\ k)\ (a\$0))\ \hat{}\ Suc\ k = a\$0$
from *a0 r0* **have** *r00*: $r\ (Suc\ k)\ (a\$0) \neq 0$ **by** *auto*
{fix *z* **have** $?r\ \hat{}\ Suc\ k\ \$\ z = a\$z$
proof(*induct z rule*: *nat-less-induct*)
fix *n* **assume** *H*: $\forall m<n.\ ?r\ \hat{}\ Suc\ k\ \$\ m = a\$m$
{assume $n = 0$ **hence** $?r\ \hat{}\ Suc\ k\ \$\ n = a\ \$n$
using *fps-radical-power-nth*[*of r Suc k a*, *OF r0*] **by** *simp***}**
moreover
{fix *n1* **assume** *n1*: $n = Suc\ n1$
have *fK*: *finite* $\{0..k\}$ **by** *simp*
have *nz*: $n \neq 0$ **using** *n1* **by** *arith*
let *?Pnk* = *natpermute n* $(k + 1)$
let *?Pnkn* = $\{xs \in ?Pnk.\ n \in set\ xs\}$
let *?Pnknn* = $\{xs \in ?Pnk.\ n \notin set\ xs\}$
have *eq*: $?Pnkn \cup ?Pnknn = ?Pnk$ **by** *blast*
have *d*: $?Pnkn \cap ?Pnknn = \{\}$ **by** *blast*

```
        have f: finite ?Pnkn finite ?Pnknn
          using finite-Un[of ?Pnkn ?Pnknn, unfolded eq]
          by (metis natpermute-finite)+
        let ?f = λv. ∏j∈{0..k}. ?r $ v ! j
         have setsum ?f ?Pnkn = setsum (λv. ?r $ n * r (Suc k) (a $ 0) ^ k)
?Pnkn
        proof(rule setsum-cong2)
          fix v assume v: v ∈ {xs ∈ natpermute n (k + 1). n ∈ set xs}
           let ?ths = (∏j∈{0..k}. fps-radical r (Suc k) a $ v ! j) = fps-radical r
(Suc k) a $ n * r (Suc k) (a $ 0) ^ k
        from v obtain i where i: i ∈ {0..k} v = replicate (k+1) 0 [i:= n]
          unfolding natpermute-contain-maximal by auto
        have (∏j∈{0..k}. fps-radical r (Suc k) a $ v ! j) = (∏j∈{0..k}. if j =
i then fps-radical r (Suc k) a $ n else r (Suc k) (a$0))
          apply (rule setprod-cong, simp)
          using i r0 by (simp del: replicate.simps)
        also have ... = (fps-radical r (Suc k) a $ n) * r (Suc k) (a$0) ^ k
          unfolding setprod-gen-delta[OF fK] using i r0 by simp
        finally show ?ths .
      qed
      then have setsum ?f ?Pnkn = of-nat (k+1) * ?r $ n * r (Suc k) (a $ 0)
^ k
        by (simp add: natpermute-max-card[OF nz, simplified])
      also have ... = a$n − setsum ?f ?Pnknn
        unfolding n1 using r00 a0 by (simp add: field-simps fps-radical-def del:
of-nat-Suc)
      finally have fn: setsum ?f ?Pnkn = a$n − setsum ?f ?Pnknn .
      have (?r ^ Suc k)$n = setsum ?f ?Pnkn + setsum ?f ?Pnknn
        unfolding fps-power-nth-Suc setsum-Un-disjoint[OF f d, unfolded eq] ..
      also have ... = a$n unfolding fn by simp
      finally have ?r ^ Suc k $ n = a $n .}
    ultimately  show ?r ^ Suc k $ n = a $n by (cases n, auto)
  qed }
 then have ?thesis using r0 by (simp add: fps-eq-iff)}
moreover
{ assume h: (fps-radical r (Suc k) a) ^ (Suc k) = a
 hence ((fps-radical r (Suc k) a) ^ (Suc k))$0 = a$0 by simp
 then have (r (Suc k) (a$0)) ^ Suc k = a$0
   unfolding fps-power-nth-Suc
   by (simp add: setprod-constant del: replicate.simps)}
ultimately show ?thesis by blast
qed


lemma eq-divide-imp': assumes c0: (c::'a::field) ~= 0 and eq: a * c = b
 shows a = b / c
proof−
 from eq have a * c * inverse c = b * inverse c by simp
 hence a * (inverse c * c) = b/c by (simp only: field-simps divide-inverse)
```

then show $a = b/c$ **unfolding** *field-inverse*[*OF c0*] **by** *simp*
qed

lemma *radical-unique*:
assumes *r0*: $(r\ (Suc\ k)\ (b\$0))\ \hat{}\ Suc\ k = b\$0$
and *a0*: $r\ (Suc\ k)\ (b\$0 :: 'a :: field\text{-}char\text{-}0) = a\$0$ **and** *b0*: $b\$0 \neq 0$
shows $a\hat{}(Suc\ k) = b \longleftrightarrow a = fps\text{-}radical\ r\ (Suc\ k)\ b$
proof−
let *?r* = *fps-radical r* (*Suc k*) *b*
have *r00*: $r\ (Suc\ k)\ (b\$0) \neq 0$ **using** *b0 r0* **by** *auto*
{assume *H*: $a = ?r$
from *H* **have** $a\hat{}Suc\ k = b$ **using** *power-radical*[*OF b0*, *of r k*, *unfolded r0*] **by** *simp*}
moreover
{assume *H*: $a\hat{}Suc\ k = b$
have *ceq*: *card* {0..*k*} = *Suc k* **by** *simp*
have *fk*: *finite* {0..*k*} **by** *simp*
from *a0* **have** *a0r0*: $a\$0 = ?r\$0$ **by** *simp*
{fix *n* **have** $a\ \$\ n = ?r\ \$\ n$
proof(*induct n rule*: *nat-less-induct*)
fix *n* **assume** *h*: $\forall m<n.\ a\$m = ?r\ \$m$
{assume $n = 0$ **hence** $a\$n = ?r\ \$n$ **using** *a0* **by** *simp* **}**
moreover
{fix *n1* **assume** *n1*: $n = Suc\ n1$
have *fK*: *finite* {0..*k*} **by** *simp*
have *nz*: $n \neq 0$ **using** *n1* **by** *arith*
let *?Pnk* = *natpermute n* (*Suc k*)
let *?Pnkn* = $\{xs \in ?Pnk.\ n \in set\ xs\}$
let *?Pnknn* = $\{xs \in ?Pnk.\ n \notin set\ xs\}$
have *eq*: *?Pnkn* ∪ *?Pnknn* = *?Pnk* **by** *blast*
have *d*: *?Pnkn* ∩ *?Pnknn* = {} **by** *blast*
have *f*: *finite ?Pnkn finite ?Pnknn*
using *finite-Un*[*of ?Pnkn ?Pnknn*, *unfolded eq*]
by (*metis natpermute-finite*)+
let *?f* = $\lambda v.\ \prod j{\in}\{0..k\}.\ ?r\ \$\ v\ !\ j$
let *?g* = $\lambda v.\ \prod j{\in}\{0..k\}.\ a\ \$\ v\ !\ j$
have *setsum ?g ?Pnkn* = *setsum* $(\lambda v.\ a\ \$\ n * (?r\$0)\ \hat{}\ k)$ *?Pnkn*
proof(*rule setsum-cong2*)
fix *v* **assume** *v*: $v \in \{xs \in natpermute\ n\ (Suc\ k).\ n \in set\ xs\}$
let *?ths* = $(\prod j{\in}\{0..k\}.\ a\ \$\ v\ !\ j) = a\ \$\ n * (?r\$0)\ \hat{}\ k$
from *v* **obtain** *i* **where** *i*: $i \in \{0..k\}$ $v = replicate\ (k+1)\ 0\ [i:= n]$
unfolding *Suc-eq-plus1 natpermute-contain-maximal* **by** (*auto simp del*: *replicate.simps*)
have $(\prod j{\in}\{0..k\}.\ a\ \$\ v\ !\ j) = (\prod j{\in}\{0..k\}.\ if\ j = i\ then\ a\ \$\ n\ else\ r\ (Suc\ k)\ (b\$0))$
apply (*rule setprod-cong*, *simp*)
using *i a0* **by** (*simp del*: *replicate.simps*)
also have $\ldots = a\ \$\ n * (?r\ \$\ 0)\ \hat{}\ k$
unfolding *setprod-gen-delta*[*OF fK*] **using** *i* **by** *simp*

```
      finally show ?ths .
    qed
    then have th0: setsum ?g ?Pnkn = of-nat (k+1) * a $ n * (?r $ 0) ^ k
      by (simp add: natpermute-max-card[OF nz, simplified])
    have th1: setsum ?g ?Pnknn = setsum ?f ?Pnknn
    proof (rule setsum-cong2, rule setprod-cong, simp)
      fix xs i assume xs: xs ∈ ?Pnknn and i: i ∈ {0..k}
      {assume c: n ≤ xs ! i
            from xs i have xs !i ≠ n by (auto simp add: in-set-conv-nth
natpermute-def)
        with c have c': n < xs!i by arith
        have fths: finite {0 ..< i} finite {i} finite {i+1..<Suc k} by simp-all
        have d: {0 ..< i} ∩ ({i} ∪ {i+1 ..< Suc k}) = {} {i} ∩ {i+1..< Suc
k} = {} by auto
        have eqs: {0..<Suc k} = {0 ..< i} ∪ ({i} ∪ {i+1 ..< Suc k}) using i
by auto
        from xs have n = foldl op + 0 xs by (simp add: natpermute-def)
       also have ... = setsum (nth xs) {0..<Suc k} unfolding foldl-add-setsum
using xs
          by (simp add: natpermute-def)
         also have ... = xs!i + setsum (nth xs) {0..<i} + setsum (nth xs)
{i+1..<Suc k}
         unfolding eqs  setsum-Un-disjoint[OF fths(1) finite-UnI[OF fths(2,3)]
d(1)]
          unfolding setsum-Un-disjoint[OF fths(2) fths(3) d(2)]
          by simp
        finally have False using c' by simp}
      then have thn: xs!i < n by arith
      from h[rule-format, OF thn]
      show a$(xs !i) = ?r$(xs!i) .
    qed
    have th00: ⋀(x::'a). of-nat (Suc k) * (x * inverse (of-nat (Suc k))) = x
      by (simp add: field-simps del: of-nat-Suc)
    from H have b$n = a ^ Suc k $ n by (simp add: fps-eq-iff)
    also have a ^ Suc k$n = setsum ?g ?Pnkn + setsum ?g ?Pnknn
      unfolding fps-power-nth-Suc
      using setsum-Un-disjoint[OF f d, unfolded Suc-eq-plus1[symmetric],
        unfolded eq, of ?g] by simp
     also have ... = of-nat (k+1) * a $ n * (?r $ 0) ^ k + setsum ?f ?Pnknn
unfolding th0 th1 ..
    finally have of-nat (k+1) * a $ n * (?r $ 0) ^ k = b$n - setsum ?f ?Pnknn
by simp
   then have a$n = (b$n - setsum ?f ?Pnknn) / (of-nat (k+1) * (?r $ 0) ^ k)
      apply -
      apply (rule eq-divide-imp')
      using r00
      apply (simp del: of-nat-Suc)
      by (simp add: mult-ac)
    then have a$n = ?r $n
```

```
        apply (simp del: of-nat-Suc)
        unfolding fps-radical-def n1
        by (simp add: field-simps n1 th00 del: of-nat-Suc)}
      ultimately show a$n = ?r $ n by (cases n, auto)
    qed}
  then have a = ?r by (simp add: fps-eq-iff)}
 ultimately show ?thesis by blast
qed


lemma radical-power:
 assumes r0: r (Suc k) ((a$0) ^ Suc k) = a$0
 and a0: (a$0 ::'a::field-char-0) ≠ 0
 shows (fps-radical r (Suc k) (a ^ Suc k)) = a
proof−
 let ?ak = a^ Suc k
 have ak0: ?ak $ 0 = (a$0) ^ Suc k by (simp add: fps-nth-power-0 del: power-Suc)
 from r0 have th0: r (Suc k) (a ^ Suc k $ 0) ^ Suc k = a ^ Suc k $ 0 using
ak0 by auto
 from r0 ak0 have th1: r (Suc k) (a ^ Suc k $ 0) = a $ 0 by auto
 from ak0 a0 have ak00: ?ak $ 0 ≠0 by auto
 from radical-unique[of r k ?ak a, OF th0 th1 ak00] show ?thesis by metis
qed

lemma fps-deriv-radical:
 fixes a:: 'a::field-char-0 fps
 assumes r0: (r (Suc k) (a$0)) ^ Suc k = a$0 and a0: a$0 ≠ 0
 shows fps-deriv (fps-radical r (Suc k) a) = fps-deriv a / (fps-const (of-nat (Suc
k)) ∗ (fps-radical r (Suc k) a) ^ k)
proof−
 let ?r= fps-radical r (Suc k) a
 let ?w = (fps-const (of-nat (Suc k)) ∗ ?r ^ k)
 from a0 r0 have r0': r (Suc k) (a$0) ≠ 0 by auto
 from r0' have w0: ?w $ 0 ≠ 0 by (simp del: of-nat-Suc)
 note th0 = inverse-mult-eq-1[OF w0]
 let ?iw = inverse ?w
 from iffD1[OF power-radical[of a r], OF a0 r0]
 have fps-deriv (?r ^ Suc k) = fps-deriv a by simp
 hence fps-deriv ?r ∗ ?w = fps-deriv a
   by (simp add: fps-deriv-power mult-ac del: power-Suc)
 hence ?iw ∗ fps-deriv ?r ∗ ?w = ?iw ∗ fps-deriv a by simp
 hence fps-deriv ?r ∗ (?iw ∗ ?w) = fps-deriv a / ?w
   by (simp add: fps-divide-def)
 then show ?thesis unfolding th0 by simp
qed

lemma radical-mult-distrib:
 fixes a:: 'a::field-char-0 fps
 assumes
```

k: $k > 0$
and *ra0*: $r\ k\ (a\ \$\ 0)\ \hat{}\ k = a\ \$\ 0$
and *rb0*: $r\ k\ (b\ \$\ 0)\ \hat{}\ k = b\ \$\ 0$
and *a0*: $a\$0 \neq 0$
and *b0*: $b\$0 \neq 0$
shows $r\ k\ ((a * b)\ \$\ 0) = r\ k\ (a\ \$\ 0) * r\ k\ (b\ \$\ 0) \longleftrightarrow$ *fps-radical r* $(k)\ (a*b) =$ *fps-radical r* $(k)\ a *$ *fps-radical r* $(k)\ (b)$
proof−
{assume *r0′*: $r\ k\ ((a * b)\ \$\ 0) = r\ k\ (a\ \$\ 0) * r\ k\ (b\ \$\ 0)$
from *r0′* **have** *r0*: $(r\ (k)\ ((a*b)\$0))\ \hat{}\ k = (a*b)\$0$
by (*simp add*: *fps-mult-nth ra0 rb0 power-mult-distrib*)
{assume $k=0$ **hence** *?thesis* **using** *r0′* **by** *simp*}
moreover
{fix *h* **assume** *k*: $k =$ *Suc h*
let *?ra* = *fps-radical r* (*Suc h*) *a*
let *?rb* = *fps-radical r* (*Suc h*) *b*
have *th0*: *r* (*Suc h*) $((a * b)\ \$\ 0) =$ (*fps-radical r* (*Suc h*) *a* ∗ *fps-radical r* (*Suc h*) *b*) $\$\ 0$
using *r0′ k* **by** (*simp add*: *fps-mult-nth*)
have *ab0*: $(a*b)\ \$\ 0 \neq 0$ **using** *a0 b0* **by** (*simp add*: *fps-mult-nth*)
from *radical-unique*[*of r h a∗b fps-radical r* (*Suc h*) *a* ∗ *fps-radical r* (*Suc h*) *b*, *OF r0*[*unfolded k*] *th0 ab0*, *symmetric*]
iffD1[*OF power-radical*[*of - r*], *OF a0 ra0*[*unfolded k*]] *iffD1*[*OF power-radical*[*of - r*], *OF b0 rb0*[*unfolded k*]] *k r0′*
have *?thesis* **by** (*auto simp add*: *power-mult-distrib simp del*: *power-Suc*)}
ultimately have *?thesis* **by** (*cases k*, *auto*)}
moreover
{assume *h*: *fps-radical r k* $(a*b) =$ *fps-radical r k a* ∗ *fps-radical r k b*
hence (*fps-radical r k* $(a*b))\$0 =$ (*fps-radical r k a* ∗ *fps-radical r k b*)$\$0$ **by** *simp*
then have $r\ k\ ((a * b)\ \$\ 0) = r\ k\ (a\ \$\ 0) * r\ k\ (b\ \$\ 0)$
using *k* **by** (*simp add*: *fps-mult-nth*)}
ultimately show *?thesis* **by** *blast*
qed

lemma *fps-divide-1*[*simp*]: (*a*:: (*′a*::*field*) *fps*) / *1* = *a*
by (*simp add*: *fps-divide-def*)

lemma *radical-divide*:
fixes *a* :: *′a*::*field-char-0 fps*
assumes
kp: *k*>*0*
and *ra0*: $(r\ k\ (a\ \$\ 0))\ \hat{}\ k = a\ \$\ 0$
and *rb0*: $(r\ k\ (b\ \$\ 0))\ \hat{}\ k = b\ \$\ 0$
and *a0*: $a\$0 \neq 0$
and *b0*: $b\$0 \neq 0$
shows $r\ k\ ((a\ \$\ 0)\ /\ (b\$0)) = r\ k\ (a\$0)\ /\ r\ k\ (b\ \$\ 0) \longleftrightarrow$ *fps-radical r k* (a/b)

= *fps-radical r k a / fps-radical r k b* (**is** *?lhs = ?rhs*)
proof−
let *?r = fps-radical r k*
from *kp* **obtain** *h* **where** *k*: *k = Suc h* **by** (*cases k*, *auto*)
have *ra0'*: *r k* (*a$0*) ≠ *0* **using** *a0 ra0 k* **by** *auto*
have *rb0'*: *r k* (*b$0*) ≠ *0* **using** *b0 rb0 k* **by** *auto*

{assume *?rhs*
then have *?r* (*a*/*b*) $ *0* = (*?r a* / *?r b*)$*0* **by** *simp*
then have *?lhs* **using** *k a0 b0 rb0'*
by (*simp add*: *fps-divide-def fps-mult-nth fps-inverse-def divide-inverse*) **}**
moreover
{assume *h*: *?lhs*
from *a0 b0* **have** *ab0*[*simp*]: (*a*/*b*)$*0* = *a$0* / *b$0*
by (*simp add*: *fps-divide-def fps-mult-nth divide-inverse fps-inverse-def*)
have *th0*: *r k* ((*a*/*b*)$*0*) ^ *k* = (*a*/*b*)$*0*
by (*simp add*: *h nonzero-power-divide*[*OF rb0'*] *ra0 rb0 del*: *k*)
from *a0 b0 ra0' rb0' kp h*
have *th1*: *r k* ((*a* / *b*) $ *0*) = (*fps-radical r k a* / *fps-radical r k b*) $ *0*
by (*simp add*: *fps-divide-def fps-mult-nth fps-inverse-def divide-inverse del*: *k*)
from *a0 b0 ra0' rb0' kp* **have** *ab0'*: (*a* / *b*) $ *0* ≠ *0*
by (*simp add*: *fps-divide-def fps-mult-nth fps-inverse-def nonzero-imp-inverse-nonzero*)
note *tha*[*simp*] = *iffD1*[*OF power-radical*[**where** *r*=*r* **and** *k*=*h*], *OF a0 ra0*[*unfolded k*], *unfolded k*[*symmetric*]]
note *thb*[*simp*] = *iffD1*[*OF power-radical*[**where** *r*=*r* **and** *k*=*h*], *OF b0 rb0*[*unfolded k*], *unfolded k*[*symmetric*]]
have *th2*: (*?r a* / *?r b*) ^*k* = *a*/*b*
by (*simp add*: *fps-divide-def power-mult-distrib fps-inverse-power*[*symmetric*])
from *iffD1*[*OF radical-unique*[**where** *r*=*r* **and** *a*=*?r a* / *?r b* **and** *b*=*a*/*b* **and** *k*=*h*], *symmetric*, *unfolded k*[*symmetric*], *OF th0 th1 ab0' th2*] **have** *?rhs* **.}**
ultimately show *?thesis* **by** *blast*
qed

lemma *radical-inverse*:
fixes *a* :: *'a*::*field-char-0 fps*
assumes
k: *k*>*0*
and *ra0*: *r k* (*a* $ *0*) ^ *k* = *a* $ *0*
and *r1*: (*r k 1*) ^*k* = *1*
and *a0*: *a$0* ≠ *0*
shows *r k* (*inverse* (*a* $ *0*)) = *r k 1* / (*r k* (*a* $ *0*)) ⟷ *fps-radical r k* (*inverse a*) = *fps-radical r k 1* / *fps-radical r k a*
using *radical-divide*[**where** *k*=*k* **and** *r*=*r* **and** *a*=*1* **and** *b*=*a*, *OF k*] *ra0 r1 a0*
by (*simp add*: *divide-inverse fps-divide-def*)

39.15 Derivative of composition

lemma *fps-compose-deriv*:
fixes *a*:: (*'a*::*idom*) *fps*

assumes *b0*: *b$0 = 0*
shows *fps-deriv (a oo b) = ((fps-deriv a) oo b) * (fps-deriv b)*
proof−
{fix *n*
have *(fps-deriv (a oo b))$n = setsum (λi. a $ i * (fps-deriv (b^i))$n) {0.. Suc n}*
by (*simp add*: *fps-compose-def field-simps setsum-right-distrib del*: *of-nat-Suc*)
also have *... = setsum (λi. a$i * ((fps-const (of-nat i)) * (fps-deriv b * (b^(i − 1))))$n) {0.. Suc n}*
by (*simp add*: *field-simps fps-deriv-power del*: *fps-mult-left-const-nth of-nat-Suc*)
also have *... = setsum (λi. of-nat i * a$i * (((b^(i − 1)) * fps-deriv b))$n) {0.. Suc n}*
unfolding *fps-mult-left-const-nth* **by** (*simp add*: *field-simps*)
also have *... = setsum (λi. of-nat i * a$i * (setsum (λj. (b^ (i − 1))$j * (fps-deriv b)$(n − j)) {0..n})) {0.. Suc n}*
unfolding *fps-mult-nth* **..**
also have *... = setsum (λi. of-nat i * a$i * (setsum (λj. (b^ (i − 1))$j * (fps-deriv b)$(n − j)) {0..n})) {1.. Suc n}*
apply (*rule setsum-mono-zero-right*)
apply (*auto simp add*: *mult-delta-left setsum-delta not-le*)
done
also have *... = setsum (λi. of-nat (i + 1) * a$(i+1) * (setsum (λj. (b^ i)$j * of-nat (n − j + 1) * b$(n − j + 1)) {0..n})) {0.. n}*
unfolding *fps-deriv-nth*
apply (*rule setsum-reindex-cong*[**where** *f*=*Suc*])
by (*auto simp add*: *mult-assoc*)
finally have *th0*: *(fps-deriv (a oo b))$n = setsum (λi. of-nat (i + 1) * a$(i+1) * (setsum (λj. (b^ i)$j * of-nat (n − j + 1) * b$(n − j + 1)) {0..n})) {0.. n}* **.**

have *(((fps-deriv a) oo b) * (fps-deriv b))$n = setsum (λi. (fps-deriv b)$ (n − i) * ((fps-deriv a) oo b)$i) {0..n}*
unfolding *fps-mult-nth* **by** (*simp add*: *mult-ac*)
also have *... = setsum (λi. setsum (λj. of-nat (n − i +1) * b$(n − i + 1) * of-nat (j + 1) * a$(j+1) * (b^j)$i) {0..n}) {0..n}*
unfolding *fps-deriv-nth fps-compose-nth setsum-right-distrib mult-assoc*
apply (*rule setsum-cong2*)
apply (*rule setsum-mono-zero-left*)
apply (*simp-all add*: *subset-eq*)
apply *clarify*
apply (*subgoal-tac b^i$x = 0*)
apply *simp*
apply (*rule startsby-zero-power-prefix*[*OF b0*, *rule-format*])
by *simp*
also have *... = setsum (λi. of-nat (i + 1) * a$(i+1) * (setsum (λj. (b^ i)$j * of-nat (n − j + 1) * b$(n − j + 1)) {0..n})) {0.. n}*
unfolding *setsum-right-distrib*
apply (*subst setsum-commute*)
by ((*rule setsum-cong2*)+) *simp*
finally have *(fps-deriv (a oo b))$n = (((fps-deriv a) oo b) * (fps-deriv b)) $n*

unfolding *th0* **by** *simp*}
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

lemma *fps-mult-X-plus-1-nth*:
((1+X)∗a) $n = (*if* n = 0 *then* (a$n :: 'a::*comm-ring-1*) *else* a$n + a$(n − 1))
proof−
{assume n = 0 **hence** *?thesis* **by** (*simp add*: *fps-mult-nth*)}
moreover
{fix m **assume** m: n = Suc m
have ((1+X)∗a) $n = *setsum* (λi. (1+X)$i ∗ a$(n−i)) {0..n}
by (*simp add*: *fps-mult-nth*)
also have ... = *setsum* (λi. (1+X)$i ∗ a$(n−i)) {0.. 1}
unfolding m
apply (*rule setsum-mono-zero-right*)
by (*auto simp add*:)
also have ... = (*if* n = 0 *then* (a$n :: 'a::*comm-ring-1*) *else* a$n + a$(n − 1))
unfolding m
by (*simp add*:)
finally have *?thesis* **.}**
ultimately show *?thesis* **by** (*cases* n, *auto*)
qed

39.16 Finite FPS (i.e. polynomials) and X

lemma *fps-poly-sum-X*:
assumes z: ∀ i > n. a$i = (0::'a::*comm-ring-1*)
shows a = *setsum* (λi. *fps-const* (a$i) ∗ X^i) {0..n} (**is** a = *?r*)
proof−
{fix i
have a$i = *?r*$i
unfolding *fps-setsum-nth fps-mult-left-const-nth X-power-nth*
by (*simp add*: *mult-delta-right setsum-delta'* z)
}
then show *?thesis* **unfolding** *fps-eq-iff* **by** *blast*
qed

39.17 Compositional inverses

fun *compinv* :: 'a *fps* ⇒ *nat* ⇒ 'a::{*field*} **where**
compinv a 0 = X$0
| *compinv* a (*Suc* n) = (X$ *Suc* n − *setsum* (λi. (*compinv* a i) ∗ (a^i)$*Suc* n) {0 .. n}) / (a$1) ^ *Suc* n

definition *fps-inv* a = *Abs-fps* (*compinv* a)

lemma *fps-inv*: **assumes** a0: a$0 = 0 **and** a1: a$1 ≠ 0
shows *fps-inv* a *oo* a = X
proof−

```
  let ?i = fps-inv a oo a
  {fix n
    have ?i $n = X$n
    proof(induct n rule: nat-less-induct)
      fix n assume h: ∀ m<n. ?i$m = X$m
      {assume n=0 hence ?i $n = X$n using a0
         by (simp add: fps-compose-nth fps-inv-def)}
      moreover
      {fix n1 assume n1: n = Suc n1
       have ?i $ n = setsum (λi. (fps-inv a $ i) * (a^i)$n) {0 .. n1} + fps-inv a
$ Suc n1 * (a $ 1) ^ Suc n1
         by (simp add: fps-compose-nth n1 startsby-zero-power-nth-same[OF a0]
                del: power-Suc)
       also have ... = setsum (λi. (fps-inv a $ i) * (a^i)$n) {0 .. n1} + (X$ Suc
n1 − setsum (λi. (fps-inv a $ i) * (a^i)$n) {0 .. n1})
         using a0 a1 n1 by (simp add: fps-inv-def)
       also have ... = X$n using n1 by simp
       finally have ?i $ n = X$n .}
      ultimately show ?i $ n = X$n by (cases n, auto)
    qed}
  then show ?thesis by (simp add: fps-eq-iff)
qed


fun gcompinv :: 'a fps ⇒ 'a fps ⇒ nat ⇒ 'a::{field} where
  gcompinv b a 0 = b$0
| gcompinv b a (Suc n) = (b$ Suc n − setsum (λi. (gcompinv b a i) * (a^i)$Suc
n) {0 .. n}) / (a$1) ^ Suc n

definition fps-ginv b a = Abs-fps (gcompinv b a)

lemma fps-ginv: assumes a0: a$0 = 0 and a1: a$1 ≠ 0
  shows fps-ginv b a oo a = b
proof−
  let ?i = fps-ginv b a oo a
  {fix n
    have ?i $n = b$n
    proof(induct n rule: nat-less-induct)
      fix n assume h: ∀ m<n. ?i$m = b$m
      {assume n=0 hence ?i $n = b$n using a0
         by (simp add: fps-compose-nth fps-ginv-def)}
      moreover
      {fix n1 assume n1: n = Suc n1
       have ?i $ n = setsum (λi. (fps-ginv b a $ i) * (a^i)$n) {0 .. n1} + fps-ginv
b a $ Suc n1 * (a $ 1) ^ Suc n1
         by (simp add: fps-compose-nth n1 startsby-zero-power-nth-same[OF a0]
                del: power-Suc)
       also have ... = setsum (λi. (fps-ginv b a $ i) * (a^i)$n) {0 .. n1} + (b$
Suc n1 − setsum (λi. (fps-ginv b a $ i) * (a^i)$n) {0 .. n1})
```

```
      using a0 a1 n1 by (simp add: fps-ginv-def)
    also have ... = b$n using n1 by simp
    finally have ?i $ n = b$n .}
   ultimately show ?i $ n = b$n by (cases n, auto)
  qed}
 then show ?thesis by (simp add: fps-eq-iff)
qed

lemma fps-inv-ginv: fps-inv = fps-ginv X
 apply (auto simp add: expand-fun-eq fps-eq-iff fps-inv-def fps-ginv-def)
 apply (induct-tac n rule: nat-less-induct, auto)
 apply (case-tac na)
 apply simp
 apply simp
 done

lemma fps-compose-1[simp]: 1 oo a = 1
 by (simp add: fps-eq-iff fps-compose-nth mult-delta-left setsum-delta)

lemma fps-compose-0[simp]: 0 oo a = 0
 by (simp add: fps-eq-iff fps-compose-nth)

lemma fps-compose-0-right[simp]: a oo 0 = fps-const (a$0)
 by (auto simp add: fps-eq-iff fps-compose-nth power-0-left setsum-0')

lemma fps-compose-add-distrib: (a + b) oo c = (a oo c) + (b oo c)
 by (simp add: fps-eq-iff fps-compose-nth field-simps setsum-addf)

lemma fps-compose-setsum-distrib: (setsum f S) oo a = setsum (λi. f i oo a) S
proof−
 {assume ¬ finite S hence ?thesis by simp}
 moreover
 {assume fS: finite S
  have ?thesis
  proof(rule finite-induct[OF fS])
   show setsum f {} oo a = (∑ i∈{}. f i oo a) by simp
  next
   fix x F assume fF: finite F and xF: x ∉ F and h: setsum f F oo a = setsum
(λi. f i oo a) F
   show setsum f (insert x F) oo a = setsum (λi. f i oo a) (insert x F)
    using fF xF h by (simp add: fps-compose-add-distrib)
  qed}
 ultimately show ?thesis by blast
qed

lemma convolution-eq:
 setsum (%i. a (i :: nat) * b (n − i)) {0 .. n} = setsum (%(i,j). a i * b j) {(i,j).
i <= n ∧ j ≤ n ∧ i + j = n}
 apply (rule setsum-reindex-cong[where f=fst])
```

apply (*clarsimp simp add*: *inj-on-def*)
apply (*auto simp add*: *expand-set-eq image-iff*)
apply (*rule-tac x*= *x* **in** *exI*)
apply *clarsimp*
apply (*rule-tac x*=*n* − *x* **in** *exI*)
apply *arith*
done

lemma *product-composition-lemma*:
assumes *c0*: *c*$*0* = (*0*::*'a*::*idom*) **and** *d0*: *d*$*0* = *0*
shows ((*a oo c*) ∗ (*b oo d*))$*n* = *setsum* (%(*k*,*m*). *a*$*k* ∗ *b*$*m* ∗ (*c* ^ *k* ∗ *d* ^ *m*) $ *n*) {(*k*,*m*). *k* + *m* ≤ *n*} (**is** *?l* = *?r*)
proof−
let *?S* = {(*k*::*nat*, *m*::*nat*). *k* + *m* ≤ *n*}
have *s*: *?S* ⊆ {*0*..*n*} <∗> {*0*..*n*} **by** (*auto simp add*: *subset-eq*)
have *f*: *finite* {(*k*::*nat*, *m*::*nat*). *k* + *m* ≤ *n*}
apply (*rule finite-subset*[*OF s*])
by *auto*
have *?r* = *setsum* (%*i*. *setsum* (%(*k*,*m*). *a*$*k* ∗ (*c* ^ *k*)$*i* ∗ *b*$*m* ∗ (*d* ^ *m*) $ (*n* − *i*)) {(*k*,*m*). *k* + *m* ≤ *n*}) {*0*..*n*}
apply (*simp add*: *fps-mult-nth setsum-right-distrib*)
apply (*subst setsum-commute*)
apply (*rule setsum-cong2*)
by (*auto simp add*: *field-simps*)
also have ... = *?l*
apply (*simp add*: *fps-mult-nth fps-compose-nth setsum-product*)
apply (*rule setsum-cong2*)
apply (*simp add*: *setsum-cartesian-product mult-assoc*)
apply (*rule setsum-mono-zero-right*[*OF f*])
apply (*simp add*: *subset-eq*) **apply** *presburger*
apply *clarsimp*
apply (*rule ccontr*)
apply (*clarsimp simp add*: *not-le*)
apply (*case-tac x* < *aa*)
apply *simp*
apply (*frule-tac startsby-zero-power-prefix*[*rule-format*, *OF c0*])
apply *blast*
apply *simp*
apply (*frule-tac startsby-zero-power-prefix*[*rule-format*, *OF d0*])
apply *blast*
done
finally show *?thesis* **by** *simp*
qed

lemma *product-composition-lemma′*:
assumes *c0*: *c*$*0* = (*0*::*'a*::*idom*) **and** *d0*: *d*$*0* = *0*
shows ((*a oo c*) ∗ (*b oo d*))$*n* = *setsum* (%*k*. *setsum* (%*m*. *a*$*k* ∗ *b*$*m* ∗ (*c* ^ *k* ∗ *d* ^ *m*) $ *n*) {*0*..*n*}) {*0*..*n*} (**is** *?l* = *?r*)
unfolding *product-composition-lemma*[*OF c0 d0*]

unfolding *setsum-cartesian-product*
apply (*rule setsum-mono-zero-left*)
apply *simp*
apply (*clarsimp simp add*: *subset-eq*)
apply *clarsimp*
apply (*rule ccontr*)
apply (*subgoal-tac* (*c^aa* ∗ *d^ba*) $ *n* = *0*)
apply *simp*
unfolding *fps-mult-nth*
apply (*rule setsum-0'*)
apply (*clarsimp simp add*: *not-le*)
apply (*case-tac aaa* < *aa*)
apply (*rule startsby-zero-power-prefix*[*OF c0*, *rule-format*])
apply *simp*
apply (*subgoal-tac n* − *aaa* < *ba*)
apply (*frule-tac k* = *ba* **in** *startsby-zero-power-prefix*[*OF d0*, *rule-format*])
apply *simp*
apply *arith*
done

lemma *setsum-pair-less-iff*:
setsum (%((*k*::*nat*),*m*). *a k* ∗ *b m* ∗ *c* (*k* + *m*)) {(*k*,*m*). *k* + *m* ≤ *n*} = *setsum* (%*s*. *setsum* (%*i*. *a i* ∗ *b* (*s* − *i*) ∗ *c s*) {*0*..*s*}) {*0*..*n*} (**is** *?l* = *?r*)
proof−
let *?KM*= {(*k*,*m*). *k* + *m* ≤ *n*}
let *?f* = %*s*. *UNION* {(*0*::*nat*)..*s*} (%*i*. {(*i*,*s* − *i*)})
have *th0*: *?KM* = *UNION* {*0*..*n*} *?f*
apply (*simp add*: *expand-set-eq*)
apply *arith*
done
show *?l* = *?r*
unfolding *th0*
apply (*subst setsum-UN-disjoint*)
apply *auto*
apply (*subst setsum-UN-disjoint*)
apply *auto*
done
qed

lemma *fps-compose-mult-distrib-lemma*:
assumes *c0*: *c*$*0* = (*0*::'*a*::*idom*)
shows ((*a oo c*) ∗ (*b oo c*))$*n* = *setsum* (%*s*. *setsum* (%*i*. *a*$*i* ∗ *b*$(*s* − *i*) ∗ (*c^s*) $ *n*) {*0*..*s*}) {*0*..*n*} (**is** *?l* = *?r*)
unfolding *product-composition-lemma*[*OF c0 c0*] *power-add*[*symmetric*]
unfolding *setsum-pair-less-iff*[**where** *a* = %*k*. *a*$*k* **and** *b*=%*m*. *b*$*m* **and** *c*=%*s*. (*c* ^ *s*)$*n* **and** *n* = *n*] **..**

lemma *fps-compose-mult-distrib*:
assumes *c0*: *c$0* = (*0*::*'a*::*idom*)
shows (*a* ∗ *b*) *oo c* = (*a oo c*) ∗ (*b oo c*) (**is** *?l* = *?r*)
apply (*simp add*: *fps-eq-iff fps-compose-mult-distrib-lemma*[*OF c0*])
by (*simp add*: *fps-compose-nth fps-mult-nth setsum-left-distrib*)
lemma *fps-compose-setprod-distrib*:
assumes *c0*: *c$0* = (*0*::*'a*::*idom*)
shows (*setprod a S*) *oo c* = *setprod* (%*k*. *a k oo c*) *S* (**is** *?l* = *?r*)
apply (*cases finite S*)
apply *simp-all*
apply (*induct S rule*: *finite-induct*)
apply *simp*
apply (*simp add*: *fps-compose-mult-distrib*[*OF c0*])
done

lemma *fps-compose-power*: **assumes** *c0*: *c$0* = (*0*::*'a*::*idom*)
shows (*a oo c*) ^*n* = *a* ^*n oo c* (**is** *?l* = *?r*)
proof−
{assume *n*=*0* **then have** *?thesis* **by** *simp***}**
moreover
{fix *m* **assume** *m*: *n* = *Suc m*
have *th0*: *a* ^*n* = *setprod* (%*k*. *a*) {*0*..*m*} (*a oo c*) ^ *n* = *setprod* (%*k*. *a oo c*) {*0*..*m*}
by (*simp-all add*: *setprod-constant m*)
then have *?thesis*
by (*simp add*: *fps-compose-setprod-distrib*[*OF c0*])**}**
ultimately show *?thesis* **by** (*cases n*, *auto*)
qed

lemma *fps-compose-uminus*: − (*a*::*'a*::*ring-1 fps*) *oo c* = − (*a oo c*)
by (*simp add*: *fps-eq-iff fps-compose-nth field-simps setsum-negf*[*symmetric*])

lemma *fps-compose-sub-distrib*:
shows (*a* − *b*) *oo* (*c*::*'a*::*ring-1 fps*) = (*a oo c*) − (*b oo c*)
unfolding *diff-minus fps-compose-uminus fps-compose-add-distrib* **..**

lemma *X-fps-compose*:*X oo a* = *Abs-fps* (λ*n*. *if n* = *0 then* (*0*::*'a*::*comm-ring-1*) *else a$n*)
by (*simp add*: *fps-eq-iff fps-compose-nth mult-delta-left setsum-delta power-Suc*)

lemma *fps-inverse-compose*:
assumes *b0*: (*b$0* :: *'a*::*field*) = *0* **and** *a0*: *a$0* ≠ *0*
shows *inverse a oo b* = *inverse* (*a oo b*)
proof−
let *?ia* = *inverse a*
let *?ab* = *a oo b*
let *?iab* = *inverse ?ab*

from *a0* **have** *ia0*: *?ia* $ *0* ≠ *0* **by** (*simp*)

from *a0* **have** *ab0*: *?ab $ 0 ≠ 0* **by** (*simp add*: *fps-compose-def*)
have (*?ia oo b*) ∗ (*a oo b*) = *1*
unfolding *fps-compose-mult-distrib*[*OF b0*, *symmetric*]
unfolding *inverse-mult-eq-1*[*OF a0*]
fps-compose-1 **..**

then have (*?ia oo b*) ∗ (*a oo b*) ∗ *?iab* = *1* ∗ *?iab* **by** *simp*
then have (*?ia oo b*) ∗ (*?iab* ∗ (*a oo b*)) = *?iab* **by** *simp*
then show *?thesis*
unfolding *inverse-mult-eq-1*[*OF ab0*] **by** *simp*
qed

lemma *fps-divide-compose*:
assumes *c0*: (*c$0* :: *'a*::*field*) = *0* **and** *b0*: *b$0* ≠ *0*
shows (*a*/*b*) *oo c* = (*a oo c*) / (*b oo c*)
unfolding *fps-divide-def fps-compose-mult-distrib*[*OF c0*]
fps-inverse-compose[*OF c0 b0*] **..**

lemma *gp*: **assumes** *a0*: *a$0* = (*0*::*'a*::*field*)
shows (*Abs-fps* (λ*n*. *1*)) *oo a* = *1*/(*1* − *a*) (**is** *?one oo a* = -)
proof−
have *o0*: *?one $ 0 ≠ 0* **by** *simp*
have *th0*: (*1* − *X*) $ *0* ≠ (*0*::*'a*) **by** *simp*
from *fps-inverse-gp*[**where** *?'a* = *'a*]
have *inverse ?one* = *1* − *X* **by** (*simp add*: *fps-eq-iff*)
hence *inverse* (*inverse ?one*) = *inverse* (*1* − *X*) **by** *simp*
hence *th*: *?one* = *1*/(*1* − *X*) **unfolding** *fps-inverse-idempotent*[*OF o0*]
by (*simp add*: *fps-divide-def*)
show *?thesis* **unfolding** *th*
unfolding *fps-divide-compose*[*OF a0 th0*]
fps-compose-1 fps-compose-sub-distrib X-fps-compose-startby0[*OF a0*] **..**
qed

lemma *fps-const-power*[*simp*]: *fps-const* (*c*::*'a*::*ring-1*) ^ *n* = *fps-const* (*c*^*n*)
by (*induct n*, *auto*)

lemma *fps-compose-radical*:
assumes *b0*: *b$0* = (*0*::*'a*::*field-char-0*)
and *ra0*: *r* (*Suc k*) (*a$0*) ^ *Suc k* = *a$0*
and *a0*: *a$0* ≠ *0*
shows *fps-radical r* (*Suc k*) *a oo b* = *fps-radical r* (*Suc k*) (*a oo b*)
proof−
let *?r* = *fps-radical r* (*Suc k*)
let *?ab* = *a oo b*
have *ab0*: *?ab $ 0* = *a$0* **by** (*simp add*: *fps-compose-def*)
from *ab0 a0 ra0* **have** *rab0*: *?ab $ 0 ≠ 0 r* (*Suc k*) (*?ab $ 0*) ^ *Suc k* = *?ab $ 0* **by** *simp-all*
have *th00*: *r* (*Suc k*) ((*a oo b*) $ *0*) = (*fps-radical r* (*Suc k*) *a oo b*) $ *0*
by (*simp add*: *ab0 fps-compose-def*)

have *th0*: (*?r a oo b*) ^ (*Suc k*) = *a oo b*
unfolding *fps-compose-power*[*OF b0*]
unfolding *iffD1*[*OF power-radical*[*of a r k*], *OF a0 ra0*] **..**
from *iffD1*[*OF radical-unique*[**where** *r*=*r* **and** *k*=*k* **and** *b*= *?ab* **and** *a* = *?r a oo b*, *OF rab0*(*2*) *th00 rab0*(*1*)], *OF th0*] **show** *?thesis* **.**
qed

lemma *fps-const-mult-apply-left*:
fps-const c ∗ (*a oo b*) = (*fps-const c* ∗ *a*) *oo b*
by (*simp add*: *fps-eq-iff fps-compose-nth setsum-right-distrib mult-assoc*)

lemma *fps-const-mult-apply-right*:
(*a oo b*) ∗ *fps-const* (*c*::'*a*::*comm-semiring-1*) = (*fps-const c* ∗ *a*) *oo b*
by (*auto simp add*: *fps-const-mult-apply-left mult-commute*)

lemma *fps-compose-assoc*:
assumes *c0*: *c*$*0* = (*0*::'*a*::*idom*) **and** *b0*: *b*$*0* = *0*
shows *a oo* (*b oo c*) = *a oo b oo c* (**is** *?l* = *?r*)
proof−
{fix *n*
have *?l*$*n* = (*setsum* (λ*i*. (*fps-const* (*a*$*i*) ∗ *b* ^ *i*) *oo c*) {*0*..*n*})$*n*
by (*simp add*: *fps-compose-nth fps-compose-power*[*OF c0*] *fps-const-mult-apply-left setsum-right-distrib mult-assoc fps-setsum-nth*)
also have ... = ((*setsum* (λ*i*. *fps-const* (*a*$*i*) ∗ *b* ^ *i*) {*0*..*n*}) *oo c*)$*n*
by (*simp add*: *fps-compose-setsum-distrib*)
also have ... = *?r*$*n*
apply (*simp add*: *fps-compose-nth fps-setsum-nth setsum-left-distrib mult-assoc*)
apply (*rule setsum-cong2*)
apply (*rule setsum-mono-zero-right*)
apply (*auto simp add*: *not-le*)
by (*erule startsby-zero-power-prefix*[*OF b0*, *rule-format*])
finally have *?l*$*n* = *?r*$*n* **.}**
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

lemma *fps-X-power-compose*:
assumes *a0*: *a*$*0*=*0* **shows** *X* ^ *k oo a* = (*a*::('*a*::*idom fps*)) ^ *k* (**is** *?l* = *?r*)
proof−
{assume *k*=*0* **hence** *?thesis* **by** *simp***}**
moreover
{fix *h* **assume** *h*: *k* = *Suc h*
{fix *n*
{assume *kn*: *k*>*n* **hence** *?l* $ *n* = *?r* $*n* **using** *a0 startsby-zero-power-prefix*[*OF a0*] *h*
by (*simp add*: *fps-compose-nth del*: *power-Suc*)**}**
moreover
{assume *kn*: *k* ≤ *n*
hence *?l*$*n* = *?r*$*n*

```
      by (simp add: fps-compose-nth mult-delta-left setsum-delta)}
    moreover have k >n ∨ k≤ n  by arith
    ultimately have ?l$n = ?r$n  by blast}
  then have ?thesis unfolding fps-eq-iff  by blast}
 ultimately show ?thesis by (cases k, auto)
qed

lemma fps-inv-right: assumes a0: a$0 = 0 and a1: a$1 ≠ 0
 shows a oo fps-inv a = X
proof−
 let ?ia = fps-inv a
 let ?iaa = a oo fps-inv a
 have th0: ?ia $ 0 = 0 by (simp add: fps-inv-def)
 have th1: ?iaa $ 0 = 0 using a0 a1
   by (simp add: fps-inv-def fps-compose-nth)
 have th2: X$0 = 0 by simp
 from fps-inv[OF a0 a1] have a oo (fps-inv a oo a) = a oo X by simp
 then have (a oo fps-inv a) oo a = X oo a
   by (simp add: fps-compose-assoc[OF a0 th0] X-fps-compose-startby0[OF a0])
 with fps-compose-inj-right[OF a0 a1]
 show ?thesis by simp
qed

lemma fps-inv-deriv:
 assumes a0:a$0 = (0::'a::{field}) and a1: a$1 ≠ 0
 shows fps-deriv (fps-inv a) = inverse (fps-deriv a oo fps-inv a)
proof−
 let ?ia = fps-inv a
 let ?d = fps-deriv a oo ?ia
 let ?dia = fps-deriv ?ia
 have ia0: ?ia$0 = 0 by (simp add: fps-inv-def)
 have th0: ?d$0 ≠ 0 using a1 by (simp add: fps-compose-nth fps-deriv-nth)
 from fps-inv-right[OF a0 a1] have ?d * ?dia = 1
   by (simp add: fps-compose-deriv[OF ia0, of a, symmetric] )
 hence inverse ?d * ?d * ?dia = inverse ?d * 1 by simp
 with inverse-mult-eq-1[OF th0]
 show ?dia = inverse ?d by simp
qed

lemma fps-inv-idempotent:
 assumes a0: a$0 = 0 and a1: a$1 ≠ 0
 shows fps-inv (fps-inv a) = a
proof−
 let ?r = fps-inv
 have ra0: ?r a $ 0 = 0 by (simp add: fps-inv-def)
 from a1 have ra1: ?r a $ 1 ≠ 0 by (simp add: fps-inv-def field-simps)
 have X0: X$0 = 0 by simp
 from fps-inv[OF ra0 ra1] have ?r (?r a) oo ?r a = X .
 then have ?r (?r a) oo ?r a oo a = X oo a by simp
```

```
  then have ?r (?r a) oo (?r a oo a) = a
    unfolding X-fps-compose-startby0[OF a0]
    unfolding fps-compose-assoc[OF a0 ra0, symmetric] .
  then show ?thesis unfolding fps-inv[OF a0 a1] by simp
qed

lemma fps-ginv-ginv:
  assumes a0: a$0 = 0 and a1: a$1 ≠ 0
  and c0: c$0 = 0 and  c1: c$1 ≠ 0
  shows fps-ginv b (fps-ginv c a) = b oo a oo fps-inv c
proof−
  let ?r = fps-ginv
  from c0 have rca0: ?r c a $0 = 0 by (simp add: fps-ginv-def)
  from a1 c1 have rca1: ?r c a $ 1 ≠ 0 by (simp add: fps-ginv-def field-simps)
  from fps-ginv[OF rca0 rca1]
  have ?r b (?r c a) oo ?r c a = b .
  then have ?r b (?r c a) oo ?r c a oo a = b oo a by simp
  then have ?r b (?r c a) oo (?r c a oo a) = b oo a
    apply (subst fps-compose-assoc)
    using a0 c0 by (auto simp add: fps-ginv-def)
  then have ?r b (?r c a) oo c = b oo a
    unfolding fps-ginv[OF a0 a1] .
  then have ?r b (?r c a) oo c oo fps-inv c= b oo a oo fps-inv c by simp
  then have ?r b (?r c a) oo (c oo fps-inv c) = b oo a oo fps-inv c
    apply (subst fps-compose-assoc)
    using a0 c0 by (auto simp add: fps-inv-def)
  then show ?thesis unfolding fps-inv-right[OF c0 c1] by simp
qed

lemma fps-ginv-deriv:
  assumes a0:a$0 = (0::'a::{field}) and a1: a$1 ≠ 0
  shows fps-deriv (fps-ginv b a) = (fps-deriv b / fps-deriv a) oo fps-ginv X a
proof−
  let ?ia = fps-ginv b a
  let ?iXa = fps-ginv X a
  let ?d = fps-deriv
  let ?dia = ?d ?ia
  have iXa0: ?iXa $ 0 = 0 by (simp add: fps-ginv-def)
  have da0: ?d a $ 0 ≠ 0 using a1 by simp
  from fps-ginv[OF a0 a1, of b] have ?d (?ia oo a) = fps-deriv b by simp
  then have (?d ?ia oo a) * ?d a = ?d b unfolding fps-compose-deriv[OF a0] .
  then have (?d ?ia oo a) * ?d a * inverse (?d a) = ?d b * inverse (?d a) by
simp
  then have (?d ?ia oo a) * (inverse (?d a) * ?d a) = ?d b / ?d a
    by (simp add: fps-divide-def)
  then have (?d ?ia oo a) oo ?iXa =  (?d b / ?d a) oo ?iXa
    unfolding inverse-mult-eq-1[OF da0] by simp
  then have ?d ?ia oo (a oo ?iXa) =  (?d b / ?d a) oo ?iXa
    unfolding fps-compose-assoc[OF iXa0 a0] .
```

then show *?thesis* **unfolding** *fps-inv-ginv*[*symmetric*]
unfolding *fps-inv-right*[*OF a0 a1*] **by** *simp*
qed

39.18 Elementary series

39.18.1 Exponential series

definition *E x* = *Abs-fps* (λ*n*. *x*^*n* / *of-nat* (*fact n*))

lemma *E-deriv*[*simp*]: *fps-deriv* (*E a*) = *fps-const* (*a*::'*a*::*field-char-0*) * *E a* (**is** *?l* = *?r*)
proof−
{fix *n*
have *?l*$*n* = *?r* $ *n*
apply (*auto simp add*: *E-def field-simps power-Suc*[*symmetric*]*simp del*: *fact-Suc of-nat-Suc power-Suc*)
by (*simp add*: *of-nat-mult field-simps*)**}**
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

lemma *E-unique-ODE*:
fps-deriv a = *fps-const c* * *a* ⟷ *a* = *fps-const* (*a*$*0*) * *E* (*c* :: '*a*::*field-char-0*)
(**is** *?lhs* ⟷ *?rhs*)
proof−
{assume *d*: *?lhs*
from *d* **have** *th*: ⋀*n*. *a* $ *Suc n* = *c* * *a*$*n* / *of-nat* (*Suc n*)
by (*simp add*: *fps-deriv-def fps-eq-iff field-simps del*: *of-nat-Suc*)
{fix *n* **have** *a*$*n* = *a*$*0* * *c* ^ *n*/ (*of-nat* (*fact n*))
apply (*induct n*)
apply *simp*
unfolding *th*
using *fact-gt-zero-nat*
apply (*simp add*: *field-simps del*: *of-nat-Suc fact-Suc*)
apply (*drule sym*)
by (*simp add*: *field-simps of-nat-mult power-Suc*)**}**
note *th*' = *this*
have *?rhs*
by (*auto simp add*: *fps-eq-iff fps-const-mult-left E-def intro* : *th*')**}**
moreover
{assume *h*: *?rhs*
have *?lhs*
apply (*subst h*)
apply *simp*
apply (*simp only*: *h*[*symmetric*])
by *simp***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *E-add-mult*: *E* (*a* + *b*) = *E* (*a*::'*a*::*field-char-0*) * *E b* (**is** *?l* = *?r*)

proof−
have *fps-deriv* (*?r*) = *fps-const* (*a*+*b*) ∗ *?r*
by (*simp add*: *fps-const-add*[*symmetric*] *field-simps del*: *fps-const-add*)
then have *?r* = *?l* **apply** (*simp only*: *E-unique-ODE*)
by (*simp add*: *fps-mult-nth E-def*)
then show *?thesis* **..**
qed

lemma *E-nth*[*simp*]: *E a* $ *n* = *a*^*n* / *of-nat* (*fact n*)
by (*simp add*: *E-def*)

lemma *E0*[*simp*]: *E* (*0*::*'a*::{*field*}) = *1*
by (*simp add*: *fps-eq-iff power-0-left*)

lemma *E-neg*: *E* (− *a*) = *inverse* (*E* (*a*::*'a*::*field-char-0*))
proof−
from *E-add-mult*[*of a* − *a*] **have** *th0*: *E a* ∗ *E* (− *a*) = *1*
by (*simp*)
have *th1*: *E a* $ *0* ≠ *0* **by** *simp*
from *fps-inverse-unique*[*OF th1 th0*] **show** *?thesis* **by** *simp*
qed

lemma *E-nth-deriv*[*simp*]: *fps-nth-deriv n* (*E* (*a*::*'a*::*field-char-0*)) = (*fps-const a*)^*n* ∗ (*E a*)
by (*induct n*, *auto simp add*: *power-Suc*)

lemma *X-compose-E*[*simp*]: *X oo E* (*a*::*'a*::{*field*}) = *E a* − *1*
by (*simp add*: *fps-eq-iff X-fps-compose*)

lemma *LE-compose*:
assumes *a*: *a*≠*0*
shows *fps-inv* (*E a* − *1*) *oo* (*E a* − *1*) = *X*
and (*E a* − *1*) *oo fps-inv* (*E a* − *1*) = *X*
proof−
let *?b* = *E a* − *1*
have *b0*: *?b* $ *0* = *0* **by** *simp*
have *b1*: *?b* $ *1* ≠ *0* **by** (*simp add*: *a*)
from *fps-inv*[*OF b0 b1*] **show** *fps-inv* (*E a* − *1*) *oo* (*E a* − *1*) = *X* **.**
from *fps-inv-right*[*OF b0 b1*] **show** (*E a* − *1*) *oo fps-inv* (*E a* − *1*) = *X* **.**
qed

lemma *fps-const-inverse*:
a ≠ *0* ⟹ *inverse* (*fps-const* (*a*::*'a*::*field*)) = *fps-const* (*inverse a*)
apply (*auto simp add*: *fps-eq-iff fps-inverse-def*) **by** (*case-tac n*, *auto*)

lemma *inverse-one-plus-X*:
inverse (*1* + *X*) = *Abs-fps* (λ*n*. (− *1* ::*'a*::{*field*})^*n*)
(**is** *inverse ?l* = *?r*)

proof−
have *th*: *?l ∗ ?r = 1*
by (*auto simp add*: *field-simps fps-eq-iff minus-one-power-iff*)
have *th'*: *?l $ 0 ≠ 0* **by** (*simp add*:)
from *fps-inverse-unique*[*OF th' th*] **show** *?thesis* **.**
qed

lemma *E-power-mult*: (*E* (*c*::*'a*::*field-char-0*)) ^ *n* = *E* (*of-nat n ∗ c*)
by (*induct n*, *auto simp add*: *field-simps E-add-mult power-Suc*)

lemma *radical-E*:
assumes *r*: *r* (*Suc k*) *1* = *1*
shows *fps-radical r* (*Suc k*) (*E* (*c*::*'a*::{*field-char-0*})) = *E* (*c* / *of-nat* (*Suc k*))
proof−
let *?ck* = (*c* / *of-nat* (*Suc k*))
let *?r* = *fps-radical r* (*Suc k*)
have *eq0*[*simp*]: *?ck ∗ of-nat* (*Suc k*) = *c of-nat* (*Suc k*) ∗ *?ck* = *c*
by (*simp-all del*: *of-nat-Suc*)
have *th0*: *E ?ck* ^ (*Suc k*) = *E c* **unfolding** *E-power-mult eq0* **..**
have *th*: *r* (*Suc k*) (*E c $0*) ^ *Suc k* = *E c $ 0*
r (*Suc k*) (*E c $ 0*) = *E ?ck $ 0 E c $ 0* ≠ *0* **using** *r* **by** *simp-all*
from *th0 radical-unique*[**where** *r*=*r* **and** *k*=*k*, *OF th*]
show *?thesis* **by** *auto*
qed

lemma *Ec-E1-eq*:
E (*1*::*'a*::{*field-char-0*}) *oo* (*fps-const c ∗ X*) = *E c*
apply (*auto simp add*: *fps-eq-iff E-def fps-compose-def power-mult-distrib*)
by (*simp add*: *cond-value-iff cond-application-beta setsum-delta' cong del*: *if-weak-cong*)

The generalized binomial theorem as a consequence of *E* (*?a* + *?b*) = *E ?a ∗ E ?b*

lemma *gbinomial-theorem*:
((*a*::*'a*::{*field-char-0*, *field-inverse-zero*})+*b*) ^ *n* = (∑ *k*=*0*..*n*. *of-nat* (*n choose k*) ∗ *a*^*k* ∗ *b*^(*n*−*k*))
proof−
from *E-add-mult*[*of a b*]
have (*E* (*a* + *b*)) $ *n* = (*E a ∗ E b*)$*n* **by** *simp*
then have (*a* + *b*) ^ *n* = (∑ *i*::*nat* = *0*::*nat*..*n*. *a* ^ *i* ∗ *b* ^ (*n* − *i*) ∗ (*of-nat* (*fact n*) / *of-nat* (*fact i ∗ fact* (*n* − *i*))))
by (*simp add*: *field-simps fps-mult-nth of-nat-mult*[*symmetric*] *setsum-right-distrib*)
then show *?thesis*
apply *simp*
apply (*rule setsum-cong2*)
apply *simp*
apply (*frule binomial-fact*[**where** *?'a* = *'a*, *symmetric*])
by (*simp add*: *field-simps of-nat-mult*)
qed

And the nat-form – also available from Binomial.thy

lemma *binomial-theorem*: $(a+b)\ \hat{}\ n = (\sum k{=}0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k))$
using *gbinomial-theorem*[*of of-nat a of-nat b n*]
unfolding *of-nat-add*[*symmetric*] *of-nat-power*[*symmetric*] *of-nat-mult*[*symmetric*] *of-nat-setsum*[*symmetric*]
by *simp*

39.18.2 Logarithmic series

lemma *Abs-fps-if-0*:
Abs-fps(%*n*. *if n=0 then* (*v*::'*a*::*ring-1*) *else f n*) = *fps-const v* + *X* ∗ *Abs-fps* (%*n*. *f* (*Suc n*))
by (*auto simp add*: *fps-eq-iff*)

definition *L*:: '*a*::{*field-char-0*} ⇒ '*a fps* **where**
L c ≡ *fps-const* (*1*/*c*) ∗ *Abs-fps* (λ*n*. *if n* = *0 then 0 else* (− *1*) ^ (*n* − *1*) / *of-nat n*)

lemma *fps-deriv-L*: *fps-deriv* (*L c*) = *fps-const* (*1*/*c*) ∗ *inverse* (*1* + *X*)
unfolding *inverse-one-plus-X*
by (*simp add*: *L-def fps-eq-iff del*: *of-nat-Suc*)

lemma *L-nth*: *L c* $ *n* = (*if n=0 then 0 else 1*/*c* ∗ ((− *1*) ^ (*n* − *1*) / *of-nat n*))
by (*simp add*: *L-def field-simps*)

lemma *L-0*[*simp*]: *L c* $ *0* = *0* **by** (*simp add*: *L-def*)
lemma *L-E-inv*:
assumes *a*: *a*≠ (*0*::'*a*::{*field-char-0*})
shows *L a* = *fps-inv* (*E a* − *1*) (**is** *?l* = *?r*)
proof−
let *?b* = *E a* − *1*
have *b0*: *?b* $ *0* = *0* **by** *simp*
have *b1*: *?b* $ *1* ≠ *0* **by** (*simp add*: *a*)
have *fps-deriv* (*E a* − *1*) *oo fps-inv* (*E a* − *1*) = (*fps-const a* ∗ (*E a* − *1*) + *fps-const a*) *oo fps-inv* (*E a* − *1*)
by (*simp add*: *field-simps*)
also have ... = *fps-const a* ∗ (*X* + *1*) **apply** (*simp add*: *fps-compose-add-distrib fps-const-mult-apply-left*[*symmetric*] *fps-inv-right*[*OF b0 b1*])
by (*simp add*: *field-simps*)
finally have *eq*: *fps-deriv* (*E a* − *1*) *oo fps-inv* (*E a* − *1*) = *fps-const a* ∗ (*X* + *1*) .
from *fps-inv-deriv*[*OF b0 b1*, *unfolded eq*]
have *fps-deriv* (*fps-inv ?b*) = *fps-const* (*inverse a*) / (*X* + *1*)
using *a*
by (*simp add*: *fps-const-inverse eq fps-divide-def fps-inverse-mult*)
hence *fps-deriv ?l* = *fps-deriv ?r*
by (*simp add*: *fps-deriv-L add-commute fps-divide-def divide-inverse*)
then show *?thesis* **unfolding** *fps-deriv-eq-iff*
by (*simp add*: *L-nth fps-inv-def*)
qed

lemma *L-mult-add*:
assumes *c0*: *c≠0* **and** *d0*: *d≠0*
shows *L c + L d = fps-const (c+d) * L (c*d)*
(**is** *?r = ?l*)
proof−
from *c0 d0* **have** *eq*: *1/c + 1/d = (c+d)/(c*d)* **by** (*simp add*: *field-simps*)
have *fps-deriv ?r = fps-const (1/c + 1/d) * inverse (1 + X)*
by (*simp add*: *fps-deriv-L fps-const-add*[*symmetric*] *algebra-simps del*: *fps-const-add*)
also have *... = fps-deriv ?l*
apply (*simp add*: *fps-deriv-L*)
by (*simp add*: *fps-eq-iff eq*)
finally show *?thesis*
unfolding *fps-deriv-eq-iff* **by** *simp*
qed

39.18.3 Binomial series

definition *fps-binomial a = Abs-fps (λn. a gchoose n)*

lemma *fps-binomial-nth*[*simp*]: *fps-binomial a $ n = a gchoose n*
by (*simp add*: *fps-binomial-def*)

lemma *fps-binomial-ODE-unique*:
fixes *c* :: *'a*::*field-char-0*
shows *fps-deriv a = (fps-const c * a) / (1 + X) ⟷ a = fps-const (a$0) * fps-binomial c*
(**is** *?lhs ⟷ ?rhs*)
proof−
let *?da = fps-deriv a*
let *?x1 = (1 + X)*:: *'a fps*
let *?l = ?x1 * ?da*
let *?r = fps-const c * a*
have *x10*: *?x1 $ 0 ≠ 0* **by** *simp*
have *?l = ?r ⟷ inverse ?x1 * ?l = inverse ?x1 * ?r* **by** *simp*
also have *... ⟷ ?da = (fps-const c * a) / ?x1*
apply (*simp only*: *fps-divide-def mult-assoc*[*symmetric*] *inverse-mult-eq-1*[*OF x10*])
by (*simp add*: *field-simps*)
finally have *eq*: *?l = ?r ⟷ ?lhs* **by** *simp*
moreover
{assume *h*: *?l = ?r*
{fix *n*
from *h* **have** *lrn*: *?l $ n = ?r$n* **by** *simp*

from *lrn*
have *a$ Suc n = ((c − of-nat n) / of-nat (Suc n)) * a $n*
apply (*simp add*: *field-simps del*: *of-nat-Suc*)
by (*cases n*, *simp-all add*: *field-simps del*: *of-nat-Suc*)

```
    }
    note th0 = this
    {fix n have a$n = (c gchoose n) * a$0
     proof(induct n)
       case 0 thus ?case by simp
     next
       case (Suc m)
       thus ?case unfolding th0
         apply (simp add: field-simps del: of-nat-Suc)
         unfolding mult-assoc[symmetric] gbinomial-mult-1
         by (simp add: field-simps)
     qed}
    note th1 = this
    have ?rhs
     apply (simp add: fps-eq-iff)
     apply (subst th1)
     by (simp add: field-simps)}
  moreover
  {assume h: ?rhs
  have th00:⋀x y. x * (a$0 * y) = a$0 * (x*y) by (simp add: mult-commute)
    have ?l = ?r
     apply (subst h)
     apply (subst (2) h)
     apply (clarsimp simp add: fps-eq-iff field-simps)
     unfolding mult-assoc[symmetric] th00 gbinomial-mult-1
     by (simp add: field-simps gbinomial-mult-1)}
  ultimately show ?thesis by blast
qed

lemma fps-binomial-deriv: fps-deriv (fps-binomial c) = fps-const c * fps-binomial
c / (1 + X)
proof−
  let ?a = fps-binomial c
  have th0: ?a = fps-const (?a$0) * ?a by (simp)
  from iffD2[OF fps-binomial-ODE-unique, OF th0] show ?thesis .
qed

lemma fps-binomial-add-mult: fps-binomial (c+d) = fps-binomial c * fps-binomial
d (is ?l = ?r)
proof−
  let ?P = ?r − ?l
  let ?b = fps-binomial
  let ?db = λx. fps-deriv (?b x)
  have fps-deriv ?P = ?db c * ?b d + ?b c * ?db d − ?db (c + d) by simp
  also have ... = inverse (1 + X) * (fps-const c * ?b c * ?b d + fps-const d * ?b
c * ?b d − fps-const (c+d) * ?b (c + d))
    unfolding fps-binomial-deriv
    by (simp add: fps-divide-def field-simps)
  also have ... = (fps-const (c + d)/ (1 + X)) * ?P
```

by (*simp add*: *field-simps fps-divide-def fps-const-add*[*symmetric*] *del*: *fps-const-add*)
finally have *th0*: *fps-deriv ?P* = *fps-const* (*c*+*d*) ∗ *?P* / (1 + *X*)
by (*simp add*: *fps-divide-def*)
have *?P* = *fps-const* (*?P*$0) ∗ *?b* (*c* + *d*)
unfolding *fps-binomial-ODE-unique*[*symmetric*]
using *th0* **by** *simp*
hence *?P* = *0* **by** (*simp add*: *fps-mult-nth*)
then show *?thesis* **by** *simp*
qed

lemma *fps-minomial-minus-one*: *fps-binomial* (− *1*) = *inverse* (*1* + *X*)
(**is** *?l* = *inverse ?r*)
proof−
have *th*: *?r*$*0* ≠ *0* **by** *simp*
have *th*′: *fps-deriv* (*inverse ?r*) = *fps-const* (− *1*) ∗ *inverse ?r* / (*1* + *X*)
by (*simp add*: *fps-inverse-deriv*[*OF th*] *fps-divide-def power2-eq-square mult-commute fps-const-neg*[*symmetric*] *del*: *fps-const-neg*)
have *eq*: *inverse ?r* $ *0* = *1*
by (*simp add*: *fps-inverse-def*)
from *iffD1*[*OF fps-binomial-ODE-unique*[*of inverse* (*1* + *X*) − *1*] *th*′] *eq*
show *?thesis* **by** (*simp add*: *fps-inverse-def*)
qed

Vandermonde's Identity as a consequence

lemma *gbinomial-Vandermonde*: *setsum* (λ*k*. (*a gchoose k*) ∗ (*b gchoose* (*n* − *k*)))
{*0*..*n*} = (*a* + *b*) *gchoose n*
proof−
let *?ba* = *fps-binomial a*
let *?bb* = *fps-binomial b*
let *?bab* = *fps-binomial* (*a* + *b*)
from *fps-binomial-add-mult*[*of a b*] **have** *?bab* $ *n* = (*?ba* ∗ *?bb*)$*n* **by** *simp*
then show *?thesis* **by** (*simp add*: *fps-mult-nth*)
qed

lemma *binomial-Vandermonde*: *setsum* (λ*k*. (*a choose k*) ∗ (*b choose* (*n* − *k*)))
{*0*..*n*} = (*a* + *b*) *choose n*
using *gbinomial-Vandermonde*[*of* (*of-nat a*) *of-nat b n*]

apply (*simp only*: *binomial-gbinomial*[*symmetric*] *of-nat-mult*[*symmetric*] *of-nat-setsum*[*symmetric*] *of-nat-add*[*symmetric*])
by *simp*

lemma *binomial-Vandermonde-same*: *setsum* (λ*k*. (*n choose k*) ^*2*) {*0*..*n*} = (*2*∗*n*) *choose n*
using *binomial-Vandermonde*[*of n n n*,*symmetric*]
unfolding *nat-mult-2* **apply** (*simp add*: *power2-eq-square*)
apply (*rule setsum-cong2*)
by (*auto intro*: *binomial-symmetric*)

lemma *Vandermonde-pochhammer-lemma*:
fixes *a* :: *'a*::*field-char-0*
assumes *b*: ∀ *j*∈{*0* ..<*n*}. *b* ≠ *of-nat j*
shows *setsum* (%*k*. (*pochhammer* (− *a*) *k* ∗ *pochhammer* (− (*of-nat n*)) *k*) / (*of-nat* (*fact k*) ∗ *pochhammer* (*b* − *of-nat n* + *1*) *k*)) {*0*..*n*} = *pochhammer* (− (*a*+ *b*)) *n* / *pochhammer* (− *b*) *n* (**is** *?l* = *?r*)
proof−
let *?m1* = %*m*. (− *1* :: *'a*) ^ *m*
let *?f* = %*m*. *of-nat* (*fact m*)
let *?p* = %(*x*::*'a*). *pochhammer* (− *x*)
from *b* **have** *bn0*: *?p b n* ≠ *0* **unfolding** *pochhammer-eq-0-iff* **by** *simp*
{fix *k* **assume** *kn*: *k* ∈ {*0*..*n*}
{assume *c*:*pochhammer* (*b* − *of-nat n* + *1*) *n* = *0*
then obtain *j* **where** *j*: *j* < *n b* − *of-nat n* + *1* = − *of-nat j*
unfolding *pochhammer-eq-0-iff* **by** *blast*
from *j* **have** *b* = *of-nat n* − *of-nat j* − *of-nat 1*
by (*simp add*: *algebra-simps*)
then have *b* = *of-nat* (*n* − *j* − *1*)
using *j kn* **by** (*simp add*: *of-nat-diff*)
with *b* **have** *False* **using** *j* **by** *auto*}
then have *nz*: *pochhammer* (*1* + *b* − *of-nat n*) *n* ≠ *0*
by (*auto simp add*: *algebra-simps*)

from *nz kn* [*simplified*] **have** *nz'*: *pochhammer* (*1* + *b* − *of-nat n*) *k* ≠ *0*
by (*rule pochhammer-neq-0-mono*)
{assume *k0*: *k* = *0* ∨ *n* =*0*
then have *b gchoose* (*n* − *k*) = (*?m1 n* ∗ *?p b n* ∗ *?m1 k* ∗ *?p* (*of-nat n*) *k*) / (*?f n* ∗ *pochhammer* (*b* − *of-nat n* + *1*) *k*)
using *kn*
by (*cases k=0*, *simp-all add*: *gbinomial-pochhammer*)}
moreover
{assume *n0*: *n* ≠ *0* **and** *k0*: *k* ≠ *0*
then obtain *m* **where** *m*: *n* = *Suc m* **by** (*cases n*, *auto*)
from *k0* **obtain** *h* **where** *h*: *k* = *Suc h* **by** (*cases k*, *auto*)
{assume *kn*: *k* = *n*
then have *b gchoose* (*n* − *k*) = (*?m1 n* ∗ *?p b n* ∗ *?m1 k* ∗ *?p* (*of-nat n*) *k*) / (*?f n* ∗ *pochhammer* (*b* − *of-nat n* + *1*) *k*)
using *kn pochhammer-minus'*[**where** *k*=*k* **and** *n*=*n* **and** *b*=*b*]
apply (*simp add*: *pochhammer-same*)
using *bn0*
by (*simp add*: *field-simps power-add*[*symmetric*])}
moreover
{assume *nk*: *k* ≠ *n*
have *m1nk*: *?m1 n* = *setprod* (%*i*. − *1*) {*0*..*m*}
?m1 k = *setprod* (%*i*. − *1*) {*0*..*h*}
by (*simp-all add*: *setprod-constant m h*)
from *kn nk* **have** *kn'*: *k* < *n* **by** *simp*
have *bnz0*: *pochhammer* (*b* − *of-nat n* + *1*) *k* ≠ *0*
using *bn0 kn*

unfolding *pochhammer-eq-0-iff*
apply *auto*
apply (*erule-tac x= n − ka − 1* **in** *allE*)
by (*auto simp add*: *algebra-simps of-nat-diff*)
have *eq1*: *setprod* (%*k*. (*1*::'*a*) + *of-nat m* − *of-nat k*) {*0* .. *h*} = *setprod of-nat* {*Suc* (*m* − *h*) .. *Suc m*}
apply (*rule strong-setprod-reindex-cong*[**where** *f*=%*k*. *Suc m* − *k*])
using *kn' h m*
apply (*auto simp add*: *inj-on-def image-def*)
apply (*rule-tac x=Suc m* − *x* **in** *bexI*)
apply (*simp-all add*: *of-nat-diff*)
done

have *th1*: (*?m1 k* ∗ *?p* (*of-nat n*) *k*) / *?f n* = *1* / *of-nat*(*fact* (*n* − *k*))
unfolding *m1nk*

unfolding *m h pochhammer-Suc-setprod*
apply (*simp add*: *field-simps del*: *fact-Suc id-def*)
unfolding *fact-altdef-nat id-def*
unfolding *of-nat-setprod*
unfolding *setprod-timesf*[*symmetric*]
apply *auto*
unfolding *eq1*
apply (*subst setprod-Un-disjoint*[*symmetric*])
apply (*auto*)
apply (*rule setprod-cong*)
apply *auto*
done
have *th20*: *?m1 n* ∗ *?p b n* = *setprod* (%*i*. *b* − *of-nat i*) {*0..m*}
unfolding *m1nk*
unfolding *m h pochhammer-Suc-setprod*
unfolding *setprod-timesf*[*symmetric*]
apply (*rule setprod-cong*)
apply *auto*
done
have *th21*:*pochhammer* (*b* − *of-nat n* + *1*) *k* = *setprod* (%*i*. *b* − *of-nat i*) {*n* − *k* .. *n* − *1*}
unfolding *h m*
unfolding *pochhammer-Suc-setprod*
apply (*rule strong-setprod-reindex-cong*[**where** *f*=%*k*. *n* − *1* − *k*])
using *kn*
apply (*auto simp add*: *inj-on-def m h image-def*)
apply (*rule-tac x= m* − *x* **in** *bexI*)
by (*auto simp add*: *of-nat-diff*)

have *?m1 n* ∗ *?p b n* = *pochhammer* (*b* − *of-nat n* + *1*) *k* ∗ *setprod* (%*i*. *b* − *of-nat i*) {*0*.. *n* − *k* − *1*}
unfolding *th20 th21*
unfolding *h m*

apply (*subst setprod-Un-disjoint*[*symmetric*])
using *kn' h m*
apply *auto*
apply (*rule setprod-cong*)
apply *auto*
done
then have *th2*: (*?m1 n * ?p b n*)/*pochhammer* (*b − of-nat n + 1*) *k* = *setprod* (%*i*. *b − of-nat i*) {*0.. n − k − 1*}
using *nz'* **by** (*simp add*: *field-simps*)
have (*?m1 n * ?p b n * ?m1 k * ?p* (*of-nat n*) *k*) / (*?f n * pochhammer* (*b − of-nat n + 1*) *k*) = ((*?m1 k * ?p* (*of-nat n*) *k*) / *?f n*) * ((*?m1 n * ?p b n*)/*pochhammer* (*b − of-nat n + 1*) *k*)
using *bnz0*
by (*simp add*: *field-simps*)
also have ... = *b gchoose* (*n − k*)
unfolding *th1 th2*
using *kn'* **by** (*simp add*: *gbinomial-def*)
finally have *b gchoose* (*n − k*) = (*?m1 n * ?p b n * ?m1 k * ?p* (*of-nat n*) *k*) / (*?f n * pochhammer* (*b − of-nat n + 1*) *k*) **by** *simp***}**
ultimately have *b gchoose* (*n − k*) = (*?m1 n * ?p b n * ?m1 k * ?p* (*of-nat n*) *k*) / (*?f n * pochhammer* (*b − of-nat n + 1*) *k*)
by (*cases k* =*n*, *auto*)**}**
ultimately have *b gchoose* (*n − k*) = (*?m1 n * ?p b n * ?m1 k * ?p* (*of-nat n*) *k*) / (*?f n * pochhammer* (*b − of-nat n + 1*) *k*) *pochhammer* (*1 + b − of-nat n*) *k* ≠ *0*
using *nz'*
apply (*cases n*=*0*, *auto*)
by (*cases k*, *auto*)**}**
note *th00* = *this*
have *?r* = ((*a + b*) *gchoose n*) * (*of-nat* (*fact n*)/ (*?m1 n * pochhammer* (*− b*) *n*))
unfolding *gbinomial-pochhammer*
using *bn0* **by** (*auto simp add*: *field-simps*)
also have ... = *?l*
unfolding *gbinomial-Vandermonde*[*symmetric*]
apply (*simp add*: *th00*)
unfolding *gbinomial-pochhammer*
using *bn0* **apply** (*simp add*: *setsum-left-distrib setsum-right-distrib field-simps*)
apply (*rule setsum-cong2*)
apply (*drule th00*(*2*))
by (*simp add*: *field-simps power-add*[*symmetric*])
finally show *?thesis* **by** *simp*
qed

lemma *Vandermonde-pochhammer*:
fixes *a* :: *'a*::*field-char-0*
assumes *c*: *ALL i* : {*0..*< *n*}. *c* ≠ *− of-nat i*
shows *setsum* (%*k*. (*pochhammer a k * pochhammer* (*−* (*of-nat n*)) *k*) / (*of-nat*

(*fact k*) * *pochhammer c k*)) {*0..n*} = *pochhammer* (*c* − *a*) *n* / *pochhammer c n*
proof−
let *?a* = − *a*
let *?b* = *c* + *of-nat n* − *1*
have *h*: ∀ *j* ∈{*0..*< *n*}. *?b* ≠ *of-nat j* **using** *c*
apply (*auto simp add*: *algebra-simps of-nat-diff*)
apply (*erule-tac x*= *n* − *j* − *1* **in** *ballE*)
by (*auto simp add*: *of-nat-diff algebra-simps*)
have *th0*: *pochhammer* (− (*?a* + *?b*)) *n* = (− *1*) ^*n* * *pochhammer* (*c* − *a*) *n*
unfolding *pochhammer-minus*[*OF le-refl*]
by (*simp add*: *algebra-simps*)
have *th1*: *pochhammer* (− *?b*) *n* = (− *1*) ^*n* * *pochhammer c n*
unfolding *pochhammer-minus*[*OF le-refl*]
by *simp*
have *nz*: *pochhammer c n* ≠ *0* **using** *c*
by (*simp add*: *pochhammer-eq-0-iff*)
from *Vandermonde-pochhammer-lemma*[**where** *a* = *?a* **and** *b*=*?b* **and** *n*=*n*, *OF h*, *unfolded th0 th1*]
show *?thesis* **using** *nz* **by** (*simp add*: *field-simps setsum-right-distrib*)
qed

39.18.4 Formal trigonometric functions

definition *fps-sin* (*c*::'*a*::*field-char-0*) =
Abs-fps (λ*n*. *if even n then 0 else* (− *1*) ^((*n* − *1*) *div 2*) * *c*^*n* /(*of-nat* (*fact n*)))

definition *fps-cos* (*c*::'*a*::*field-char-0*) =
Abs-fps (λ*n*. *if even n then* (− *1*) ^ (*n div 2*) * *c*^*n* / (*of-nat* (*fact n*)) *else 0*)

lemma *fps-sin-deriv*:
fps-deriv (*fps-sin c*) = *fps-const c* * *fps-cos c*
(**is** *?lhs* = *?rhs*)
proof (*rule fps-ext*)
fix *n*::*nat*
{**assume** *en*: *even n*
have *?lhs*$*n* = *of-nat* (*n*+*1*) * (*fps-sin c* $ (*n*+*1*)) **by** *simp*
also have ... = *of-nat* (*n*+*1*) * ((− *1*) ^(*n div 2*) * *c*^*Suc n* / *of-nat* (*fact* (*Suc n*)))
using *en* **by** (*simp add*: *fps-sin-def*)
also have ... = (− *1*) ^(*n div 2*) * *c*^*Suc n* * (*of-nat* (*n*+*1*) / (*of-nat* (*Suc n*) * *of-nat* (*fact n*)))
unfolding *fact-Suc of-nat-mult*
by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)
also have ... = (− *1*) ^(*n div 2*) *c*^*Suc n* / *of-nat* (*fact n*)
by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)
finally have *?lhs* $*n* = *?rhs*$*n* **using** *en*
by (*simp add*: *fps-cos-def field-simps power-Suc*)}
then show *?lhs* $ *n* = *?rhs* $ *n*

by (*cases even n*, *simp-all add*: *fps-deriv-def fps-sin-def fps-cos-def*)
qed

lemma *fps-cos-deriv*:
fps-deriv (*fps-cos c*) = *fps-const* (− *c*)∗ (*fps-sin c*)
(**is** *?lhs* = *?rhs*)
proof (*rule fps-ext*)
have *th0*: ⋀*n*. − ((− *1*::′*a*) ^ *n*) = (− *1*) ^*Suc n* **by** (*simp add*: *power-Suc*)
have *th1*: ⋀*n*. *odd n* ⟹ *Suc* ((*n* − *1*) *div 2*) = *Suc n div 2*
by (*case-tac n*, *simp-all*)
fix *n*::*nat*
{assume *en*: *odd n*
from *en* **have** *n0*: *n* ≠*0* **by** *presburger*
have *?lhs*$*n* = *of-nat* (*n*+*1*) ∗ (*fps-cos c* $ (*n*+*1*)) **by** *simp*
also have ... = *of-nat* (*n*+*1*) ∗ ((− *1*) ^((*n* + *1*) *div 2*) ∗ *c* ^*Suc n* / *of-nat* (*fact* (*Suc n*)))
using *en* **by** (*simp add*: *fps-cos-def*)
also have ... = (− *1*) ^((*n* + *1*) *div 2*)∗*c* ^*Suc n* ∗ (*of-nat* (*n*+*1*) / (*of-nat* (*Suc n*) ∗ *of-nat* (*fact n*)))
unfolding *fact-Suc of-nat-mult*
by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)
also have ... = (− *1*) ^((*n* + *1*) *div 2*) ∗ *c* ^*Suc n* / *of-nat* (*fact n*)
by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)
also have ... = (− ((− *1*) ^((*n* − *1*) *div 2*))) ∗ *c* ^*Suc n* / *of-nat* (*fact n*)
unfolding *th0* **unfolding** *th1*[*OF en*] **by** *simp*
finally have *?lhs* $*n* = *?rhs*$*n* **using** *en*
by (*simp add*: *fps-sin-def field-simps power-Suc*)**}**
then show *?lhs* $ *n* = *?rhs* $ *n*
by (*cases even n*, *simp-all add*: *fps-deriv-def fps-sin-def*
fps-cos-def)
qed

lemma *fps-sin-cos-sum-of-squares*:
fps-cos c ^ *2* + *fps-sin c* ^ *2* = *1* (**is** *?lhs* = *1*)
proof−
have *fps-deriv ?lhs* = *0*
apply (*simp add*: *fps-deriv-power fps-sin-deriv fps-cos-deriv power-Suc*)
by (*simp add*: *field-simps fps-const-neg*[*symmetric*] *del*: *fps-const-neg*)
then have *?lhs* = *fps-const* (*?lhs* $ *0*)
unfolding *fps-deriv-eq-0-iff* **.**
also have ... = *1*
by (*auto simp add*: *fps-eq-iff numeral-2-eq-2 fps-mult-nth fps-cos-def fps-sin-def*)
finally show *?thesis* **.**
qed

lemma *divide-eq-iff*: *a* ≠ (*0*::′*a*::*field*) ⟹ *x* / *a* = *y* ⟷ *x* = *y* ∗ *a*
by *auto*

lemma *eq-divide-iff*: *a* ≠ (*0*::′*a*::*field*) ⟹ *x* = *y* / *a* ⟷ *x* ∗ *a* = *y*

by *auto*

lemma *fps-sin-nth-0* [*simp*]: *fps-sin c $ 0 = 0*
unfolding *fps-sin-def* **by** *simp*

lemma *fps-sin-nth-1* [*simp*]: *fps-sin c $ 1 = c*
unfolding *fps-sin-def* **by** *simp*

lemma *fps-sin-nth-add-2*:
fps-sin c $ (n + 2) = − (c ∗ c ∗ fps-sin c $ n / (of-nat(n+1) ∗ of-nat(n+2)))
unfolding *fps-sin-def*
apply (*cases n*, *simp*)
apply (*simp add*: *divide-eq-iff eq-divide-iff del*: *of-nat-Suc fact-Suc*)
apply (*simp add*: *of-nat-mult del*: *of-nat-Suc mult-Suc*)
done

lemma *fps-cos-nth-0* [*simp*]: *fps-cos c $ 0 = 1*
unfolding *fps-cos-def* **by** *simp*

lemma *fps-cos-nth-1* [*simp*]: *fps-cos c $ 1 = 0*
unfolding *fps-cos-def* **by** *simp*

lemma *fps-cos-nth-add-2*:
fps-cos c $ (n + 2) = − (c ∗ c ∗ fps-cos c $ n / (of-nat(n+1) ∗ of-nat(n+2)))
unfolding *fps-cos-def*
apply (*simp add*: *divide-eq-iff eq-divide-iff del*: *of-nat-Suc fact-Suc*)
apply (*simp add*: *of-nat-mult del*: *of-nat-Suc mult-Suc*)
done

lemma *nat-induct2*:
⟦*P 0*; *P 1*; ⋀*n*. *P n* ⟹ *P* (*n* + *2*)⟧ ⟹ *P* (*n*::*nat*)
unfolding *One-nat-def numeral-2-eq-2*
apply (*induct n rule*: *nat-less-induct*)
apply (*case-tac n*, *simp*)
apply (*rename-tac m*, *case-tac m*, *simp*)
apply (*rename-tac k*, *case-tac k*, *simp-all*)
done

lemma *nat-add-1-add-1*: (*n*::*nat*) + *1* + *1* = *n* + *2*
by *simp*

lemma *eq-fps-sin*:
assumes *0*: *a $ 0 = 0* **and** *1*: *a $ 1 = c*
and *2*: *fps-deriv* (*fps-deriv a*) = − (*fps-const c* ∗ *fps-const c* ∗ *a*)
shows *a* = *fps-sin c*
apply (*rule fps-ext*)
apply (*induct-tac n rule*: *nat-induct2*)
apply (*simp add*: *fps-sin-nth-0 0*)
apply (*simp add*: *fps-sin-nth-1 1 del*: *One-nat-def*)

apply (*rename-tac m*, *cut-tac f=λa. a $ m* **in** *arg-cong* [*OF 2*])
apply (*simp add*: *nat-add-1-add-1 fps-sin-nth-add-2*
del: *One-nat-def of-nat-Suc of-nat-add add-2-eq-Suc'*)
apply (*subst minus-divide-left*)
apply (*subst eq-divide-iff*)
apply (*simp del*: *of-nat-add of-nat-Suc*)
apply (*simp only*: *mult-ac*)
done

lemma *eq-fps-cos*:
assumes *0*: *a $ 0 = 1* **and** *1*: *a $ 1 = 0*
and *2*: *fps-deriv* (*fps-deriv a*) = − (*fps-const c* ∗ *fps-const c* ∗ *a*)
shows *a = fps-cos c*
apply (*rule fps-ext*)
apply (*induct-tac n rule*: *nat-induct2*)
apply (*simp add*: *fps-cos-nth-0 0*)
apply (*simp add*: *fps-cos-nth-1 1 del*: *One-nat-def*)
apply (*rename-tac m*, *cut-tac f=λa. a $ m* **in** *arg-cong* [*OF 2*])
apply (*simp add*: *nat-add-1-add-1 fps-cos-nth-add-2*
del: *One-nat-def of-nat-Suc of-nat-add add-2-eq-Suc'*)
apply (*subst minus-divide-left*)
apply (*subst eq-divide-iff*)
apply (*simp del*: *of-nat-add of-nat-Suc*)
apply (*simp only*: *mult-ac*)
done

lemma *mult-nth-0* [*simp*]: (*a* ∗ *b*) $ *0 = a $ 0* ∗ *b $ 0*
by (*simp add*: *fps-mult-nth*)

lemma *mult-nth-1* [*simp*]: (*a* ∗ *b*) $ *1 = a $ 0* ∗ *b $ 1* + *a $ 1* ∗ *b $ 0*
by (*simp add*: *fps-mult-nth*)

lemma *fps-sin-add*:
fps-sin (*a* + *b*) = *fps-sin a* ∗ *fps-cos b* + *fps-cos a* ∗ *fps-sin b*
apply (*rule eq-fps-sin* [*symmetric*], *simp*, *simp del*: *One-nat-def*)
apply (*simp del*: *fps-const-neg fps-const-add fps-const-mult*
add: *fps-const-add* [*symmetric*] *fps-const-neg* [*symmetric*]
fps-sin-deriv fps-cos-deriv algebra-simps)
done

lemma *fps-cos-add*:
fps-cos (*a* + *b*) = *fps-cos a* ∗ *fps-cos b* − *fps-sin a* ∗ *fps-sin b*
apply (*rule eq-fps-cos* [*symmetric*], *simp*, *simp del*: *One-nat-def*)
apply (*simp del*: *fps-const-neg fps-const-add fps-const-mult*
add: *fps-const-add* [*symmetric*] *fps-const-neg* [*symmetric*]
fps-sin-deriv fps-cos-deriv algebra-simps)
done

lemma *fps-sin-even*: *fps-sin* (− *c*) = − *fps-sin c*

by (*auto simp add*: *fps-eq-iff fps-sin-def*)

lemma *fps-cos-odd*: *fps-cos* (− *c*) = *fps-cos c*
by (*auto simp add*: *fps-eq-iff fps-cos-def*)

definition *fps-tan c* = *fps-sin c* / *fps-cos c*

lemma *fps-tan-deriv*: *fps-deriv*(*fps-tan c*) = *fps-const c*/ (*fps-cos c* ^ *2*)
proof−
have *th0*: *fps-cos c* $ *0* ≠ *0* **by** (*simp add*: *fps-cos-def*)
show *?thesis*
using *fps-sin-cos-sum-of-squares*[*of c*]
apply (*simp add*: *fps-tan-def fps-divide-deriv*[*OF th0*] *fps-sin-deriv fps-cos-deriv add*: *fps-const-neg*[*symmetric*] *field-simps power2-eq-square del*: *fps-const-neg*)
unfolding *right-distrib*[*symmetric*]
by *simp*
qed

Connection to E c over the complex numbers — Euler and De Moivre

lemma *Eii-sin-cos*:
E (*ii* ∗ *c*) = *fps-cos c* + *fps-const ii* ∗ *fps-sin c*
(**is** *?l* = *?r*)
proof−
{fix *n*::*nat*
{assume *en*: *even n*
from *en* **obtain** *m* **where** *m*: *n* = *2*∗*m*
unfolding *even-mult-two-ex* **by** *blast*

have *?l* $*n* = *?r*$*n*
by (*simp add*: *m fps-sin-def fps-cos-def power-mult-distrib power-mult power-minus*)**}**
moreover
{assume *on*: *odd n*
from *on* **obtain** *m* **where** *m*: *n* = *2*∗*m* + *1*
unfolding *odd-nat-equiv-def2* **by** (*auto simp add*: *nat-mult-2*)
have *?l* $*n* = *?r*$*n*
by (*simp add*: *m fps-sin-def fps-cos-def power-mult-distrib power-mult power-minus*)**}**
ultimately have *?l* $*n* = *?r*$*n* **by** *blast***}**
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

lemma *E-minus-ii-sin-cos*: *E* (− (*ii* ∗ *c*)) = *fps-cos c* − *fps-const ii* ∗ *fps-sin c*
unfolding *minus-mult-right Eii-sin-cos* **by** (*simp add*: *fps-sin-even fps-cos-odd*)

lemma *fps-const-minus*: *fps-const* (*c*::′*a*::*group-add*) − *fps-const d* = *fps-const* (*c* − *d*)
by (*simp add*: *fps-eq-iff fps-const-def*)

lemma *fps-number-of-fps-const*: *number-of i = fps-const* (*number-of i* :: *'a*:: {*comm-ring-1*, *number-ring*})
apply (*subst* (*2*) *number-of-eq*)
apply(*rule int-induct* [*of - 0*])
apply (*simp-all add*: *number-of-fps-def*)
by (*simp-all add*: *fps-const-add*[*symmetric*] *fps-const-minus*[*symmetric*])

lemma *fps-cos-Eii*:
fps-cos c = (*E* (*ii* ∗ *c*) + *E* (− *ii* ∗ *c*)) / *fps-const 2*
proof−
have *th*: *fps-cos c* + *fps-cos c* = *fps-cos c* ∗ *fps-const 2*
by (*simp add*: *fps-eq-iff fps-number-of-fps-const complex-number-of-def* [*symmetric*])
show *?thesis*
unfolding *Eii-sin-cos minus-mult-commute*
by (*simp add*: *fps-sin-even fps-cos-odd fps-number-of-fps-const*
fps-divide-def fps-const-inverse th complex-number-of-def [*symmetric*])
qed

lemma *fps-sin-Eii*:
fps-sin c = (*E* (*ii* ∗ *c*) − *E* (− *ii* ∗ *c*)) / *fps-const* (*2*∗*ii*)
proof−
have *th*: *fps-const* i ∗ *fps-sin c* + *fps-const* i ∗ *fps-sin c* = *fps-sin c* ∗ *fps-const* (*2* ∗ *ii*)
by (*simp add*: *fps-eq-iff fps-number-of-fps-const complex-number-of-def* [*symmetric*])
show *?thesis*
unfolding *Eii-sin-cos minus-mult-commute*
by (*simp add*: *fps-sin-even fps-cos-odd fps-divide-def fps-const-inverse th*)
qed

lemma *fps-tan-Eii*:
fps-tan c = (*E* (*ii* ∗ *c*) − *E* (− *ii* ∗ *c*)) / (*fps-const ii* ∗ (*E* (*ii* ∗ *c*) + *E* (− *ii* ∗ *c*)))
unfolding *fps-tan-def fps-sin-Eii fps-cos-Eii mult-minus-left E-neg*
apply (*simp add*: *fps-divide-def fps-inverse-mult fps-const-mult*[*symmetric*] *fps-const-inverse del*: *fps-const-mult*)
by *simp*

lemma *fps-demoivre*: (*fps-cos a* + *fps-const ii* ∗ *fps-sin a*) ^*n* = *fps-cos* (*of-nat n* ∗ *a*) + *fps-const ii* ∗ *fps-sin* (*of-nat n* ∗ *a*)
unfolding *Eii-sin-cos*[*symmetric*] *E-power-mult*
by (*simp add*: *mult-ac*)

39.19 Hypergeometric series

definition *F as bs* (*c*::*'a*::{*field-char-0*, *field-inverse-zero*}) = *Abs-fps* (%*n*. (*foldl* (%*r a*. *r*∗ *pochhammer a n*) *1 as* ∗ *c* ^*n*)/ (*foldl* (%*r b*. *r* ∗ *pochhammer b n*) *1 bs* ∗ *of-nat* (*fact n*)))

lemma *F-nth*[*simp*]: *F as bs c* $ *n* = (*foldl* (%*r a*. *r*∗ *pochhammer a n*) *1 as* ∗

*c ^ n)/ (foldl (%r b. r * pochhammer b n) 1 bs * of-nat (fact n))*
by (*simp add*: *F-def*)

lemma *foldl-mult-start*:
*foldl (%r x. r * f x) (v::'a::comm-ring-1) as * x = foldl (%r x. r * f x) (v * x) as*
by (*induct as arbitrary*: *x v*, *auto simp add*: *algebra-simps*)

lemma *foldr-mult-foldl*: *foldr (%x r. r * f x) as v = foldl (%r x. r * f x) (v :: 'a::comm-ring-1) as*
by (*induct as arbitrary*: *v*, *auto simp add*: *foldl-mult-start*)

lemma *F-nth-alt*: *F as bs c $ n = foldr (λa r. r * pochhammer a n) as (c ^ n) / foldr (λb r. r * pochhammer b n) bs (of-nat (fact n))*
by (*simp add*: *foldl-mult-start foldr-mult-foldl*)

lemma *F-E*[*simp*]: *F [] [] c = E c*
by (*simp add*: *fps-eq-iff*)

lemma *F-1-0*[*simp*]: *F [1] [] c = 1/(1 − fps-const c * X)*
proof−
let *?a = (Abs-fps (λn. 1)) oo (fps-const c * X)*
have *th0*: *(fps-const c * X) $ 0 = 0* **by** *simp*
show *?thesis* **unfolding** *gp*[*OF th0*, *symmetric*]
by (*auto simp add*: *fps-eq-iff pochhammer-fact*[*symmetric*] *fps-compose-nth power-mult-distrib cond-value-iff setsum-delta' cong del*: *if-weak-cong*)
qed

lemma *F-B*[*simp*]: *F [−a] [] (− 1) = fps-binomial a*
by (*simp add*: *fps-eq-iff gbinomial-pochhammer algebra-simps*)

lemma *F-0*[*simp*]: *F as bs c $0 = 1*
apply *simp*
apply (*subgoal-tac ALL as. foldl (%(r::'a) (a::'a). r) 1 as = 1*)
apply *auto*
apply (*induct-tac as*, *auto*)
done

lemma *foldl-prod-prod*: *foldl (%(r::'b::comm-ring-1) (x::'a::comm-ring-1). r * f x) v as * foldl (%r x. r * g x) w as = foldl (%r x. r * f x * g x) (v*w) as*
by (*induct as arbitrary*: *v w*, *auto simp add*: *algebra-simps*)

lemma *F-rec*: *F as bs c $ Suc n = ((foldl (%r a. r* (a + of-nat n)) c as)/ (foldl (%r b. r * (b + of-nat n)) (of-nat (Suc n)) bs)) * F as bs c $ n*
apply (*simp del*: *of-nat-Suc of-nat-add fact-Suc*)
apply (*simp add*: *foldl-mult-start del*: *fact-Suc of-nat-Suc*)
unfolding *foldl-prod-prod*[*unfolded foldl-mult-start*] *pochhammer-Suc*
by (*simp add*: *algebra-simps of-nat-mult*)

lemma *XD-nth*[*simp*]: *XD a $ n = (if n=0 then 0 else of-nat n * a$n)*
by (*simp add*: *XD-def*)

lemma *XD-0th*[*simp*]: *XD a $ 0 = 0* **by** *simp*
lemma *XD-Suc*[*simp*]: *XD a $ Suc n = of-nat (Suc n) * a $ Suc n* **by** *simp*

definition *XDp c a = XD a + fps-const c * a*

lemma *XDp-nth*[*simp*]: *XDp c a $ n = (c + of-nat n) * a$n*
by (*simp add*: *XDp-def algebra-simps*)

lemma *XDp-commute*:
shows *XDp b o XDp (c::'a::comm-ring-1) = XDp c o XDp b*
by (*auto simp add*: *XDp-def expand-fun-eq fps-eq-iff algebra-simps*)

lemma *XDp0*[*simp*]: *XDp 0 = XD*
by (*simp add*: *expand-fun-eq fps-eq-iff*)

lemma *XDp-fps-integral*[*simp*]:*XDp 0 (fps-integral a c) = X * a*
by (*simp add*: *fps-eq-iff fps-integral-def*)

lemma *F-minus-nat*:
F [− of-nat n] [− of-nat (n + m)] (c::'a::{field-char-0, field-inverse-zero}) $ k
*= (if k <= n then pochhammer (− of-nat n) k * c ^ k /*
*(pochhammer (− of-nat (n + m)) k * of-nat (fact k)) else 0)*
F [− of-nat m] [− of-nat (m + n)] (c::'a::{field-char-0, field-inverse-zero}) $ k
*= (if k <= m then pochhammer (− of-nat m) k * c ^ k /*
*(pochhammer (− of-nat (m + n)) k * of-nat (fact k)) else 0)*
by (*auto simp add*: *pochhammer-eq-0-iff*)

lemma *setsum-eq-if*: *setsum f {(n::nat) .. m} = (if m < n then 0 else f n + setsum f {n+1 .. m})*
apply *simp*
apply (*subst setsum-insert*[*symmetric*])
by (*auto simp add*: *not-less setsum-head-Suc*)

lemma *pochhammer-rec-if*:
*pochhammer a n = (if n = 0 then 1 else a * pochhammer (a + 1) (n − 1))*
by (*cases n*, *simp-all add*: *pochhammer-rec*)

lemma *XDp-foldr-nth*[*simp*]: *foldr (%c r. XDp c o r) cs (%c. XDp c a) c0 $ n =*

*foldr (%c r. (c + of-nat n) * r) cs (c0 + of-nat n) * a$n*
by (*induct cs arbitrary*: *c0*, *auto simp add*: *algebra-simps*)

lemma *genric-XDp-foldr-nth*:
assumes
f: *ALL n c a. f c a $ n = (of-nat n + k c) * a$n*

shows *foldr* (%*c r*. *f c o r*) *cs* (%*c*. *g c a*) *c0* $ *n* =
foldr (%*c r*. (*k c* + *of-nat n*) ∗ *r*) *cs* (*g c0 a* $ *n*)
by (*induct cs arbitrary*: *c0*, *auto simp add*: *algebra-simps f*)

end

40 Fraction-Field: A formalization of the fraction field of any integral domain A generalization of Rat.thy from int to any integral domain

theory *Fraction-Field*
imports *Main*
begin

40.1 General fractions construction

40.1.1 Construction of the type of fractions

definition *fractrel* :: (('*a*::*idom* ∗ '*a*) ∗ ('*a* ∗ '*a*)) *set* **where**
fractrel == {(*x*, *y*). *snd x* ≠ *0* ∧ *snd y* ≠ *0* ∧ *fst x* ∗ *snd y* = *fst y* ∗ *snd x*}

lemma *fractrel-iff* [*simp*]:
(*x*, *y*) ∈ *fractrel* ⟷ *snd x* ≠ *0* ∧ *snd y* ≠ *0* ∧ *fst x* ∗ *snd y* = *fst y* ∗ *snd x*
by (*simp add*: *fractrel-def*)

lemma *refl-fractrel*: *refl-on* {*x*. *snd x* ≠ *0*} *fractrel*
by (*auto simp add*: *refl-on-def fractrel-def*)

lemma *sym-fractrel*: *sym fractrel*
by (*simp add*: *fractrel-def sym-def*)

lemma *trans-fractrel*: *trans fractrel*
proof (*rule transI*, *unfold split-paired-all*)
fix *a b a' b' a'' b''* :: '*a*
assume *A*: ((*a*, *b*), (*a'*, *b'*)) ∈ *fractrel*
assume *B*: ((*a'*, *b'*), (*a''*, *b''*)) ∈ *fractrel*
have *b'* ∗ (*a* ∗ *b''*) = *b''* ∗ (*a* ∗ *b'*) **by** (*simp add*: *mult-ac*)
also from *A* **have** *a* ∗ *b'* = *a'* ∗ *b* **by** *auto*
also have *b''* ∗ (*a'* ∗ *b*) = *b* ∗ (*a'* ∗ *b''*) **by** (*simp add*: *mult-ac*)
also from *B* **have** *a'* ∗ *b''* = *a''* ∗ *b'* **by** *auto*
also have *b* ∗ (*a''* ∗ *b'*) = *b'* ∗ (*a''* ∗ *b*) **by** (*simp add*: *mult-ac*)
finally have *b'* ∗ (*a* ∗ *b''*) = *b'* ∗ (*a''* ∗ *b*) **.**
moreover from *B* **have** *b'* ≠ *0* **by** *auto*
ultimately have *a* ∗ *b''* = *a''* ∗ *b* **by** *simp*
with *A B* **show** ((*a*, *b*), (*a''*, *b''*)) ∈ *fractrel* **by** *auto*
qed

lemma *equiv-fractrel*: *equiv* {*x*. *snd x* ≠ *0*} *fractrel*
by (*rule equiv.intro* [*OF refl-fractrel sym-fractrel trans-fractrel*])

lemmas *UN-fractrel* = *UN-equiv-class* [*OF equiv-fractrel*]
lemmas *UN-fractrel2* = *UN-equiv-class2* [*OF equiv-fractrel equiv-fractrel*]

lemma *equiv-fractrel-iff* [*iff*]:
assumes *snd x* ≠ *0* **and** *snd y* ≠ *0*
shows *fractrel* `` {*x*} = *fractrel* `` {*y*} ⟷ (*x*, *y*) ∈ *fractrel*
by (*rule eq-equiv-class-iff*, *rule equiv-fractrel*) (*auto simp add*: *assms*)

typedef *'a fract* = {(*x*::*'a*×*'a*). *snd x* ≠ (*0*::*'a*::*idom*)} // *fractrel*
proof
have (*0*::*'a*, *1*::*'a*) ∈ {*x*. *snd x* ≠ *0*} **by** *simp*
then show *fractrel* `` {(*0*::*'a*, *1*)} ∈ {*x*. *snd x* ≠ *0*} // *fractrel* **by** (*rule quotientI*)
qed

lemma *fractrel-in-fract* [*simp*]: *snd x* ≠ *0* ⟹ *fractrel* `` {*x*} ∈ *fract*
by (*simp add*: *fract-def quotientI*)

declare *Abs-fract-inject* [*simp*] *Abs-fract-inverse* [*simp*]

40.1.2 Representation and basic operations

definition
Fract :: *'a*::*idom* ⇒ *'a* ⇒ *'a fract* **where**
[*code del*]: *Fract a b* = *Abs-fract* (*fractrel* `` {*if b* = *0 then* (*0*, *1*) *else* (*a*, *b*)})

code-datatype *Fract*

lemma *Fract-cases* [*case-names Fract*, *cases type*: *fract*]:
assumes ⋀*a b*. *q* = *Fract a b* ⟹ *b* ≠ *0* ⟹ *C*
shows *C*
using *assms* **by** (*cases q*) (*clarsimp simp add*: *Fract-def fract-def quotient-def*)

lemma *Fract-induct* [*case-names Fract*, *induct type*: *fract*]:
assumes ⋀*a b*. *b* ≠ *0* ⟹ *P* (*Fract a b*)
shows *P q*
using *assms* **by** (*cases q*) *simp*

lemma *eq-fract*:
shows ⋀*a b c d*. *b* ≠ *0* ⟹ *d* ≠ *0* ⟹ *Fract a b* = *Fract c d* ⟷ *a* ∗ *d* = *c* ∗ *b*
and ⋀*a*. *Fract a 0* = *Fract 0 1*
and ⋀*a c*. *Fract 0 a* = *Fract 0 c*
by (*simp-all add*: *Fract-def*)

instantiation *fract* :: (*idom*) {*comm-ring-1*, *power*}
begin

definition
Zero-fract-def [*code*, *code-unfold*]: $0 = Fract\ 0\ 1$

definition
One-fract-def [*code*, *code-unfold*]: $1 = Fract\ 1\ 1$

definition
add-fract-def [*code del*]:
$q + r = Abs\text{-}fract\ (\bigcup x \in Rep\text{-}fract\ q.\ \bigcup y \in Rep\text{-}fract\ r.$
$fractrel\ ``\ \{(fst\ x * snd\ y + fst\ y * snd\ x,\ snd\ x * snd\ y)\})$

lemma *add-fract* [*simp*]:
assumes $b \neq (0::'a::idom)$ **and** $d \neq 0$
shows $Fract\ a\ b + Fract\ c\ d = Fract\ (a * d + c * b)\ (b * d)$
proof −
have $(\lambda x\ y.\ fractrel``\{(fst\ x * snd\ y + fst\ y * snd\ x,\ snd\ x * snd\ y :: 'a)\})$
respects2 fractrel
apply (*rule equiv-fractrel* [*THEN congruent2-commuteI*]) **apply** (*auto simp add*: *algebra-simps*)
unfolding *mult-assoc*[*symmetric*] **.**
with *assms* **show** *?thesis* **by** (*simp add*: *Fract-def add-fract-def UN-fractrel2*)
qed

definition
minus-fract-def [*code del*]:
$- q = Abs\text{-}fract\ (\bigcup x \in Rep\text{-}fract\ q.\ fractrel\ ``\ \{(- fst\ x,\ snd\ x)\})$

lemma *minus-fract* [*simp*, *code*]: $- Fract\ a\ b = Fract\ (- a)\ (b::'a::idom)$
proof −
have $(\lambda x.\ fractrel\ ``\ \{(- fst\ x,\ snd\ x :: 'a)\})\ respects\ fractrel$
by (*simp add*: *congruent-def*)
then show *?thesis* **by** (*simp add*: *Fract-def minus-fract-def UN-fractrel*)
qed

lemma *minus-fract-cancel* [*simp*]: $Fract\ (- a)\ (- b) = Fract\ a\ b$
by (*cases* $b = 0$) (*simp-all add*: *eq-fract*)

definition
diff-fract-def [*code del*]: $q - r = q + - (r::'a\ fract)$

lemma *diff-fract* [*simp*]:
assumes $b \neq 0$ **and** $d \neq 0$
shows $Fract\ a\ b - Fract\ c\ d = Fract\ (a * d - c * b)\ (b * d)$
using *assms* **by** (*simp add*: *diff-fract-def diff-minus*)

definition
mult-fract-def [*code del*]:
$q * r = Abs\text{-}fract\ (\bigcup x \in Rep\text{-}fract\ q.\ \bigcup y \in Rep\text{-}fract\ r.$

*fractrel``{(fst x * fst y, snd x * snd y)})*

lemma *mult-fract* [*simp*]: *Fract* (*a*::'*a*::*idom*) *b* * *Fract c d* = *Fract* (*a* * *c*) (*b* * *d*)
proof −
have (λ*x y*. *fractrel `` {(fst x * fst y, snd x * snd y* :: '*a*)}) *respects2 fractrel*
apply (*rule equiv-fractrel* [*THEN congruent2-commuteI*]) **apply** (*auto simp add*: *algebra-simps*)
unfolding *mult-assoc*[*symmetric*] .
then show *?thesis* **by** (*simp add*: *Fract-def mult-fract-def UN-fractrel2*)
qed

lemma *mult-fract-cancel*:
assumes *c* ≠ *0*
shows *Fract* (*c* * *a*) (*c* * *b*) = *Fract a b*
proof −
from *assms* **have** *Fract c c* = *Fract 1 1* **by** (*simp add*: *Fract-def*)
then show *?thesis* **by** (*simp add*: *mult-fract* [*symmetric*])
qed

instance proof
fix *q r s* :: '*a fract* **show** (*q* * *r*) * *s* = *q* * (*r* * *s*)
by (*cases q*, *cases r*, *cases s*) (*simp add*: *eq-fract algebra-simps*)
next
fix *q r* :: '*a fract* **show** *q* * *r* = *r* * *q*
by (*cases q*, *cases r*) (*simp add*: *eq-fract algebra-simps*)
next
fix *q* :: '*a fract* **show** *1* * *q* = *q*
by (*cases q*) (*simp add*: *One-fract-def eq-fract*)
next
fix *q r s* :: '*a fract* **show** (*q* + *r*) + *s* = *q* + (*r* + *s*)
by (*cases q*, *cases r*, *cases s*) (*simp add*: *eq-fract algebra-simps*)
next
fix *q r* :: '*a fract* **show** *q* + *r* = *r* + *q*
by (*cases q*, *cases r*) (*simp add*: *eq-fract algebra-simps*)
next
fix *q* :: '*a fract* **show** *0* + *q* = *q*
by (*cases q*) (*simp add*: *Zero-fract-def eq-fract*)
next
fix *q* :: '*a fract* **show** − *q* + *q* = *0*
by (*cases q*) (*simp add*: *Zero-fract-def eq-fract*)
next
fix *q r* :: '*a fract* **show** *q* − *r* = *q* + − *r*
by (*cases q*, *cases r*) (*simp add*: *eq-fract*)
next
fix *q r s* :: '*a fract* **show** (*q* + *r*) * *s* = *q* * *s* + *r* * *s*
by (*cases q*, *cases r*, *cases s*) (*simp add*: *eq-fract algebra-simps*)
next
show (*0*::'*a fract*) ≠ *1* **by** (*simp add*: *Zero-fract-def One-fract-def eq-fract*)

qed

end

lemma *of-nat-fract*: *of-nat k = Fract (of-nat k) 1*
by (*induct k*) (*simp-all add*: *Zero-fract-def One-fract-def*)

lemma *Fract-of-nat-eq*: *Fract (of-nat k) 1 = of-nat k*
by (*rule of-nat-fract* [*symmetric*])

lemma *fract-collapse* [*code-post*]:
Fract 0 k = 0
Fract 1 1 = 1
Fract k 0 = 0
by (*cases k = 0*)
(*simp-all add*: *Zero-fract-def One-fract-def eq-fract Fract-def*)

lemma *fract-expand* [*code-unfold*]:
0 = Fract 0 1
1 = Fract 1 1
by (*simp-all add*: *fract-collapse*)

lemma *Fract-cases-nonzero* [*case-names Fract 0*]:
assumes *Fract*: $\bigwedge a\ b.\ q = Fract\ a\ b \Longrightarrow b \neq 0 \Longrightarrow a \neq 0 \Longrightarrow C$
assumes *0*: $q = 0 \Longrightarrow C$
shows *C*
proof (*cases q = 0*)
case *True* **then show** *C* **using** *0* **by** *auto*
next
case *False*
then obtain *a b* **where** *q = Fract a b* **and** *b ≠ 0* **by** (*cases q*) *auto*
moreover with *False* **have** *0 ≠ Fract a b* **by** *simp*
with ‹*b ≠ 0*› **have** *a ≠ 0* **by** (*simp add*: *Zero-fract-def eq-fract*)
with *Fract* ‹*q = Fract a b*› ‹*b ≠ 0*› **show** *C* **by** *auto*
qed

40.1.3 The field of rational numbers

context *idom*
begin
subclass *ring-no-zero-divisors* **..**
thm *mult-eq-0-iff*
end

instantiation *fract* :: (*idom*) *field-inverse-zero*
begin

definition
inverse-fract-def [*code del*]:

inverse q = Abs-fract (⋃x ∈ Rep-fract q.
fractrel `` {if fst x = 0 then (0, 1) else (snd x, fst x)})

lemma *inverse-fract* [*simp*]: *inverse (Fract a b) = Fract (b::'a::idom) a*
proof −
have *stupid*: *⋀x. (0::'a) = x ⟷ x = 0* **by** *auto*
have *(λx. fractrel `` {if fst x = 0 then (0, 1) else (snd x, fst x :: 'a)}) respects fractrel*
by (*auto simp add*: *congruent-def stupid algebra-simps*)
then show *?thesis* **by** (*simp add*: *Fract-def inverse-fract-def UN-fractrel*)
qed

definition
divide-fract-def [*code del*]: *q / r = q * inverse (r:: 'a fract)*

lemma *divide-fract* [*simp*]: *Fract a b / Fract c d = Fract (a * d) (b * c)*
by (*simp add*: *divide-fract-def*)

instance proof
fix *q :: 'a fract*
assume *q ≠ 0*
then show *inverse q * q = 1* **apply** (*cases q rule*: *Fract-cases-nonzero*)
by (*simp-all add*: *mult-fract inverse-fract fract-expand eq-fract mult-commute*)
next
fix *q r :: 'a fract*
show *q / r = q * inverse r* **by** (*simp add*: *divide-fract-def*)
next
show *inverse 0 = (0:: 'a fract)* **by** (*simp add*: *fract-expand*)
(*simp add*: *fract-collapse*)
qed

end

40.1.4 The ordered field of fractions over an ordered idom

lemma *le-congruent2*:
(λx y::'a × 'a::linordered-idom.
*{(fst x * snd y)*(snd x * snd y) ≤ (fst y * snd x)*(snd x * snd y)})*
respects2 fractrel
proof (*clarsimp simp add*: *congruent2-def*)
fix *a b a' b' c d c' d' :: 'a*
assume *neq*: *b ≠ 0 b' ≠ 0 d ≠ 0 d' ≠ 0*
assume *eq1*: *a * b' = a' * b*
assume *eq2*: *c * d' = c' * d*

let *?le = λa b c d. ((a * d) * (b * d) ≤ (c * b) * (b * d))*
{
fix *a b c d x :: 'a* **assume** *x*: *x ≠ 0*

```
    have ?le a b c d = ?le (a * x) (b * x) c d
    proof −
      from x have 0 < x * x by (auto simp add: zero-less-mult-iff)
      hence ?le a b c d =
          ((a * d) * (b * d) * (x * x) ≤ (c * b) * (b * d) * (x * x))
        by (simp add: mult-le-cancel-right)
      also have ... = ?le (a * x) (b * x) c d
        by (simp add: mult-ac)
      finally show ?thesis .
    qed
  } note le-factor = this

  let ?D = b * d and ?D' = b' * d'
  from neq have D: ?D ≠ 0 by simp
  from neq have ?D' ≠ 0 by simp
  hence ?le a b c d = ?le (a * ?D') (b * ?D') c d
    by (rule le-factor)
  also have ... = ((a * b') * ?D * ?D' * d * d' ≤ (c * d') * ?D * ?D' * b * b')
    by (simp add: mult-ac)
  also have ... = ((a' * b) * ?D * ?D' * d * d' ≤ (c' * d) * ?D * ?D' * b * b')
    by (simp only: eq1 eq2)
  also have ... = ?le (a' * ?D) (b' * ?D) c' d'
    by (simp add: mult-ac)
  also from D have ... = ?le a' b' c' d'
    by (rule le-factor [symmetric])
  finally show ?le a b c d = ?le a' b' c' d' .
qed

instantiation fract :: (linordered-idom) linorder
begin

definition
  le-fract-def [code del]:
   q ≤ r ⟷ contents (⋃x ∈ Rep-fract q. ⋃y ∈ Rep-fract r.
     {(fst x * snd y)*(snd x * snd y) ≤ (fst y * snd x)*(snd x * snd y)})

definition
  less-fract-def [code del]: z < (w::'a fract) ⟷ z ≤ w ∧ ¬ w ≤ z

lemma le-fract [simp]:
  assumes b ≠ 0 and d ≠ 0
  shows Fract a b ≤ Fract c d ⟷ (a * d) * (b * d) ≤ (c * b) * (b * d)
by (simp add: Fract-def le-fract-def le-congruent2 UN-fractrel2 assms)

lemma less-fract [simp]:
  assumes b ≠ 0 and d ≠ 0
  shows Fract a b < Fract c d ⟷ (a * d) * (b * d) < (c * b) * (b * d)
by (simp add: less-fract-def less-le-not-le mult-ac assms)
```

```
instance proof
  fix q r s :: 'a fract
  assume q ≤ r and r ≤ s thus q ≤ s
  proof (induct q, induct r, induct s)
    fix a b c d e f :: 'a
    assume neq: b ≠ 0  d ≠ 0  f ≠ 0
    assume 1: Fract a b ≤ Fract c d and 2: Fract c d ≤ Fract e f
    show Fract a b ≤ Fract e f
    proof −
      from neq obtain bb: 0 < b * b and dd: 0 < d * d and ff: 0 < f * f
        by (auto simp add: zero-less-mult-iff linorder-neq-iff)
      have (a * d) * (b * d) * (f * f) ≤ (c * b) * (b * d) * (f * f)
      proof −
        from neq 1 have (a * d) * (b * d) ≤ (c * b) * (b * d)
          by simp
        with ff show ?thesis by (simp add: mult-le-cancel-right)
      qed
      also have ... = (c * f) * (d * f) * (b * b)
        by (simp only: mult-ac)
      also have ... ≤ (e * d) * (d * f) * (b * b)
      proof −
        from neq 2 have (c * f) * (d * f) ≤ (e * d) * (d * f)
          by simp
        with bb show ?thesis by (simp add: mult-le-cancel-right)
      qed
      finally have (a * f) * (b * f) * (d * d) ≤ e * b * (b * f) * (d * d)
        by (simp only: mult-ac)
      with dd have (a * f) * (b * f) ≤ (e * b) * (b * f)
        by (simp add: mult-le-cancel-right)
      with neq show ?thesis by simp
    qed
  qed
next
  fix q r :: 'a fract
  assume q ≤ r and r ≤ q thus q = r
  proof (induct q, induct r)
    fix a b c d :: 'a
    assume neq: b ≠ 0  d ≠ 0
    assume 1: Fract a b ≤ Fract c d and 2: Fract c d ≤ Fract a b
    show Fract a b = Fract c d
    proof −
      from neq 1 have (a * d) * (b * d) ≤ (c * b) * (b * d)
        by simp
      also have ... ≤ (a * d) * (b * d)
      proof −
        from neq 2 have (c * b) * (d * b) ≤ (a * d) * (d * b)
          by simp
        thus ?thesis by (simp only: mult-ac)
      qed
```

```
      finally have (a * d) * (b * d) = (c * b) * (b * d) .
      moreover from neq have b * d ≠ 0 by simp
      ultimately have a * d = c * b by simp
      with neq show ?thesis by (simp add: eq-fract)
    qed
  qed
next
  fix q r :: 'a fract
  show q ≤ q
    by (induct q) simp
  show (q < r) = (q ≤ r ∧ ¬ r ≤ q)
    by (simp only: less-fract-def)
  show q ≤ r ∨ r ≤ q
    by (induct q, induct r)
       (simp add: mult-commute, rule linorder-linear)
qed

end

instantiation fract :: (linordered-idom) {distrib-lattice, abs-if, sgn-if}
begin

definition
  abs-fract-def: |q| = (if q < 0 then −q else (q::'a fract))

definition
  sgn-fract-def:
    sgn (q::'a fract) = (if q=0 then 0 else if 0<q then 1 else − 1)

theorem abs-fract [simp]: |Fract a b| = Fract |a| |b|
  by (auto simp add: abs-fract-def Zero-fract-def le-less
      eq-fract zero-less-mult-iff mult-less-0-iff split: abs-split)

definition
  inf-fract-def:
    (inf :: 'a fract ⇒ 'a fract ⇒ 'a fract) = min

definition
  sup-fract-def:
    (sup :: 'a fract ⇒ 'a fract ⇒ 'a fract) = max

instance by intro-classes
  (auto simp add: abs-fract-def sgn-fract-def
    min-max.sup-inf-distrib1 inf-fract-def sup-fract-def)

end

instance fract :: (linordered-idom) linordered-field-inverse-zero
proof
```

```
  fix q r s :: 'a fract
  show q ≤ r ==> s + q ≤ s + r
  proof (induct q, induct r, induct s)
    fix a b c d e f :: 'a
    assume neq: b ≠ 0  d ≠ 0  f ≠ 0
    assume le: Fract a b ≤ Fract c d
    show Fract e f + Fract a b ≤ Fract e f + Fract c d
    proof −
      let ?F = f * f from neq have F: 0 < ?F
        by (auto simp add: zero-less-mult-iff)
      from neq le have (a * d) * (b * d) ≤ (c * b) * (b * d)
        by simp
      with F have (a * d) * (b * d) * ?F * ?F ≤ (c * b) * (b * d) * ?F * ?F
        by (simp add: mult-le-cancel-right)
      with neq show ?thesis by (simp add: field-simps)
    qed
  qed
  show q < r ==> 0 < s ==> s * q < s * r
  proof (induct q, induct r, induct s)
    fix a b c d e f :: 'a
    assume neq: b ≠ 0  d ≠ 0  f ≠ 0
    assume le: Fract a b < Fract c d
    assume gt: 0 < Fract e f
    show Fract e f * Fract a b < Fract e f * Fract c d
    proof −
      let ?E = e * f and ?F = f * f
      from neq gt have 0 < ?E
        by (auto simp add: Zero-fract-def order-less-le eq-fract)
      moreover from neq have 0 < ?F
        by (auto simp add: zero-less-mult-iff)
      moreover from neq le have (a * d) * (b * d) < (c * b) * (b * d)
        by simp
     ultimately have (a * d) * (b * d) * ?E * ?F < (c * b) * (b * d) * ?E * ?F
        by (simp add: mult-less-cancel-right)
      with neq show ?thesis
        by (simp add: mult-ac)
    qed
  qed
qed

lemma fract-induct-pos [case-names Fract]:
  fixes P :: 'a::linordered-idom fract ⇒ bool
  assumes step: ⋀a b. 0 < b ⟹ P (Fract a b)
  shows P q
proof (cases q)
  have step': ⋀a b. b < 0 ⟹ P (Fract a b)
  proof −
    fix a::'a and b::'a
    assume b: b < 0
```

```
    hence 0 < −b by simp
    hence P (Fract (−a) (−b)) by (rule step)
    thus P (Fract a b) by (simp add: order-less-imp-not-eq [OF b])
  qed
  case (Fract a b)
  thus P q by (force simp add: linorder-neq-iff step step')
qed

lemma zero-less-Fract-iff:
  0 < b ⟹ 0 < Fract a b ⟷ 0 < a
  by (auto simp add: Zero-fract-def zero-less-mult-iff)

lemma Fract-less-zero-iff:
  0 < b ⟹ Fract a b < 0 ⟷ a < 0
  by (auto simp add: Zero-fract-def mult-less-0-iff)

lemma zero-le-Fract-iff:
  0 < b ⟹ 0 ≤ Fract a b ⟷ 0 ≤ a
  by (auto simp add: Zero-fract-def zero-le-mult-iff)

lemma Fract-le-zero-iff:
  0 < b ⟹ Fract a b ≤ 0 ⟷ a ≤ 0
  by (auto simp add: Zero-fract-def mult-le-0-iff)

lemma one-less-Fract-iff:
  0 < b ⟹ 1 < Fract a b ⟷ b < a
  by (auto simp add: One-fract-def mult-less-cancel-right-disj)

lemma Fract-less-one-iff:
  0 < b ⟹ Fract a b < 1 ⟷ a < b
  by (auto simp add: One-fract-def mult-less-cancel-right-disj)

lemma one-le-Fract-iff:
  0 < b ⟹ 1 ≤ Fract a b ⟷ b ≤ a
  by (auto simp add: One-fract-def mult-le-cancel-right)

lemma Fract-le-one-iff:
  0 < b ⟹ Fract a b ≤ 1 ⟷ a ≤ b
  by (auto simp add: One-fract-def mult-le-cancel-right)

end
```

41 FuncSet: Pi and Function Sets

```
theory FuncSet
imports Hilbert-Choice Main
begin
```

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

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

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

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

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

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

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

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

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

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

41.1 Basic Properties of *Pi*

lemma *Pi-I*[*intro*!]: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B x*) ==> *f* ∈ *Pi A B*
by (*simp add*: *Pi-def*)

lemma *Pi-I'*[*simp*]: (!!*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 *PiE* [*elim*]:
f : *Pi A B* ==> (*f x* : *B x* ==> *Q*) ==> (*x* ~: *A* ==> *Q*) ==> *Q*
by(*auto simp*: *Pi-def*)

lemma *funcset-id* [*simp*]: (λ*x*. *x*) ∈ *A* → *A*
by (*auto intro*: *Pi-I*)

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*

lemma *Pi-eq-empty*[*simp*]: ((*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 *auto*

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: *A*′ <= *A* ==> *Pi A B* <= *Pi A*′ *B*
by *auto*

lemma *prod-final*:
assumes *1*: *fst* ∘ *f* ∈ *Pi A B* **and** *2*: *snd* ∘ *f* ∈ *Pi A C*
shows *f* ∈ (Π *z* ∈ *A*. *B z* × *C z*)
proof (*rule Pi-I*)
fix *z*
assume *z*: *z* ∈ *A*
have *f z* = (*fst* (*f z*), *snd* (*f z*))
by *simp*

also have ... $\in B\ z \times C\ z$
by (*metis SigmaI PiE o-apply 1 2 z*)
finally show $f\ z \in B\ z \times C\ z$ **.**
qed

41.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*:
$[|\ f \in A \to B;\ g \in B \to C\ |] \Longrightarrow$ *compose* $A\ g\ f \in A \to C$
by (*simp add*: *Pi-def compose-def restrict-def*)

lemma *compose-assoc*:
$[|\ f \in A \to B;\ g \in B \to C;\ h \in C \to D\ |]$
$\Longrightarrow$ *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*: $x \in A \Longrightarrow$ *compose* $A\ g\ f\ x = g(f(x))$
by (*simp add*: *compose-def restrict-def*)

lemma *surj-compose*: $[|\ f\ `\ A = B;\ g\ `\ B = C\ |] \Longrightarrow$ *compose* $A\ g\ f\ `\ A = C$
by (*auto simp add*: *image-def compose-eq*)

41.3 Bounded Abstraction: *restrict*

lemma *restrict-in-funcset*: $(!!x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in A \to B$
by (*simp add*: *Pi-def restrict-def*)

lemma *restrictI*[*intro*!]: $(!!x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in Pi\ A\ B$
by (*simp add*: *Pi-def restrict-def*)

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

lemma *restrict-ext*:
$(!!x.\ x \in A \Longrightarrow f\ x = g\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) = (\lambda x{\in}A.\ g\ x)$
by (*simp add*: *expand-fun-eq 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*:
$[|f \in A \to B;\ f \in extensional\ A|] \Longrightarrow$ *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*:
$[|g \in A \to B;\ g \in extensional\ A|] \Longrightarrow$ *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*)

41.4 Bijections Between Sets

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

lemma *bij-betw-imp-funcset*: *bij-betw f A B* ⟹ *f* ∈ *A* → *B*
by (*auto simp add*: *bij-betw-def*)

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*)

41.5 Extensionality

lemma *extensional-arb*: [|*f* ∈ *extensional A*; *x*∉ *A*|] ==> *f x* = *undefined*
by (*simp add*: *extensional-def*)

lemma *restrict-extensional* [*simp*]: *restrict f A* ∈ *extensional A*
by (*simp add*: *restrict-def extensional-def*)

lemma *compose-extensional* [*simp*]: *compose A f g* ∈ *extensional A*
by (*simp add*: *compose-def*)

lemma *extensionalityI*:
[| *f* ∈ *extensional A*; *g* ∈ *extensional A*;
!!*x*. *x*∈*A* ==> *f x* = *g x* |] ==> *f* = *g*
by (*force simp add*: *expand-fun-eq extensional-def*)

lemma *inv-into-funcset*: *f* ' *A* = *B* ==> (λ*x*∈*B*. *inv-into A f x*) : *B* -> *A*
by (*unfold inv-into-def*) (*fast intro*: *someI2*)

lemma *compose-inv-into-id*:
bij-betw f A B ==> *compose A* (λ*y*∈*B*. *inv-into A f y*) *f* = (λ*x*∈*A*. *x*)
apply (*simp add*: *bij-betw-def compose-def*)
apply (*rule restrict-ext*, *auto*)
done

lemma *compose-id-inv-into*:
f ' A = B ==> compose B f ($\lambda y{\in}B.$ *inv-into A f y*) = ($\lambda x{\in}B.$ *x*)
apply (*simp add*: *compose-def*)
apply (*rule restrict-ext*)
apply (*simp add*: *f-inv-into-f*)
done

41.6 Cardinality

lemma *card-inj*: [|*f* $\in$ *A*$\rightarrow$*B*; *inj-on f A*; *finite B*|] ==> *card*(*A*) $\leq$ *card*(*B*)
by (*rule card-inj-on-le*) *auto*

lemma *card-bij*:
[|*f* $\in$ *A*$\rightarrow$*B*; *inj-on f A*;
g $\in$ *B*$\rightarrow$*A*; *inj-on g B*; *finite A*; *finite B*|] ==> *card*(*A*) = *card*(*B*)
by (*blast intro*: *card-inj order-antisym*)

end

42 Polynomial: Univariate Polynomials

theory *Polynomial*
imports *Main*
begin

42.1 Definition of type *poly*

typedef (*Poly*) *'a poly* = {*f*::*nat* $\Rightarrow$ *'a*::*zero*. $\exists\, n.\ \forall\, i{>}n.\ f\ i = 0$}
morphisms *coeff Abs-poly*
by *auto*

lemma *expand-poly-eq*: *p* = *q* $\longleftrightarrow$ ($\forall\, n.$ *coeff p n* = *coeff q n*)
by (*simp add*: *coeff-inject* [*symmetric*] *expand-fun-eq*)

lemma *poly-ext*: ($\bigwedge n.$ *coeff p n* = *coeff q n*) $\Longrightarrow$ *p* = *q*
by (*simp add*: *expand-poly-eq*)

42.2 Degree of a polynomial

definition
degree :: *'a*::*zero poly* $\Rightarrow$ *nat* **where**
degree p = (*LEAST n*. $\forall\, i{>}n.$ *coeff p i* = *0*)

lemma *coeff-eq-0*: *degree p* < *n* $\Longrightarrow$ *coeff p n* = *0*
proof −
have *coeff p* $\in$ *Poly*
by (*rule coeff*)
hence $\exists\, n.\ \forall\, i{>}n.$ *coeff p i* = *0*

unfolding *Poly-def* **by** *simp*
hence $\forall i{>}degree\ p.\ coeff\ p\ i = 0$
unfolding *degree-def* **by** (*rule LeastI-ex*)
moreover assume $degree\ p < n$
ultimately show *?thesis* **by** *simp*
qed

lemma *le-degree*: $coeff\ p\ n \neq 0 \Longrightarrow n \leq degree\ p$
by (*erule contrapos-np*, *rule coeff-eq-0*, *simp*)

lemma *degree-le*: $\forall i{>}n.\ coeff\ p\ i = 0 \Longrightarrow degree\ p \leq n$
unfolding *degree-def* **by** (*erule Least-le*)

lemma *less-degree-imp*: $n < degree\ p \Longrightarrow \exists i{>}n.\ coeff\ p\ i \neq 0$
unfolding *degree-def* **by** (*drule not-less-Least*, *simp*)

42.3 The zero polynomial

instantiation *poly* :: (*zero*) *zero*
begin

definition
zero-poly-def: $0 = Abs\text{-}poly\ (\lambda n.\ 0)$

instance ..
end

lemma *coeff-0* [*simp*]: $coeff\ 0\ n = 0$
unfolding *zero-poly-def*
by (*simp add*: *Abs-poly-inverse Poly-def*)

lemma *degree-0* [*simp*]: $degree\ 0 = 0$
by (*rule order-antisym* [*OF degree-le le0*]) *simp*

lemma *leading-coeff-neq-0*:
assumes $p \neq 0$ **shows** $coeff\ p\ (degree\ p) \neq 0$
proof (*cases degree p*)
case *0*
from ‹$p \neq 0$› **have** $\exists n.\ coeff\ p\ n \neq 0$
by (*simp add*: *expand-poly-eq*)
then obtain *n* **where** $coeff\ p\ n \neq 0$ **..**
hence $n \leq degree\ p$ **by** (*rule le-degree*)
with ‹$coeff\ p\ n \neq 0$› **and** ‹$degree\ p = 0$›
show $coeff\ p\ (degree\ p) \neq 0$ **by** *simp*
next
case (*Suc n*)
from ‹$degree\ p = Suc\ n$› **have** $n < degree\ p$ **by** *simp*
hence $\exists i{>}n.\ coeff\ p\ i \neq 0$ **by** (*rule less-degree-imp*)
then obtain *i* **where** $n < i$ **and** $coeff\ p\ i \neq 0$ **by** *fast*

from ‹*degree p = Suc n*› **and** ‹*n < i*› **have** *degree p ≤ i* **by** *simp*
also from ‹*coeff p i ≠ 0*› **have** *i ≤ degree p* **by** (*rule le-degree*)
finally have *degree p = i* **.**
with ‹*coeff p i ≠ 0*› **show** *coeff p (degree p) ≠ 0* **by** *simp*
qed

lemma *leading-coeff-0-iff* [*simp*]: *coeff p (degree p) = 0 ⟷ p = 0*
by (*cases p = 0*, *simp*, *simp add*: *leading-coeff-neq-0*)

42.4 List-style constructor for polynomials

definition
pCons :: *'a*::*zero* ⇒ *'a poly* ⇒ *'a poly*
where
[*code del*]: *pCons a p = Abs-poly* (*nat-case a* (*coeff p*))

syntax
-poly :: *args* ⇒ *'a poly* ([:(-):])

translations
[:*x*, *xs*:] == *CONST pCons x* [:*xs*:]
[:*x*:] == *CONST pCons x 0*
[:*x*:] <= *CONST pCons x* (*-constrain 0 t*)

lemma *Poly-nat-case*: *f ∈ Poly ⟹ nat-case a f ∈ Poly*
unfolding *Poly-def* **by** (*auto split*: *nat.split*)

lemma *coeff-pCons*:
coeff (*pCons a p*) = *nat-case a* (*coeff p*)
unfolding *pCons-def*
by (*simp add*: *Abs-poly-inverse Poly-nat-case coeff*)

lemma *coeff-pCons-0* [*simp*]: *coeff* (*pCons a p*) *0* = *a*
by (*simp add*: *coeff-pCons*)

lemma *coeff-pCons-Suc* [*simp*]: *coeff* (*pCons a p*) (*Suc n*) = *coeff p n*
by (*simp add*: *coeff-pCons*)

lemma *degree-pCons-le*: *degree* (*pCons a p*) ≤ *Suc* (*degree p*)
by (*rule degree-le*, *simp add*: *coeff-eq-0 coeff-pCons split*: *nat.split*)

lemma *degree-pCons-eq*:
p ≠ 0 ⟹ degree (*pCons a p*) = *Suc* (*degree p*)
apply (*rule order-antisym* [*OF degree-pCons-le*])
apply (*rule le-degree*, *simp*)
done

lemma *degree-pCons-0*: *degree* (*pCons a 0*) = *0*
apply (*rule order-antisym* [*OF - le0*])

apply (*rule degree-le*, *simp add*: *coeff-pCons split*: *nat.split*)
done

lemma *degree-pCons-eq-if* [*simp*]:
  degree (*pCons a p*) = (*if p = 0 then 0 else Suc* (*degree p*))
apply (*cases p = 0*, *simp-all*)
apply (*rule order-antisym* [*OF - le0*])
apply (*rule degree-le*, *simp add*: *coeff-pCons split*: *nat.split*)
apply (*rule order-antisym* [*OF degree-pCons-le*])
apply (*rule le-degree*, *simp*)
done

lemma *pCons-0-0* [*simp*]: *pCons 0 0 = 0*
by (*rule poly-ext*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *pCons-eq-iff* [*simp*]:
  pCons a p = pCons b q ⟷ *a = b* ∧ *p = q*
proof (*safe*)
  assume *pCons a p = pCons b q*
  then have *coeff* (*pCons a p*) *0 = coeff* (*pCons b q*) *0* **by** *simp*
  then show *a = b* **by** *simp*
next
  assume *pCons a p = pCons b q*
  then have ∀ *n*. *coeff* (*pCons a p*) (*Suc n*) =
              coeff (*pCons b q*) (*Suc n*) **by** *simp*
  then show *p = q* **by** (*simp add*: *expand-poly-eq*)
qed

lemma *pCons-eq-0-iff* [*simp*]: *pCons a p = 0* ⟷ *a = 0* ∧ *p = 0*
  using *pCons-eq-iff* [*of a p 0 0*] **by** *simp*

lemma *Poly-Suc*: *f* ∈ *Poly* ⟹ (λ*n*. *f* (*Suc n*)) ∈ *Poly*
  unfolding *Poly-def*
  by (*clarify*, *rule-tac x*=*n* **in** *exI*, *simp*)

lemma *pCons-cases* [*cases type*: *poly*]:
  obtains (*pCons*) *a q* **where** *p = pCons a q*
proof
  show *p = pCons* (*coeff p 0*) (*Abs-poly* (λ*n*. *coeff p* (*Suc n*)))
    by (*rule poly-ext*)
      (*simp add*: *Abs-poly-inverse Poly-Suc coeff coeff-pCons*
          split: *nat.split*)
qed

lemma *pCons-induct* [*case-names 0 pCons*, *induct type*: *poly*]:
  assumes *zero*: *P 0*
  assumes *pCons*: ⋀*a p*. *P p* ⟹ *P* (*pCons a p*)
  shows *P p*
proof (*induct p rule*: *measure-induct-rule* [**where** *f*=*degree*])

case (*less p*)
obtain *a q* **where** *p* = *pCons a q* **by** (*rule pCons-cases*)
have *P q*
proof (*cases q* = *0*)
case *True*
then show *P q* **by** (*simp add*: *zero*)
next
case *False*
then have *degree* (*pCons a q*) = *Suc* (*degree q*)
by (*rule degree-pCons-eq*)
then have *degree q* < *degree p*
using ‹*p* = *pCons a q*› **by** *simp*
then show *P q*
by (*rule less.hyps*)
qed
then have *P* (*pCons a q*)
by (*rule pCons*)
then show *?case*
using ‹*p* = *pCons a q*› **by** *simp*
qed

42.5 Recursion combinator for polynomials

function
poly-rec :: *'b* ⇒ (*'a*::*zero* ⇒ *'a poly* ⇒ *'b* ⇒ *'b*) ⇒ *'a poly* ⇒ *'b*
where
poly-rec-pCons-eq-if [*simp del*, *code del*]:
poly-rec z f (*pCons a p*) = *f a p* (*if p* = *0 then z else poly-rec z f p*)
by (*case-tac x*, *rename-tac q*, *case-tac q*, *auto*)

termination *poly-rec*
by (*relation measure* (*degree* ∘ *snd* ∘ *snd*), *simp*)
(*simp add*: *degree-pCons-eq*)

lemma *poly-rec-0*:
f 0 0 z = *z* ⟹ *poly-rec z f 0* = *z*
using *poly-rec-pCons-eq-if* [*of z f 0 0*] **by** *simp*

lemma *poly-rec-pCons*:
f 0 0 z = *z* ⟹ *poly-rec z f* (*pCons a p*) = *f a p* (*poly-rec z f p*)
by (*simp add*: *poly-rec-pCons-eq-if poly-rec-0*)

42.6 Monomials

definition
monom :: *'a* ⇒ *nat* ⇒ *'a*::*zero poly* **where**
monom a m = *Abs-poly* (λ*n*. *if m* = *n then a else 0*)

lemma *coeff-monom* [*simp*]: *coeff* (*monom a m*) *n* = (*if m*=*n then a else 0*)
unfolding *monom-def*

by (*subst Abs-poly-inverse*, *auto simp add*: *Poly-def*)

lemma *monom-0*: *monom a 0* = *pCons a 0*
by (*rule poly-ext*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *monom-Suc*: *monom a* (*Suc n*) = *pCons 0* (*monom a n*)
by (*rule poly-ext*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *monom-eq-0* [*simp*]: *monom 0 n* = *0*
by (*rule poly-ext*) *simp*

lemma *monom-eq-0-iff* [*simp*]: *monom a n* = *0* ⟷ *a* = *0*
by (*simp add*: *expand-poly-eq*)

lemma *monom-eq-iff* [*simp*]: *monom a n* = *monom b n* ⟷ *a* = *b*
by (*simp add*: *expand-poly-eq*)

lemma *degree-monom-le*: *degree* (*monom a n*) ≤ *n*
by (*rule degree-le*, *simp*)

lemma *degree-monom-eq*: *a* ≠ *0* ⟹ *degree* (*monom a n*) = *n*
apply (*rule order-antisym* [*OF degree-monom-le*])
apply (*rule le-degree*, *simp*)
done

42.7 Addition and subtraction

instantiation *poly* :: (*comm-monoid-add*) *comm-monoid-add*
begin

definition
plus-poly-def [*code del*]:
p + *q* = *Abs-poly* (λ*n*. *coeff p n* + *coeff q n*)

lemma *Poly-add*:
fixes *f g* :: *nat* ⇒ *'a*
shows ⟦*f* ∈ *Poly*; *g* ∈ *Poly*⟧ ⟹ (λ*n*. *f n* + *g n*) ∈ *Poly*
unfolding *Poly-def*
apply (*clarify*, *rename-tac m n*)
apply (*rule-tac x*=*max m n* **in** *exI*, *simp*)
done

lemma *coeff-add* [*simp*]:
coeff (*p* + *q*) *n* = *coeff p n* + *coeff q n*
unfolding *plus-poly-def*
by (*simp add*: *Abs-poly-inverse coeff Poly-add*)

instance proof
fix *p q r* :: *'a poly*

show $(p + q) + r = p + (q + r)$
by (*simp add*: *expand-poly-eq add-assoc*)
show $p + q = q + p$
by (*simp add*: *expand-poly-eq add-commute*)
show $0 + p = p$
by (*simp add*: *expand-poly-eq*)
qed

end

instance *poly* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add*
proof
fix $p\ q\ r$ :: $'a$ *poly*
assume $p + q = p + r$ **thus** $q = r$
by (*simp add*: *expand-poly-eq*)
qed

instantiation *poly* :: (*ab-group-add*) *ab-group-add*
begin

definition
uminus-poly-def [*code del*]:
$- p = Abs\text{-}poly\ (\lambda n.\ -\ coeff\ p\ n)$

definition
minus-poly-def [*code del*]:
$p - q = Abs\text{-}poly\ (\lambda n.\ coeff\ p\ n - coeff\ q\ n)$

lemma *Poly-minus*:
fixes f :: $nat \Rightarrow 'a$
shows $f \in Poly \Longrightarrow (\lambda n.\ -\ f\ n) \in Poly$
unfolding *Poly-def* **by** *simp*

lemma *Poly-diff*:
fixes $f\ g$ :: $nat \Rightarrow 'a$
shows $[\![f \in Poly;\ g \in Poly]\!] \Longrightarrow (\lambda n.\ f\ n - g\ n) \in Poly$
unfolding *diff-minus* **by** (*simp add*: *Poly-add Poly-minus*)

lemma *coeff-minus* [*simp*]: $coeff\ (-\ p)\ n = -\ coeff\ p\ n$
unfolding *uminus-poly-def*
by (*simp add*: *Abs-poly-inverse coeff Poly-minus*)

lemma *coeff-diff* [*simp*]:
$coeff\ (p - q)\ n = coeff\ p\ n - coeff\ q\ n$
unfolding *minus-poly-def*
by (*simp add*: *Abs-poly-inverse coeff Poly-diff*)

instance proof
fix $p\ q$ :: $'a$ *poly*

show $-\ p + p = 0$
by (*simp add*: *expand-poly-eq*)
show $p - q = p + - q$
by (*simp add*: *expand-poly-eq diff-minus*)
qed

end

lemma *add-pCons* [*simp*]:
$pCons\ a\ p + pCons\ b\ q = pCons\ (a + b)\ (p + q)$
by (*rule poly-ext*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *minus-pCons* [*simp*]:
$-\ pCons\ a\ p = pCons\ (-\ a)\ (-\ p)$
by (*rule poly-ext*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *diff-pCons* [*simp*]:
$pCons\ a\ p - pCons\ b\ q = pCons\ (a - b)\ (p - q)$
by (*rule poly-ext*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *degree-add-le-max*: $degree\ (p + q) \leq max\ (degree\ p)\ (degree\ q)$
by (*rule degree-le*, *auto simp add*: *coeff-eq-0*)

lemma *degree-add-le*:
$[\![degree\ p \leq n;\ degree\ q \leq n]\!] \Longrightarrow degree\ (p + q) \leq n$
by (*auto intro*: *order-trans degree-add-le-max*)

lemma *degree-add-less*:
$[\![degree\ p < n;\ degree\ q < n]\!] \Longrightarrow degree\ (p + q) < n$
by (*auto intro*: *le-less-trans degree-add-le-max*)

lemma *degree-add-eq-right*:
$degree\ p < degree\ q \Longrightarrow degree\ (p + q) = degree\ q$
apply (*cases* $q = 0$, *simp*)
apply (*rule order-antisym*)
apply (*simp add*: *degree-add-le*)
apply (*rule le-degree*)
apply (*simp add*: *coeff-eq-0*)
done

lemma *degree-add-eq-left*:
$degree\ q < degree\ p \Longrightarrow degree\ (p + q) = degree\ p$
using *degree-add-eq-right* [*of q p*]
by (*simp add*: *add-commute*)

lemma *degree-minus* [*simp*]: $degree\ (-\ p) = degree\ p$
unfolding *degree-def* **by** *simp*

lemma *degree-diff-le-max*: $degree\ (p - q) \leq max\ (degree\ p)\ (degree\ q)$

using *degree-add-le* [**where** $p{=}p$ **and** $q{=}{-}q$]
by (*simp add*: *diff-minus*)

lemma *degree-diff-le*:
⟦*degree* $p \leq n$; *degree* $q \leq n$⟧ $\Longrightarrow$ *degree* $(p - q) \leq n$
by (*simp add*: *diff-minus degree-add-le*)

lemma *degree-diff-less*:
⟦*degree* $p < n$; *degree* $q < n$⟧ $\Longrightarrow$ *degree* $(p - q) < n$
by (*simp add*: *diff-minus degree-add-less*)

lemma *add-monom*: *monom* $a\ n$ + *monom* $b\ n$ = *monom* $(a + b)\ n$
by (*rule poly-ext*) *simp*

lemma *diff-monom*: *monom* $a\ n$ − *monom* $b\ n$ = *monom* $(a - b)\ n$
by (*rule poly-ext*) *simp*

lemma *minus-monom*: − *monom* $a\ n$ = *monom* $(-a)\ n$
by (*rule poly-ext*) *simp*

lemma *coeff-setsum*: *coeff* $(\sum x{\in}A.\ p\ x)\ i = (\sum x{\in}A.\ coeff\ (p\ x)\ i)$
by (*cases finite A*, *induct set*: *finite*, *simp-all*)

lemma *monom-setsum*: *monom* $(\sum x{\in}A.\ a\ x)\ n = (\sum x{\in}A.\ monom\ (a\ x)\ n)$
by (*rule poly-ext*) (*simp add*: *coeff-setsum*)

42.8 Multiplication by a constant

definition
smult :: *'a*::*comm-semiring-0* ⇒ *'a poly* ⇒ *'a poly* **where**
smult a p = *Abs-poly* $(\lambda n.\ a * coeff\ p\ n)$

lemma *Poly-smult*:
fixes *f* :: *nat* ⇒ *'a*::*comm-semiring-0*
shows $f \in Poly \Longrightarrow (\lambda n.\ a * f\ n) \in Poly$
unfolding *Poly-def*
by (*clarify*, *rule-tac* $x{=}n$ **in** *exI*, *simp*)

lemma *coeff-smult* [*simp*]: *coeff* (*smult a p*) *n* = *a* * *coeff p n*
unfolding *smult-def*
by (*simp add*: *Abs-poly-inverse Poly-smult coeff*)

lemma *degree-smult-le*: *degree* (*smult a p*) ≤ *degree p*
by (*rule degree-le*, *simp add*: *coeff-eq-0*)

lemma *smult-smult* [*simp*]: *smult a* (*smult b p*) = *smult* (*a* * *b*) *p*
by (*rule poly-ext*, *simp add*: *mult-assoc*)

lemma *smult-0-right* [*simp*]: *smult a 0* = *0*

by (*rule poly-ext*, *simp*)

lemma *smult-0-left* [*simp*]: *smult 0 p = 0*
by (*rule poly-ext*, *simp*)

lemma *smult-1-left* [*simp*]: *smult (1::'a::comm-semiring-1) p = p*
by (*rule poly-ext*, *simp*)

lemma *smult-add-right*:
smult a (p + q) = smult a p + smult a q
by (*rule poly-ext*, *simp add*: *algebra-simps*)

lemma *smult-add-left*:
smult (a + b) p = smult a p + smult b p
by (*rule poly-ext*, *simp add*: *algebra-simps*)

lemma *smult-minus-right* [*simp*]:
smult (a::'a::comm-ring) (− p) = − smult a p
by (*rule poly-ext*, *simp*)

lemma *smult-minus-left* [*simp*]:
smult (− a::'a::comm-ring) p = − smult a p
by (*rule poly-ext*, *simp*)

lemma *smult-diff-right*:
smult (a::'a::comm-ring) (p − q) = smult a p − smult a q
by (*rule poly-ext*, *simp add*: *algebra-simps*)

lemma *smult-diff-left*:
smult (a − b::'a::comm-ring) p = smult a p − smult b p
by (*rule poly-ext*, *simp add*: *algebra-simps*)

lemmas *smult-distribs* =
smult-add-left smult-add-right
smult-diff-left smult-diff-right

lemma *smult-pCons* [*simp*]:
*smult a (pCons b p) = pCons (a * b) (smult a p)*
by (*rule poly-ext*, *simp add*: *coeff-pCons split*: *nat.split*)

lemma *smult-monom*: *smult a (monom b n) = monom (a * b) n*
by (*induct n*, *simp add*: *monom-0*, *simp add*: *monom-Suc*)

lemma *degree-smult-eq* [*simp*]:
fixes *a* :: *'a::idom*
shows *degree (smult a p) = (if a = 0 then 0 else degree p)*
by (*cases a = 0*, *simp*, *simp add*: *degree-def*)

lemma *smult-eq-0-iff* [*simp*]:

fixes $a :: 'a::idom$
shows $smult\ a\ p = 0 \longleftrightarrow a = 0 \vee p = 0$
by (*simp add*: *expand-poly-eq*)

42.9 Multiplication of polynomials

TODO: move to SetInterval.thy

lemma *setsum-atMost-Suc-shift*:
fixes $f :: nat \Rightarrow 'a::comm\text{-}monoid\text{-}add$
shows $(\sum i{\le}Suc\ n.\ f\ i) = f\ 0 + (\sum i{\le}n.\ f\ (Suc\ i))$
proof (*induct n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*) **note** *IH* = *this*
have $(\sum i{\le}Suc\ (Suc\ n).\ f\ i) = (\sum i{\le}Suc\ n.\ f\ i) + f\ (Suc\ (Suc\ n))$
by (*rule setsum-atMost-Suc*)
also have $(\sum i{\le}Suc\ n.\ f\ i) = f\ 0 + (\sum i{\le}n.\ f\ (Suc\ i))$
by (*rule IH*)
also have $f\ 0 + (\sum i{\le}n.\ f\ (Suc\ i)) + f\ (Suc\ (Suc\ n)) =$
$f\ 0 + ((\sum i{\le}n.\ f\ (Suc\ i)) + f\ (Suc\ (Suc\ n)))$
by (*rule add-assoc*)
also have $(\sum i{\le}n.\ f\ (Suc\ i)) + f\ (Suc\ (Suc\ n)) = (\sum i{\le}Suc\ n.\ f\ (Suc\ i))$
by (*rule setsum-atMost-Suc* [*symmetric*])
finally show *?case* **.**
qed

instantiation *poly* :: (*comm-semiring-0*) *comm-semiring-0*
begin

definition
times-poly-def [*code del*]:
$p * q = poly\text{-}rec\ 0\ (\lambda a\ p\ pq.\ smult\ a\ q + pCons\ 0\ pq)\ p$

lemma *mult-poly-0-left*: $(0::'a\ poly) * q = 0$
unfolding *times-poly-def* **by** (*simp add*: *poly-rec-0*)

lemma *mult-pCons-left* [*simp*]:
$pCons\ a\ p * q = smult\ a\ q + pCons\ 0\ (p * q)$
unfolding *times-poly-def* **by** (*simp add*: *poly-rec-pCons*)

lemma *mult-poly-0-right*: $p * (0::'a\ poly) = 0$
by (*induct p*, *simp add*: *mult-poly-0-left*, *simp*)

lemma *mult-pCons-right* [*simp*]:
$p * pCons\ a\ q = smult\ a\ p + pCons\ 0\ (p * q)$
by (*induct p*, *simp add*: *mult-poly-0-left*, *simp add*: *algebra-simps*)

lemmas *mult-poly-0* = *mult-poly-0-left mult-poly-0-right*

lemma *mult-smult-left* [*simp*]: *smult a p * q = smult a (p * q)*
by (*induct p*, *simp add*: *mult-poly-0*, *simp add*: *smult-add-right*)

lemma *mult-smult-right* [*simp*]: *p * smult a q = smult a (p * q)*
by (*induct q*, *simp add*: *mult-poly-0*, *simp add*: *smult-add-right*)

lemma *mult-poly-add-left*:
fixes *p q r* :: *'a poly*
shows *(p + q) * r = p * r + q * r*
by (*induct r*, *simp add*: *mult-poly-0*,
simp add: *smult-distribs algebra-simps*)

instance proof
fix *p q r* :: *'a poly*
show *0*: *0 * p = 0*
by (*rule mult-poly-0-left*)
show *p * 0 = 0*
by (*rule mult-poly-0-right*)
show *(p + q) * r = p * r + q * r*
by (*rule mult-poly-add-left*)
show *(p * q) * r = p * (q * r)*
by (*induct p*, *simp add*: *mult-poly-0*, *simp add*: *mult-poly-add-left*)
show *p * q = q * p*
by (*induct p*, *simp add*: *mult-poly-0*, *simp*)
qed

end

instance *poly* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel* **..**

lemma *coeff-mult*:
*coeff (p * q) n = (∑ i≤n. coeff p i * coeff q (n−i))*
proof (*induct p arbitrary*: *n*)
case *0* **show** *?case* **by** *simp*
next
case (*pCons a p n*) **thus** *?case*
by (*cases n*, *simp*, *simp add*: *setsum-atMost-Suc-shift*
del: *setsum-atMost-Suc*)
qed

lemma *degree-mult-le*: *degree (p * q) ≤ degree p + degree q*
apply (*rule degree-le*)
apply (*induct p*)
apply *simp*
apply (*simp add*: *coeff-eq-0 coeff-pCons split*: *nat.split*)
done

lemma *mult-monom*: *monom a m * monom b n = monom (a * b) (m + n)*
by (*induct m*, *simp add*: *monom-0 smult-monom*, *simp add*: *monom-Suc*)

42.10 The unit polynomial and exponentiation

instantiation *poly* :: (*comm-semiring-1*) *comm-semiring-1*
begin

definition
one-poly-def:
1 = pCons 1 0

instance proof
fix *p* :: *'a poly* **show** *1 * p = p*
unfolding *one-poly-def*
by *simp*
next
show *0 ≠ (1::'a poly)*
unfolding *one-poly-def* **by** *simp*
qed

end

instance *poly* :: (*comm-semiring-1-cancel*) *comm-semiring-1-cancel* **..**

lemma *coeff-1* [*simp*]: *coeff 1 n = (if n = 0 then 1 else 0)*
unfolding *one-poly-def*
by (*simp add*: *coeff-pCons split*: *nat.split*)

lemma *degree-1* [*simp*]: *degree 1 = 0*
unfolding *one-poly-def*
by (*rule degree-pCons-0*)

Lemmas about divisibility

lemma *dvd-smult*: *p dvd q ⟹ p dvd smult a q*
proof −
assume *p dvd q*
then obtain *k* **where** *q = p * k* **..**
then have *smult a q = p * smult a k* **by** *simp*
then show *p dvd smult a q* **..**
qed

lemma *dvd-smult-cancel*:
fixes *a* :: *'a*::*field*
shows *p dvd smult a q ⟹ a ≠ 0 ⟹ p dvd q*
by (*drule dvd-smult* [**where** *a*=*inverse a*]) *simp*

lemma *dvd-smult-iff*:
fixes *a* :: *'a*::*field*
shows *a ≠ 0 ⟹ p dvd smult a q ⟷ p dvd q*
by (*safe elim*!: *dvd-smult dvd-smult-cancel*)

lemma *smult-dvd-cancel*:

```
  smult a p dvd q ⟹ p dvd q
proof −
  assume smult a p dvd q
  then obtain k where q = smult a p ∗ k ..
  then have q = p ∗ smult a k by simp
  then show p dvd q ..
qed

lemma smult-dvd:
  fixes a :: 'a::field
  shows p dvd q ⟹ a ≠ 0 ⟹ smult a p dvd q
  by (rule smult-dvd-cancel [where a=inverse a]) simp

lemma smult-dvd-iff:
  fixes a :: 'a::field
  shows smult a p dvd q ⟷ (if a = 0 then q = 0 else p dvd q)
  by (auto elim: smult-dvd smult-dvd-cancel)

lemma degree-power-le: degree (p ^ n) ≤ degree p ∗ n
by (induct n, simp, auto intro: order-trans degree-mult-le)

instance poly :: (comm-ring) comm-ring ..

instance poly :: (comm-ring-1) comm-ring-1 ..

instantiation poly :: (comm-ring-1) number-ring
begin

definition
  number-of k = (of-int k :: 'a poly)

instance
  by default (rule number-of-poly-def)

end
```

42.11 Polynomials form an integral domain

```
lemma coeff-mult-degree-sum:
  coeff (p ∗ q) (degree p + degree q) =
   coeff p (degree p) ∗ coeff q (degree q)
  by (induct p, simp, simp add: coeff-eq-0)

instance poly :: (idom) idom
proof
  fix p q :: 'a poly
  assume p ≠ 0 and q ≠ 0
  have coeff (p ∗ q) (degree p + degree q) =
        coeff p (degree p) ∗ coeff q (degree q)
```

by (*rule coeff-mult-degree-sum*)
also have *coeff p* (*degree p*) ∗ *coeff q* (*degree q*) ≠ *0*
using ‹*p* ≠ *0*› **and** ‹*q* ≠ *0*› **by** *simp*
finally have ∃ *n*. *coeff* (*p* ∗ *q*) *n* ≠ *0* **..**
thus *p* ∗ *q* ≠ *0* **by** (*simp add*: *expand-poly-eq*)
qed

lemma *degree-mult-eq*:
fixes *p q* :: *'a*::*idom poly*
shows ⟦*p* ≠ *0*; *q* ≠ *0*⟧ ⟹ *degree* (*p* ∗ *q*) = *degree p* + *degree q*
apply (*rule order-antisym* [*OF degree-mult-le le-degree*])
apply (*simp add*: *coeff-mult-degree-sum*)
done

lemma *dvd-imp-degree-le*:
fixes *p q* :: *'a*::*idom poly*
shows ⟦*p dvd q*; *q* ≠ *0*⟧ ⟹ *degree p* ≤ *degree q*
by (*erule dvdE*, *simp add*: *degree-mult-eq*)

42.12 Polynomials form an ordered integral domain

definition
pos-poly :: *'a*::*linordered-idom poly* ⇒ *bool*
where
pos-poly p ⟷ *0* < *coeff p* (*degree p*)

lemma *pos-poly-pCons*:
pos-poly (*pCons a p*) ⟷ *pos-poly p* ∨ (*p* = *0* ∧ *0* < *a*)
unfolding *pos-poly-def* **by** *simp*

lemma *not-pos-poly-0* [*simp*]: ¬ *pos-poly 0*
unfolding *pos-poly-def* **by** *simp*

lemma *pos-poly-add*: ⟦*pos-poly p*; *pos-poly q*⟧ ⟹ *pos-poly* (*p* + *q*)
apply (*induct p arbitrary*: *q*, *simp*)
apply (*case-tac q*, *force simp add*: *pos-poly-pCons add-pos-pos*)
done

lemma *pos-poly-mult*: ⟦*pos-poly p*; *pos-poly q*⟧ ⟹ *pos-poly* (*p* ∗ *q*)
unfolding *pos-poly-def*
apply (*subgoal-tac p* ≠ *0* ∧ *q* ≠ *0*)
apply (*simp add*: *degree-mult-eq coeff-mult-degree-sum mult-pos-pos*)
apply *auto*
done

lemma *pos-poly-total*: *p* = *0* ∨ *pos-poly p* ∨ *pos-poly* (− *p*)
by (*induct p*) (*auto simp add*: *pos-poly-pCons*)

instantiation *poly* :: (*linordered-idom*) *linordered-idom*

begin

definition
[*code del*]:
$x < y \longleftrightarrow$ *pos-poly* $(y - x)$

definition
[*code del*]:
$x \leq y \longleftrightarrow x = y \vee$ *pos-poly* $(y - x)$

definition
[*code del*]:
abs $(x::'a\ poly) =$ (*if* $x < 0$ *then* $-x$ *else* x)

definition
[*code del*]:
sgn $(x::'a\ poly) =$ (*if* $x = 0$ *then* 0 *else if* $0 < x$ *then* 1 *else* -1)

instance proof
fix $x\ y :: 'a\ poly$
show $x < y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$
unfolding *less-eq-poly-def less-poly-def*
apply *safe*
apply *simp*
apply (*drule* (*1*) *pos-poly-add*)
apply *simp*
done
next
fix $x :: 'a\ poly$ **show** $x \leq x$
unfolding *less-eq-poly-def* **by** *simp*
next
fix $x\ y\ z :: 'a\ poly$
assume $x \leq y$ **and** $y \leq z$ **thus** $x \leq z$
unfolding *less-eq-poly-def*
apply *safe*
apply (*drule* (*1*) *pos-poly-add*)
apply (*simp add*: *algebra-simps*)
done
next
fix $x\ y :: 'a\ poly$
assume $x \leq y$ **and** $y \leq x$ **thus** $x = y$
unfolding *less-eq-poly-def*
apply *safe*
apply (*drule* (*1*) *pos-poly-add*)
apply *simp*
done
next
fix $x\ y\ z :: 'a\ poly$
assume $x \leq y$ **thus** $z + x \leq z + y$

unfolding *less-eq-poly-def*
apply *safe*
apply (*simp add*: *algebra-simps*)
done
next
fix *x y* :: *'a poly*
show $x \leq y \vee y \leq x$
unfolding *less-eq-poly-def*
using *pos-poly-total* [*of* $x - y$]
by *auto*
next
fix *x y z* :: *'a poly*
assume $x < y$ **and** $0 < z$
thus $z * x < z * y$
unfolding *less-poly-def*
by (*simp add*: *right-diff-distrib* [*symmetric*] *pos-poly-mult*)
next
fix *x* :: *'a poly*
show $|x| = (\textit{if } x < 0 \textit{ then } -x \textit{ else } x)$
by (*rule abs-poly-def*)
next
fix *x* :: *'a poly*
show $\textit{sgn } x = (\textit{if } x = 0 \textit{ then } 0 \textit{ else if } 0 < x \textit{ then } 1 \textit{ else } -1)$
by (*rule sgn-poly-def*)
qed

end

TODO: Simplification rules for comparisons

42.13 Long division of polynomials

definition
pdivmod-rel :: *'a*::*field poly* ⇒ *'a poly* ⇒ *'a poly* ⇒ *'a poly* ⇒ *bool*
where
[*code del*]:
pdivmod-rel x y q r ⟷
$x = q * y + r \wedge (\textit{if } y = 0 \textit{ then } q = 0 \textit{ else } r = 0 \vee \textit{degree } r < \textit{degree } y)$

lemma *pdivmod-rel-0*:
pdivmod-rel 0 y 0 0
unfolding *pdivmod-rel-def* **by** *simp*

lemma *pdivmod-rel-by-0*:
pdivmod-rel x 0 0 x
unfolding *pdivmod-rel-def* **by** *simp*

lemma *eq-zero-or-degree-less*:
assumes *degree p* ≤ *n* **and** *coeff p n* = *0*
shows $p = 0 \vee \textit{degree } p < n$

```
proof (cases n)
  case 0
  with ‹degree p ≤ n› and ‹coeff p n = 0›
  have coeff p (degree p) = 0 by simp
  then have p = 0 by simp
  then show ?thesis ..
next
  case (Suc m)
  have ∀ i>n. coeff p i = 0
    using ‹degree p ≤ n› by (simp add: coeff-eq-0)
  then have ∀ i≥n. coeff p i = 0
    using ‹coeff p n = 0› by (simp add: le-less)
  then have ∀ i>m. coeff p i = 0
    using ‹n = Suc m› by (simp add: less-eq-Suc-le)
  then have degree p ≤ m
    by (rule degree-le)
  then have degree p < n
    using ‹n = Suc m› by (simp add: less-Suc-eq-le)
  then show ?thesis ..
qed

lemma pdivmod-rel-pCons:
  assumes rel: pdivmod-rel x y q r
  assumes y: y ≠ 0
  assumes b: b = coeff (pCons a r) (degree y) / coeff y (degree y)
  shows pdivmod-rel (pCons a x) y (pCons b q) (pCons a r − smult b y)
    (is pdivmod-rel ?x y ?q ?r)
proof −
  have x: x = q * y + r and r: r = 0 ∨ degree r < degree y
    using assms unfolding pdivmod-rel-def by simp-all

  have 1: ?x = ?q * y + ?r
    using b x by simp

  have 2: ?r = 0 ∨ degree ?r < degree y
  proof (rule eq-zero-or-degree-less)
    show degree ?r ≤ degree y
    proof (rule degree-diff-le)
      show degree (pCons a r) ≤ degree y
        using r by auto
      show degree (smult b y) ≤ degree y
        by (rule degree-smult-le)
    qed
  next
    show coeff ?r (degree y) = 0
      using ‹y ≠ 0› unfolding b by simp
  qed

  from 1 2 show ?thesis
```

```
    unfolding pdivmod-rel-def
    using ‹y ≠ 0› by simp
qed

lemma pdivmod-rel-exists: ∃ q r. pdivmod-rel x y q r
apply (cases y = 0)
apply (fast intro!: pdivmod-rel-by-0)
apply (induct x)
apply (fast intro!: pdivmod-rel-0)
apply (fast intro!: pdivmod-rel-pCons)
done

lemma pdivmod-rel-unique:
  assumes 1: pdivmod-rel x y q1 r1
  assumes 2: pdivmod-rel x y q2 r2
  shows q1 = q2 ∧ r1 = r2
proof (cases y = 0)
  assume y = 0 with assms show ?thesis
    by (simp add: pdivmod-rel-def)
next
  assume [simp]: y ≠ 0
  from 1 have q1: x = q1 * y + r1 and r1: r1 = 0 ∨ degree r1 < degree y
    unfolding pdivmod-rel-def by simp-all
  from 2 have q2: x = q2 * y + r2 and r2: r2 = 0 ∨ degree r2 < degree y
    unfolding pdivmod-rel-def by simp-all
  from q1 q2 have q3: (q1 - q2) * y = r2 - r1
    by (simp add: algebra-simps)
  from r1 r2 have r3: (r2 - r1) = 0 ∨ degree (r2 - r1) < degree y
    by (auto intro: degree-diff-less)

  show q1 = q2 ∧ r1 = r2
  proof (rule ccontr)
    assume ¬ (q1 = q2 ∧ r1 = r2)
    with q3 have q1 ≠ q2 and r1 ≠ r2 by auto
    with r3 have degree (r2 - r1) < degree y by simp
    also have degree y ≤ degree (q1 - q2) + degree y by simp
    also have ... = degree ((q1 - q2) * y)
      using ‹q1 ≠ q2› by (simp add: degree-mult-eq)
    also have ... = degree (r2 - r1)
      using q3 by simp
    finally have degree (r2 - r1) < degree (r2 - r1) .
    then show False by simp
  qed
qed

lemma pdivmod-rel-0-iff: pdivmod-rel 0 y q r ⟷ q = 0 ∧ r = 0
by (auto dest: pdivmod-rel-unique intro: pdivmod-rel-0)

lemma pdivmod-rel-by-0-iff: pdivmod-rel x 0 q r ⟷ q = 0 ∧ r = x
```

by (*auto dest*: *pdivmod-rel-unique intro*: *pdivmod-rel-by-0*)

lemmas *pdivmod-rel-unique-div* =
pdivmod-rel-unique [*THEN conjunct1*, *standard*]

lemmas *pdivmod-rel-unique-mod* =
pdivmod-rel-unique [*THEN conjunct2*, *standard*]

instantiation *poly* :: (*field*) *ring-div*
begin

definition *div-poly* **where**
[*code del*]: *x div y* = (*THE q*. ∃ *r*. *pdivmod-rel x y q r*)

definition *mod-poly* **where**
[*code del*]: *x mod y* = (*THE r*. ∃ *q*. *pdivmod-rel x y q r*)

lemma *div-poly-eq*:
pdivmod-rel x y q r ⟹ *x div y* = *q*
unfolding *div-poly-def*
by (*fast elim*: *pdivmod-rel-unique-div*)

lemma *mod-poly-eq*:
pdivmod-rel x y q r ⟹ *x mod y* = *r*
unfolding *mod-poly-def*
by (*fast elim*: *pdivmod-rel-unique-mod*)

lemma *pdivmod-rel*:
pdivmod-rel x y (*x div y*) (*x mod y*)
proof −
from *pdivmod-rel-exists*
obtain *q r* **where** *pdivmod-rel x y q r* **by** *fast*
thus *?thesis*
by (*simp add*: *div-poly-eq mod-poly-eq*)
qed

instance proof
fix *x y* :: *'a poly*
show *x div y* ∗ *y* + *x mod y* = *x*
using *pdivmod-rel* [*of x y*]
by (*simp add*: *pdivmod-rel-def*)
next
fix *x* :: *'a poly*
have *pdivmod-rel x 0 0 x*
by (*rule pdivmod-rel-by-0*)
thus *x div 0* = *0*
by (*rule div-poly-eq*)
next
fix *y* :: *'a poly*

```
  have pdivmod-rel 0 y 0 0
    by (rule pdivmod-rel-0)
  thus 0 div y = 0
    by (rule div-poly-eq)
next
  fix x y z :: 'a poly
  assume y ≠ 0
  hence pdivmod-rel (x + z * y) y (z + x div y) (x mod y)
    using pdivmod-rel [of x y]
    by (simp add: pdivmod-rel-def left-distrib)
  thus (x + z * y) div y = z + x div y
    by (rule div-poly-eq)
next
  fix x y z :: 'a poly
  assume x ≠ 0
  show (x * y) div (x * z) = y div z
  proof (cases y ≠ 0 ∧ z ≠ 0)
    have ⋀x::'a poly. pdivmod-rel x 0 0 x
      by (rule pdivmod-rel-by-0)
    then have [simp]: ⋀x::'a poly. x div 0 = 0
      by (rule div-poly-eq)
    have ⋀x::'a poly. pdivmod-rel 0 x 0 0
      by (rule pdivmod-rel-0)
    then have [simp]: ⋀x::'a poly. 0 div x = 0
      by (rule div-poly-eq)
    case False then show ?thesis by auto
  next
    case True then have y ≠ 0 and z ≠ 0 by auto
    with ‹x ≠ 0›
    have ⋀q r. pdivmod-rel y z q r ⟹ pdivmod-rel (x * y) (x * z) q (x * r)
      by (auto simp add: pdivmod-rel-def algebra-simps)
        (rule classical, simp add: degree-mult-eq)
    moreover from pdivmod-rel have pdivmod-rel y z (y div z) (y mod z) .
    ultimately have pdivmod-rel (x * y) (x * z) (y div z) (x * (y mod z)) .
    then show ?thesis by (simp add: div-poly-eq)
  qed
qed

end

lemma degree-mod-less:
  y ≠ 0 ⟹ x mod y = 0 ∨ degree (x mod y) < degree y
  using pdivmod-rel [of x y]
  unfolding pdivmod-rel-def by simp

lemma div-poly-less: degree x < degree y ⟹ x div y = 0
proof −
  assume degree x < degree y
  hence pdivmod-rel x y 0 x
```

```
    by (simp add: pdivmod-rel-def)
  thus x div y = 0 by (rule div-poly-eq)
qed

lemma mod-poly-less: degree x < degree y ⟹ x mod y = x
proof −
  assume degree x < degree y
  hence pdivmod-rel x y 0 x
    by (simp add: pdivmod-rel-def)
  thus x mod y = x by (rule mod-poly-eq)
qed

lemma pdivmod-rel-smult-left:
  pdivmod-rel x y q r
    ⟹ pdivmod-rel (smult a x) y (smult a q) (smult a r)
  unfolding pdivmod-rel-def by (simp add: smult-add-right)

lemma div-smult-left: (smult a x) div y = smult a (x div y)
  by (rule div-poly-eq, rule pdivmod-rel-smult-left, rule pdivmod-rel)

lemma mod-smult-left: (smult a x) mod y = smult a (x mod y)
  by (rule mod-poly-eq, rule pdivmod-rel-smult-left, rule pdivmod-rel)

lemma poly-div-minus-left [simp]:
  fixes x y :: 'a::field poly
  shows (− x) div y = − (x div y)
  using div-smult-left [of − 1::'a] by simp

lemma poly-mod-minus-left [simp]:
  fixes x y :: 'a::field poly
  shows (− x) mod y = − (x mod y)
  using mod-smult-left [of − 1::'a] by simp

lemma pdivmod-rel-smult-right:
  ⟦a ≠ 0; pdivmod-rel x y q r⟧
    ⟹ pdivmod-rel x (smult a y) (smult (inverse a) q) r
  unfolding pdivmod-rel-def by simp

lemma div-smult-right:
  a ≠ 0 ⟹ x div (smult a y) = smult (inverse a) (x div y)
  by (rule div-poly-eq, erule pdivmod-rel-smult-right, rule pdivmod-rel)

lemma mod-smult-right: a ≠ 0 ⟹ x mod (smult a y) = x mod y
  by (rule mod-poly-eq, erule pdivmod-rel-smult-right, rule pdivmod-rel)

lemma poly-div-minus-right [simp]:
  fixes x y :: 'a::field poly
  shows x div (− y) = − (x div y)
  using div-smult-right [of − 1::'a]
```

by (*simp add*: *nonzero-inverse-minus-eq*)

lemma *poly-mod-minus-right* [*simp*]:
fixes *x y* :: *'a*::*field poly*
shows *x mod* (− *y*) = *x mod y*
using *mod-smult-right* [*of* − *1*::*'a*] **by** *simp*

lemma *pdivmod-rel-mult*:
⟦*pdivmod-rel x y q r*; *pdivmod-rel q z q' r'*⟧
⟹ *pdivmod-rel x* (*y* ∗ *z*) *q'* (*y* ∗ *r'* + *r*)
apply (*cases z* = *0*, *simp add*: *pdivmod-rel-def*)
apply (*cases y* = *0*, *simp add*: *pdivmod-rel-by-0-iff pdivmod-rel-0-iff*)
apply (*cases r* = *0*)
apply (*cases r'* = *0*)
apply (*simp add*: *pdivmod-rel-def*)
apply (*simp add*: *pdivmod-rel-def field-simps degree-mult-eq*)
apply (*cases r'* = *0*)
apply (*simp add*: *pdivmod-rel-def degree-mult-eq*)
apply (*simp add*: *pdivmod-rel-def field-simps*)
apply (*simp add*: *degree-mult-eq degree-add-less*)
done

lemma *poly-div-mult-right*:
fixes *x y z* :: *'a*::*field poly*
shows *x div* (*y* ∗ *z*) = (*x div y*) *div z*
by (*rule div-poly-eq*, *rule pdivmod-rel-mult*, (*rule pdivmod-rel*)+)

lemma *poly-mod-mult-right*:
fixes *x y z* :: *'a*::*field poly*
shows *x mod* (*y* ∗ *z*) = *y* ∗ (*x div y mod z*) + *x mod y*
by (*rule mod-poly-eq*, *rule pdivmod-rel-mult*, (*rule pdivmod-rel*)+)

lemma *mod-pCons*:
fixes *a* **and** *x*
assumes *y*: *y* ≠ *0*
defines *b*: *b* ≡ *coeff* (*pCons a* (*x mod y*)) (*degree y*) / *coeff y* (*degree y*)
shows (*pCons a x*) *mod y* = (*pCons a* (*x mod y*) − *smult b y*)
unfolding *b*
apply (*rule mod-poly-eq*)
apply (*rule pdivmod-rel-pCons* [*OF pdivmod-rel y refl*])
done

42.14 GCD of polynomials

function
poly-gcd :: *'a*::*field poly* ⇒ *'a poly* ⇒ *'a poly* **where**
poly-gcd x 0 = *smult* (*inverse* (*coeff x* (*degree x*))) *x*
| *y* ≠ *0* ⟹ *poly-gcd x y* = *poly-gcd y* (*x mod y*)
by *auto*

termination *poly-gcd*
by (*relation measure* (λ(*x*, *y*). *if y = 0 then 0 else Suc* (*degree y*)))
(*auto dest*: *degree-mod-less*)

declare *poly-gcd.simps* [*simp del*, *code del*]

lemma *poly-gcd-dvd1* [*iff*]: *poly-gcd x y dvd x*
and *poly-gcd-dvd2* [*iff*]: *poly-gcd x y dvd y*
apply (*induct x y rule*: *poly-gcd.induct*)
apply (*simp-all add*: *poly-gcd.simps*)
apply (*fastsimp simp add*: *smult-dvd-iff dest*: *inverse-zero-imp-zero*)
apply (*blast dest*: *dvd-mod-imp-dvd*)
done

lemma *poly-gcd-greatest*: *k dvd x* ⟹ *k dvd y* ⟹ *k dvd poly-gcd x y*
by (*induct x y rule*: *poly-gcd.induct*)
(*simp-all add*: *poly-gcd.simps dvd-mod dvd-smult*)

lemma *dvd-poly-gcd-iff* [*iff*]:
k dvd poly-gcd x y ⟷ *k dvd x* ∧ *k dvd y*
by (*blast intro*!: *poly-gcd-greatest intro*: *dvd-trans*)

lemma *poly-gcd-monic*:
coeff (*poly-gcd x y*) (*degree* (*poly-gcd x y*)) =
(*if x = 0* ∧ *y = 0 then 0 else 1*)
by (*induct x y rule*: *poly-gcd.induct*)
(*simp-all add*: *poly-gcd.simps nonzero-imp-inverse-nonzero*)

lemma *poly-gcd-zero-iff* [*simp*]:
poly-gcd x y = 0 ⟷ *x = 0* ∧ *y = 0*
by (*simp only*: *dvd-0-left-iff* [*symmetric*] *dvd-poly-gcd-iff*)

lemma *poly-gcd-0-0* [*simp*]: *poly-gcd 0 0 = 0*
by *simp*

lemma *poly-dvd-antisym*:
fixes *p q* :: *'a*::*idom poly*
assumes *coeff*: *coeff p* (*degree p*) = *coeff q* (*degree q*)
assumes *dvd1*: *p dvd q* **and** *dvd2*: *q dvd p* **shows** *p = q*
proof (*cases p = 0*)
case *True* **with** *coeff* **show** *p = q* **by** *simp*
next
case *False* **with** *coeff* **have** *q* ≠ *0* **by** *auto*
have *degree*: *degree p = degree q*
using ‹*p dvd q*› ‹*q dvd p*› ‹*p* ≠ *0*› ‹*q* ≠ *0*›
by (*intro order-antisym dvd-imp-degree-le*)

from ‹*p dvd q*› **obtain** *a* **where** *a*: *q = p* ∗ *a* **..**

with ‹$q \neq 0$› **have** $a \neq 0$ **by** *auto*
with *degree a* ‹$p \neq 0$› **have** *degree a* $= 0$
by (*simp add*: *degree-mult-eq*)
with *coeff a* **show** $p = q$
by (*cases a*, *auto split*: *if-splits*)
qed

lemma *poly-gcd-unique*:
assumes *dvd1*: *d dvd x* **and** *dvd2*: *d dvd y*
and *greatest*: $\bigwedge k.\ k\ dvd\ x \Longrightarrow k\ dvd\ y \Longrightarrow k\ dvd\ d$
and *monic*: *coeff d* (*degree d*) = (*if* $x = 0 \wedge y = 0$ *then 0 else 1*)
shows *poly-gcd x y* = *d*
proof −
have *coeff* (*poly-gcd x y*) (*degree* (*poly-gcd x y*)) = *coeff d* (*degree d*)
by (*simp-all add*: *poly-gcd-monic monic*)
moreover have *poly-gcd x y dvd d*
using *poly-gcd-dvd1 poly-gcd-dvd2* **by** (*rule greatest*)
moreover have *d dvd poly-gcd x y*
using *dvd1 dvd2* **by** (*rule poly-gcd-greatest*)
ultimately show *?thesis*
by (*rule poly-dvd-antisym*)
qed

interpretation *poly-gcd*!: *abel-semigroup poly-gcd*
proof
fix *x y z* :: *'a poly*
show *poly-gcd* (*poly-gcd x y*) *z* = *poly-gcd x* (*poly-gcd y z*)
by (*rule poly-gcd-unique*) (*auto intro*: *dvd-trans simp add*: *poly-gcd-monic*)
show *poly-gcd x y* = *poly-gcd y x*
by (*rule poly-gcd-unique*) (*simp-all add*: *poly-gcd-monic*)
qed

lemmas *poly-gcd-assoc* = *poly-gcd.assoc*
lemmas *poly-gcd-commute* = *poly-gcd.commute*
lemmas *poly-gcd-left-commute* = *poly-gcd.left-commute*

lemmas *poly-gcd-ac* = *poly-gcd-assoc poly-gcd-commute poly-gcd-left-commute*

lemma *poly-gcd-1-left* [*simp*]: *poly-gcd 1 y* = *1*
by (*rule poly-gcd-unique*) *simp-all*

lemma *poly-gcd-1-right* [*simp*]: *poly-gcd x 1* = *1*
by (*rule poly-gcd-unique*) *simp-all*

lemma *poly-gcd-minus-left* [*simp*]: *poly-gcd* (− *x*) *y* = *poly-gcd x y*
by (*rule poly-gcd-unique*) (*simp-all add*: *poly-gcd-monic*)

lemma *poly-gcd-minus-right* [*simp*]: *poly-gcd x* (− *y*) = *poly-gcd x y*
by (*rule poly-gcd-unique*) (*simp-all add*: *poly-gcd-monic*)

42.15 Evaluation of polynomials

definition
poly :: *'a*::*comm-semiring-0 poly* ⇒ *'a* ⇒ *'a* **where**
poly = *poly-rec* (λ*x*. *0*) (λ*a p f x*. *a* + *x* ∗ *f x*)

lemma *poly-0* [*simp*]: *poly 0 x* = *0*
unfolding *poly-def* **by** (*simp add*: *poly-rec-0*)

lemma *poly-pCons* [*simp*]: *poly* (*pCons a p*) *x* = *a* + *x* ∗ *poly p x*
unfolding *poly-def* **by** (*simp add*: *poly-rec-pCons*)

lemma *poly-1* [*simp*]: *poly 1 x* = *1*
unfolding *one-poly-def* **by** *simp*

lemma *poly-monom*:
fixes *a x* :: *'a*::{*comm-semiring-1*}
shows *poly* (*monom a n*) *x* = *a* ∗ *x* ^ *n*
by (*induct n*, *simp add*: *monom-0*, *simp add*: *monom-Suc power-Suc mult-ac*)

lemma *poly-add* [*simp*]: *poly* (*p* + *q*) *x* = *poly p x* + *poly q x*
apply (*induct p arbitrary*: *q*, *simp*)
apply (*case-tac q*, *simp*, *simp add*: *algebra-simps*)
done

lemma *poly-minus* [*simp*]:
fixes *x* :: *'a*::*comm-ring*
shows *poly* (− *p*) *x* = − *poly p x*
by (*induct p*, *simp-all*)

lemma *poly-diff* [*simp*]:
fixes *x* :: *'a*::*comm-ring*
shows *poly* (*p* − *q*) *x* = *poly p x* − *poly q x*
by (*simp add*: *diff-minus*)

lemma *poly-setsum*: *poly* (∑ *k*∈*A*. *p k*) *x* = (∑ *k*∈*A*. *poly* (*p k*) *x*)
by (*cases finite A*, *induct set*: *finite*, *simp-all*)

lemma *poly-smult* [*simp*]: *poly* (*smult a p*) *x* = *a* ∗ *poly p x*
by (*induct p*, *simp*, *simp add*: *algebra-simps*)

lemma *poly-mult* [*simp*]: *poly* (*p* ∗ *q*) *x* = *poly p x* ∗ *poly q x*
by (*induct p*, *simp-all*, *simp add*: *algebra-simps*)

lemma *poly-power* [*simp*]:
fixes *p* :: *'a*::{*comm-semiring-1*} *poly*
shows *poly* (*p* ^ *n*) *x* = *poly p x* ^ *n*
by (*induct n*, *simp*, *simp add*: *power-Suc*)

42.16 Synthetic division

Synthetic division is simply division by the linear polynomial $x - c$.

definition
synthetic-divmod :: *'a*::*comm-semiring-0 poly* ⇒ *'a* ⇒ *'a poly* × *'a*
where [*code del*]:
synthetic-divmod p c =
poly-rec (*0*, *0*) (λ*a p* (*q*, *r*). (*pCons r q*, *a* + *c* ∗ *r*)) *p*

definition
synthetic-div :: *'a*::*comm-semiring-0 poly* ⇒ *'a* ⇒ *'a poly*
where
synthetic-div p c = *fst* (*synthetic-divmod p c*)

lemma *synthetic-divmod-0* [*simp*]:
synthetic-divmod 0 c = (*0*, *0*)
unfolding *synthetic-divmod-def*
by (*simp add*: *poly-rec-0*)

lemma *synthetic-divmod-pCons* [*simp*]:
synthetic-divmod (*pCons a p*) *c* =
(λ(*q*, *r*). (*pCons r q*, *a* + *c* ∗ *r*)) (*synthetic-divmod p c*)
unfolding *synthetic-divmod-def*
by (*simp add*: *poly-rec-pCons*)

lemma *snd-synthetic-divmod*: *snd* (*synthetic-divmod p c*) = *poly p c*
by (*induct p*, *simp*, *simp add*: *split-def*)

lemma *synthetic-div-0* [*simp*]: *synthetic-div 0 c* = *0*
unfolding *synthetic-div-def* **by** *simp*

lemma *synthetic-div-pCons* [*simp*]:
synthetic-div (*pCons a p*) *c* = *pCons* (*poly p c*) (*synthetic-div p c*)
unfolding *synthetic-div-def*
by (*simp add*: *split-def snd-synthetic-divmod*)

lemma *synthetic-div-eq-0-iff*:
synthetic-div p c = *0* ⟷ *degree p* = *0*
by (*induct p*, *simp*, *case-tac p*, *simp*)

lemma *degree-synthetic-div*:
degree (*synthetic-div p c*) = *degree p* − *1*
by (*induct p*, *simp*, *simp add*: *synthetic-div-eq-0-iff*)

lemma *synthetic-div-correct*:
p + *smult c* (*synthetic-div p c*) = *pCons* (*poly p c*) (*synthetic-div p c*)
by (*induct p*) *simp-all*

lemma *synthetic-div-unique-lemma*: *smult c p* = *pCons a p* ⟹ *p* = *0*

by (*induct p arbitrary*: *a*) *simp-all*

lemma *synthetic-div-unique*:
p + *smult c q* = *pCons r q* ⟹ *r* = *poly p c* ∧ *q* = *synthetic-div p c*
apply (*induct p arbitrary*: *q r*)
apply (*simp*, *frule synthetic-div-unique-lemma*, *simp*)
apply (*case-tac q*, *force*)
done

lemma *synthetic-div-correct'*:
fixes *c* :: *'a*::*comm-ring-1*
shows [:−*c*, *1*:] ∗ *synthetic-div p c* + [:*poly p c*:] = *p*
using *synthetic-div-correct* [*of p c*]
by (*simp add*: *algebra-simps*)

lemma *poly-eq-0-iff-dvd*:
fixes *c* :: *'a*::*idom*
shows *poly p c* = *0* ⟷ [:−*c*, *1*:] *dvd p*
proof
assume *poly p c* = *0*
with *synthetic-div-correct'* [*of c p*]
have *p* = [:−*c*, *1*:] ∗ *synthetic-div p c* **by** *simp*
then show [:−*c*, *1*:] *dvd p* **..**
next
assume [:−*c*, *1*:] *dvd p*
then obtain *k* **where** *p* = [:−*c*, *1*:] ∗ *k* **by** (*rule dvdE*)
then show *poly p c* = *0* **by** *simp*
qed

lemma *dvd-iff-poly-eq-0*:
fixes *c* :: *'a*::*idom*
shows [:*c*, *1*:] *dvd p* ⟷ *poly p* (−*c*) = *0*
by (*simp add*: *poly-eq-0-iff-dvd*)

lemma *poly-roots-finite*:
fixes *p* :: *'a*::*idom poly*
shows *p* ≠ *0* ⟹ *finite* {*x*. *poly p x* = *0*}
proof (*induct n* ≡ *degree p arbitrary*: *p*)
case (*0 p*)
then obtain *a* **where** *a* ≠ *0* **and** *p* = [:*a*:]
by (*cases p*, *simp split*: *if-splits*)
then show *finite* {*x*. *poly p x* = *0*} **by** *simp*
next
case (*Suc n p*)
show *finite* {*x*. *poly p x* = *0*}
proof (*cases* ∃ *x*. *poly p x* = *0*)
case *False*
then show *finite* {*x*. *poly p x* = *0*} **by** *simp*
next

```
    case True
    then obtain a where poly p a = 0 ..
    then have [:-a, 1:] dvd p by (simp only: poly-eq-0-iff-dvd)
    then obtain k where k: p = [:-a, 1:] * k ..
    with ‹p ≠ 0› have k ≠ 0 by auto
    with k have degree p = Suc (degree k)
      by (simp add: degree-mult-eq del: mult-pCons-left)
    with ‹Suc n = degree p› have n = degree k by simp
    then have finite {x. poly k x = 0} using ‹k ≠ 0› by (rule Suc.hyps)
    then have finite (insert a {x. poly k x = 0}) by simp
    then show finite {x. poly p x = 0}
      by (simp add: k uminus-add-conv-diff Collect-disj-eq
            del: mult-pCons-left)
  qed
qed

lemma poly-zero:
  fixes p :: 'a::{idom,ring-char-0} poly
  shows poly p = poly 0 ⟷ p = 0
apply (cases p = 0, simp-all)
apply (drule poly-roots-finite)
apply (auto simp add: infinite-UNIV-char-0)
done

lemma poly-eq-iff:
  fixes p q :: 'a::{idom,ring-char-0} poly
  shows poly p = poly q ⟷ p = q
  using poly-zero [of p − q]
  by (simp add: expand-fun-eq)
```

42.17 Composition of polynomials

```
definition
  pcompose :: 'a::comm-semiring-0 poly ⇒ 'a poly ⇒ 'a poly
where
  pcompose p q = poly-rec 0 (λa - c. [:a:] + q * c) p

lemma pcompose-0 [simp]: pcompose 0 q = 0
  unfolding pcompose-def by (simp add: poly-rec-0)

lemma pcompose-pCons:
  pcompose (pCons a p) q = [:a:] + q * pcompose p q
  unfolding pcompose-def by (simp add: poly-rec-pCons)

lemma poly-pcompose: poly (pcompose p q) x = poly p (poly q x)
  by (induct p) (simp-all add: pcompose-pCons)

lemma degree-pcompose-le:
  degree (pcompose p q) ≤ degree p * degree q
```

apply (*induct p*, *simp*)
apply (*simp add*: *pcompose-pCons*, *clarify*)
apply (*rule degree-add-le*, *simp*)
apply (*rule order-trans* [*OF degree-mult-le*], *simp*)
done

42.18 Order of polynomial roots

definition
  order :: *'a*::*idom* ⇒ *'a poly* ⇒ *nat*
where
  [*code del*]:
  order a p = (*LEAST n*. ¬ [:−*a*, *1*:] ^ *Suc n dvd p*)

lemma *coeff-linear-power*:
  fixes *a* :: *'a*::*comm-semiring-1*
  shows *coeff* ([:*a*, *1*:] ^ *n*) *n* = *1*
apply (*induct n*, *simp-all*)
apply (*subst coeff-eq-0*)
apply (*auto intro*: *le-less-trans degree-power-le*)
done

lemma *degree-linear-power*:
  fixes *a* :: *'a*::*comm-semiring-1*
  shows *degree* ([:*a*, *1*:] ^ *n*) = *n*
apply (*rule order-antisym*)
apply (*rule ord-le-eq-trans* [*OF degree-power-le*], *simp*)
apply (*rule le-degree*, *simp add*: *coeff-linear-power*)
done

lemma *order-1*: [:−*a*, *1*:] ^ *order a p dvd p*
apply (*cases p* = *0*, *simp*)
apply (*cases order a p*, *simp*)
apply (*subgoal-tac nat* < (*LEAST n*. ¬ [:−*a*, *1*:] ^ *Suc n dvd p*))
apply (*drule not-less-Least*, *simp*)
apply (*fold order-def*, *simp*)
done

lemma *order-2*: *p* ≠ *0* ⟹ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
unfolding *order-def*
apply (*rule LeastI-ex*)
apply (*rule-tac x*=*degree p* **in** *exI*)
apply (*rule notI*)
apply (*drule* (*1*) *dvd-imp-degree-le*)
apply (*simp only*: *degree-linear-power*)
done

lemma *order*:
  p ≠ *0* ⟹ [:−*a*, *1*:] ^ *order a p dvd p* ∧ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*

by (*rule conjI* [*OF order-1 order-2*])

lemma *order-degree*:
  assumes *p*: $p \neq 0$
  shows *order a p* $\leq$ *degree p*
proof −
  have *order a p* = *degree* ([:−*a*, *1*:] ^ *order a p*)
    by (*simp only*: *degree-linear-power*)
  also have ... $\leq$ *degree p*
    using *order-1 p* **by** (*rule dvd-imp-degree-le*)
  finally show *?thesis* **.**
qed

lemma *order-root*: *poly p a* = *0* $\longleftrightarrow$ *p* = *0* $\vee$ *order a p* $\neq$ *0*
apply (*cases p* = *0*, *simp-all*)
apply (*rule iffI*)
apply (*rule ccontr*, *simp*)
apply (*frule order-2* [**where** *a*=*a*], *simp*)
apply (*simp add*: *poly-eq-0-iff-dvd*)
apply (*simp add*: *poly-eq-0-iff-dvd*)
apply (*simp only*: *order-def*)
apply (*drule not-less-Least*, *simp*)
done

42.19 Configuration of the code generator

code-datatype *0*::*'a*::*zero poly pCons*

declare *pCons-0-0* [*code-post*]

instantiation *poly* :: ({*zero*,*eq*}) *eq*
begin

definition [*code del*]:
  eq-class.eq (*p*::*'a poly*) *q* $\longleftrightarrow$ *p* = *q*

instance
  by *default* (*rule eq-poly-def*)

end

lemma *eq-poly-code* [*code*]:
  eq-class.eq (*0*::- *poly*) (*0*::- *poly*) $\longleftrightarrow$ *True*
  eq-class.eq (*0*::- *poly*) (*pCons b q*) $\longleftrightarrow$ *eq-class.eq 0 b* $\wedge$ *eq-class.eq 0 q*
  eq-class.eq (*pCons a p*) (*0*::- *poly*) $\longleftrightarrow$ *eq-class.eq a 0* $\wedge$ *eq-class.eq p 0*
  eq-class.eq (*pCons a p*) (*pCons b q*) $\longleftrightarrow$ *eq-class.eq a b* $\wedge$ *eq-class.eq p q*
unfolding *eq* **by** *simp-all*

lemmas *coeff-code* [*code*] =

coeff-0 coeff-pCons-0 coeff-pCons-Suc

lemmas *degree-code* [*code*] =
degree-0 degree-pCons-eq-if

lemmas *monom-poly-code* [*code*] =
monom-0 monom-Suc

lemma *add-poly-code* [*code*]:
$0 + q = (q :: - poly)$
$p + 0 = (p :: - poly)$
$pCons\ a\ p + pCons\ b\ q = pCons\ (a + b)\ (p + q)$
by *simp-all*

lemma *minus-poly-code* [*code*]:
$- 0 = (0 :: - poly)$
$- pCons\ a\ p = pCons\ (- a)\ (- p)$
by *simp-all*

lemma *diff-poly-code* [*code*]:
$0 - q = (- q :: - poly)$
$p - 0 = (p :: - poly)$
$pCons\ a\ p - pCons\ b\ q = pCons\ (a - b)\ (p - q)$
by *simp-all*

lemmas *smult-poly-code* [*code*] =
smult-0-right smult-pCons

lemma *mult-poly-code* [*code*]:
$0 * q = (0 :: - poly)$
$pCons\ a\ p * q = smult\ a\ q + pCons\ 0\ (p * q)$
by *simp-all*

lemmas *poly-code* [*code*] =
poly-0 poly-pCons

lemmas *synthetic-divmod-code* [*code*] =
synthetic-divmod-0 synthetic-divmod-pCons

code generator setup for div and mod

definition
$pdivmod :: 'a::field\ poly \Rightarrow 'a\ poly \Rightarrow 'a\ poly \times 'a\ poly$
where
[*code del*]: $pdivmod\ x\ y = (x\ div\ y,\ x\ mod\ y)$

lemma *div-poly-code* [*code*]: $x\ div\ y = fst\ (pdivmod\ x\ y)$
unfolding *pdivmod-def* **by** *simp*

lemma *mod-poly-code* [*code*]: $x\ mod\ y = snd\ (pdivmod\ x\ y)$

unfolding *pdivmod-def* **by** *simp*

lemma *pdivmod-0* [*code*]: *pdivmod 0 y* = (*0*, *0*)
unfolding *pdivmod-def* **by** *simp*

lemma *pdivmod-pCons* [*code*]:
pdivmod (*pCons a x*) *y* =
(*if y* = *0 then* (*0*, *pCons a x*) *else*
(*let* (*q*, *r*) = *pdivmod x y*;
b = *coeff* (*pCons a r*) (*degree y*) / *coeff y* (*degree y*)
in (*pCons b q*, *pCons a r* − *smult b y*)))
apply (*simp add*: *pdivmod-def Let-def*, *safe*)
apply (*rule div-poly-eq*)
apply (*erule pdivmod-rel-pCons* [*OF pdivmod-rel - refl*])
apply (*rule mod-poly-eq*)
apply (*erule pdivmod-rel-pCons* [*OF pdivmod-rel - refl*])
done

lemma *poly-gcd-code* [*code*]:
poly-gcd x y =
(*if y* = *0 then smult* (*inverse* (*coeff x* (*degree x*))) *x*
else poly-gcd y (*x mod y*))
by (*simp add*: *poly-gcd.simps*)

end

43 Fundamental-Theorem-Algebra: Fundamental Theorem of Algebra

theory *Fundamental-Theorem-Algebra*
imports *Polynomial Complex*
begin

43.1 Square root of complex numbers

definition *csqrt* :: *complex* ⇒ *complex* **where**
csqrt z = (*if Im z* = *0 then*
if 0 ≤ *Re z then Complex* (*sqrt*(*Re z*)) *0*
else Complex 0 (*sqrt*(− *Re z*))
else Complex (*sqrt*((*cmod z* + *Re z*) /*2*))
((*Im z* / *abs*(*Im z*)) ∗ *sqrt*((*cmod z* − *Re z*) /*2*)))

lemma *csqrt*[*algebra*]: *csqrt z* ^ *2* = *z*
proof−
obtain *x y* **where** *xy*: *z* = *Complex x y* **by** (*cases z*)
{assume *y0*: *y* = *0*
{assume *x0*: *x* ≥ *0*

then have *?thesis* **using** *y0 xy real-sqrt-pow2*[*OF x0*]
by (*simp add*: *csqrt-def power2-eq-square*)**}**
moreover
{assume ¬ *x* ≥ *0* **hence** *x0*: − *x* ≥ *0* **by** *arith*
then have *?thesis* **using** *y0 xy real-sqrt-pow2*[*OF x0*]
by (*simp add*: *csqrt-def power2-eq-square*) **}**
ultimately have *?thesis* **by** *blast***}**
moreover
{assume *y0*: *y*≠*0*
{fix *x y*
let *?z* = *Complex x y*
from *abs-Re-le-cmod*[*of ?z*] **have** *tha*: *abs x* ≤ *cmod ?z* **by** *auto*
hence *cmod ?z* − *x* ≥ *0 cmod ?z* + *x* ≥ *0* **by** *arith*+
hence (*sqrt* (*x* ∗ *x* + *y* ∗ *y*) + *x*) / *2* ≥ *0* (*sqrt* (*x* ∗ *x* + *y* ∗ *y*) − *x*) / *2* ≥ *0* **by** (*simp-all add*: *power2-eq-square*) **}**
note *th* = *this*
have *sq4*: ⋀*x*::*real*. *x*^*2* / *4* = (*x* / *2*) ^ *2*
by (*simp add*: *power2-eq-square*)
from *th*[*of x y*]
have *sq4'*: *sqrt* (((*sqrt* (*x* ∗ *x* + *y* ∗ *y*) + *x*)^*2* / *4*)) = (*sqrt* (*x* ∗ *x* + *y* ∗ *y*) + *x*) / *2 sqrt* (((*sqrt* (*x* ∗ *x* + *y* ∗ *y*) − *x*)^*2* / *4*)) = (*sqrt* (*x* ∗ *x* + *y* ∗ *y*) − *x*) / *2* **unfolding** *sq4* **by** *simp-all*
then have *th1*: *sqrt* ((*sqrt* (*x* ∗ *x* + *y* ∗ *y*) + *x*) ∗ (*sqrt* (*x* ∗ *x* + *y* ∗ *y*) + *x*) / *4*) − *sqrt* ((*sqrt* (*x* ∗ *x* + *y* ∗ *y*) − *x*) ∗ (*sqrt* (*x* ∗ *x* + *y* ∗ *y*) − *x*) / *4*) = *x*
unfolding *power2-eq-square* **by** *simp*
have *sqrt 4* = *sqrt* (*2*^*2*) **by** *simp*
hence *sqrt4*: *sqrt 4* = *2* **by** (*simp only*: *real-sqrt-abs*)
have *th2*: *2* ∗(*y* ∗ *sqrt* ((*sqrt* (*x* ∗ *x* + *y* ∗ *y*) − *x*) ∗ (*sqrt* (*x* ∗ *x* + *y* ∗ *y*) + *x*) / *4*)) / |*y*| = *y*
using *iffD2*[*OF real-sqrt-pow2-iff sum-power2-ge-zero*[*of x y*]] *y0*
unfolding *power2-eq-square*
by (*simp add*: *algebra-simps real-sqrt-divide sqrt4*)
from *y0 xy* **have** *?thesis* **apply** (*simp add*: *csqrt-def power2-eq-square*)
apply (*simp add*: *real-sqrt-sum-squares-mult-ge-zero*[*of x y*] *real-sqrt-pow2*[*OF th*(*1*)[*of x y*], *unfolded power2-eq-square*] *real-sqrt-pow2*[*OF th*(*2*)[*of x y*], *unfolded power2-eq-square*] *real-sqrt-mult*[*symmetric*])
using *th1 th2* **..}**
ultimately show *?thesis* **by** *blast*
qed

43.2 More lemmas about module of complex numbers

lemma *complex-of-real-power*: *complex-of-real x* ^ *n* = *complex-of-real* (*x*^*n*)
by (*rule of-real-power* [*symmetric*])

lemma *real-down2*: (*0*::*real*) < *d1* ⟹ *0* < *d2* ==> *EX e*. *0* < *e* & *e* < *d1* & *e* < *d2*
apply (*rule exI*[**where** *x* = *min d1 d2* / *2*])
by (*simp add*: *field-simps min-def*)

The triangle inequality for cmod

lemma *complex-mod-triangle-sub*: *cmod w* ≤ *cmod* (*w* + *z*) + *norm z*
using *complex-mod-triangle-ineq2*[*of w* + *z* −*z*] **by** *auto*

43.3 Basic lemmas about complex polynomials

lemma *poly-bound-exists*:
shows ∃ *m*. *m* > *0* ∧ (∀ *z*. *cmod z* <= *r* ⟶ *cmod* (*poly p z*) ≤ *m*)
proof(*induct p*)
case *0* **thus** *?case* **by** (*rule exI*[**where** *x*=*1*], *simp*)
next
case (*pCons c cs*)
from *pCons.hyps* **obtain** *m* **where** *m*: ∀ *z*. *cmod z* ≤ *r* ⟶ *cmod* (*poly cs z*) ≤ *m*
by *blast*
let *?k* = *1* + *cmod c* + |*r* ∗ *m*|
have *kp*: *?k* > *0* **using** *abs-ge-zero*[*of r*∗*m*] *norm-ge-zero*[*of c*] **by** *arith*
{fix *z*
assume *H*: *cmod z* ≤ *r*
from *m H* **have** *th*: *cmod* (*poly cs z*) ≤ *m* **by** *blast*
from *H* **have** *rp*: *r* ≥ *0* **using** *norm-ge-zero*[*of z*] **by** *arith*
have *cmod* (*poly* (*pCons c cs*) *z*) ≤ *cmod c* + *cmod* (*z*∗ *poly cs z*)
using *norm-triangle-ineq*[*of c z*∗ *poly cs z*] **by** *simp*
also have ... ≤ *cmod c* + *r*∗*m* **using** *mult-mono*[*OF H th rp norm-ge-zero*[*of poly cs z*]] **by** (*simp add*: *norm-mult*)
also have ... ≤ *?k* **by** *simp*
finally have *cmod* (*poly* (*pCons c cs*) *z*) ≤ *?k* **.}**
with *kp* **show** *?case* **by** *blast*
qed

Offsetting the variable in a polynomial gives another of same degree

definition
offset-poly p h = *poly-rec 0* (λ*a p q*. *smult h q* + *pCons a q*) *p*

lemma *offset-poly-0*: *offset-poly 0 h* = *0*
unfolding *offset-poly-def* **by** (*simp add*: *poly-rec-0*)

lemma *offset-poly-pCons*:
offset-poly (*pCons a p*) *h* =
smult h (*offset-poly p h*) + *pCons a* (*offset-poly p h*)
unfolding *offset-poly-def* **by** (*simp add*: *poly-rec-pCons*)

lemma *offset-poly-single*: *offset-poly* [:*a*:] *h* = [:*a*:]
by (*simp add*: *offset-poly-pCons offset-poly-0*)

lemma *poly-offset-poly*: *poly* (*offset-poly p h*) *x* = *poly p* (*h* + *x*)
apply (*induct p*)
apply (*simp add*: *offset-poly-0*)
apply (*simp add*: *offset-poly-pCons algebra-simps*)

done

lemma *offset-poly-eq-0-lemma*: *smult c p + pCons a p = 0 ⟹ p = 0*
by (*induct p arbitrary*: *a*, *simp*, *force*)

lemma *offset-poly-eq-0-iff*: *offset-poly p h = 0 ⟷ p = 0*
apply (*safe intro*!: *offset-poly-0*)
apply (*induct p*, *simp*)
apply (*simp add*: *offset-poly-pCons*)
apply (*frule offset-poly-eq-0-lemma*, *simp*)
done

lemma *degree-offset-poly*: *degree* (*offset-poly p h*) = *degree p*
apply (*induct p*)
apply (*simp add*: *offset-poly-0*)
apply (*case-tac p = 0*)
apply (*simp add*: *offset-poly-0 offset-poly-pCons*)
apply (*simp add*: *offset-poly-pCons*)
apply (*subst degree-add-eq-right*)
apply (*rule le-less-trans* [*OF degree-smult-le*])
apply (*simp add*: *offset-poly-eq-0-iff*)
apply (*simp add*: *offset-poly-eq-0-iff*)
done

definition
psize p = (*if p = 0 then 0 else Suc* (*degree p*))

lemma *psize-eq-0-iff* [*simp*]: *psize p = 0 ⟷ p = 0*
unfolding *psize-def* **by** *simp*

lemma *poly-offset*: ∃ *q*. *psize q = psize p* ∧ (∀ *x*. *poly q* (*x*::*complex*) = *poly p* (*a* + *x*))
proof (*intro exI conjI*)
show *psize* (*offset-poly p a*) = *psize p*
unfolding *psize-def*
by (*simp add*: *offset-poly-eq-0-iff degree-offset-poly*)
show ∀ *x*. *poly* (*offset-poly p a*) *x* = *poly p* (*a* + *x*)
by (*simp add*: *poly-offset-poly*)
qed

An alternative useful formulation of completeness of the reals

lemma *real-sup-exists*: **assumes** *ex*: ∃ *x*. *P x* **and** *bz*: ∃ *z*. ∀ *x*. *P x* ⟶ *x* < *z*
shows ∃ (*s*::*real*). ∀ *y*. (∃ *x*. *P x* ∧ *y* < *x*) ⟷ *y* < *s*
proof−
from *ex bz* **obtain** *x Y* **where** *x*: *P x* **and** *Y*: ⋀*x*. *P x* ⟹ *x* < *Y* **by** *blast*
from *ex* **have** *thx*:∃ *x*. *x* ∈ *Collect P* **by** *blast*
from *bz* **have** *thY*: ∃ *Y*. *isUb UNIV* (*Collect P*) *Y*
by(*auto simp add*: *isUb-def isLub-def setge-def setle-def leastP-def Ball-def order-le-less*)

from *reals-complete*[*OF thx thY*] **obtain** *L* **where** *L*: *isLub UNIV* (*Collect P*) *L*
by *blast*
from *Y*[*OF x*] **have** *xY*: $x < Y$ **.**
from *L* **have** *L'*: $\forall x.\ P\ x \longrightarrow x \leq L$ **by** (*auto simp add*: *isUb-def isLub-def setge-def setle-def leastP-def Ball-def*)
from *Y* **have** *Y'*: $\forall x.\ P\ x \longrightarrow x \leq Y$
apply (*clarsimp*, *atomize* (*full*)) **by** *auto*
from *L Y'* **have** $L \leq Y$ **by** (*auto simp add*: *isUb-def isLub-def setge-def setle-def leastP-def Ball-def*)
{fix *y*
{fix *z* **assume** *z*: $P\ z\ y < z$
from *L' z* **have** $y < L$ **by** *auto* **}**
moreover
{assume *yL*: $y < L\ \forall z.\ P\ z \longrightarrow \neg\ y < z$
hence *nox*: $\forall z.\ P\ z \longrightarrow y \geq z$ **by** *auto*
from *nox L* **have** $y \geq L$ **by** (*auto simp add*: *isUb-def isLub-def setge-def setle-def leastP-def Ball-def*)
with *yL*(*1*) **have** *False* **by** *arith***}**
ultimately have $(\exists x.\ P\ x \wedge y < x) \longleftrightarrow y < L$ **by** *blast***}**
thus *?thesis* **by** *blast*
qed

43.4 Fundamental theorem of algebra

lemma *unimodular-reduce-norm*:
assumes *md*: *cmod z = 1*
shows *cmod* $(z + 1) < 1 \vee$ *cmod* $(z - 1) < 1 \vee$ *cmod* $(z + ii) < 1 \vee$ *cmod* $(z - ii) < 1$
proof−
obtain *x y* **where** *z*: *z = Complex x y* **by** (*cases z*, *auto*)
from *md z* **have** *xy*: $x\hat{}2 + y\hat{}2 = 1$ **by** (*simp add*: *cmod-def*)
{assume *C*: *cmod* $(z + 1) \geq 1$ *cmod* $(z - 1) \geq 1$ *cmod* $(z + ii) \geq 1$ *cmod* $(z - ii) \geq 1$
from *C z xy* **have** $2{*}x \leq 1\ 2{*}x \geq -1\ 2{*}y \leq 1\ 2{*}y \geq -1$
by (*simp-all add*: *cmod-def power2-eq-square algebra-simps*)
hence *abs* $(2{*}x) \leq 1$ *abs* $(2{*}y) \leq 1$ **by** *simp-all*
hence $(abs\ (2 * x))\hat{}2 <= 1\hat{}2\ (abs\ (2 * y))\ \hat{}2 <= 1\hat{}2$
by − (*rule power-mono*, *simp*, *simp*)+
hence *th0*: $4{*}x\hat{}2 \leq 1\ 4{*}y\hat{}2 \leq 1$
by (*simp-all add*: *power2-abs power-mult-distrib*)
from *add-mono*[*OF th0*] *xy* **have** *False* **by** *simp* **}**
thus *?thesis* **unfolding** *linorder-not-le*[*symmetric*] **by** *blast*
qed

Hence we can always reduce modulus of $1 + b\ z\hat{}n$ if nonzero

lemma *reduce-poly-simple*:
assumes *b*: $b \neq 0$ **and** *n*: *n≠0*
shows $\exists z.$ *cmod* $(1 + b * z\hat{}n) < 1$
using *n*

proof(*induct n rule*: *nat-less-induct*)
fix *n*
assume *IH*: $\forall\, m{<}n.\ m \neq 0 \longrightarrow (\exists\, z.\ cmod\ (1 + b * z\ \hat{}\ m) < 1)$ **and** *n*: $n \neq 0$
let *?P* = $\lambda z\ n.\ cmod\ (1 + b * z\ \hat{}\ n) < 1$
{assume *e*: *even n*
hence $\exists\, m.\ n = 2{*}m$ **by** *presburger*
then obtain *m* **where** *m*: $n = 2{*}m$ **by** *blast*
from *n m* **have** $m{\neq}0\ m < n$ **by** *presburger+*
with *IH*[*rule-format*, *of m*] **obtain** *z* **where** *z*: *?P z m* **by** *blast*
from *z* **have** *?P* (*csqrt z*) *n* **by** (*simp add*: *m power-mult csqrt*)
hence $\exists\, z.$ *?P z n* **..}**
moreover
{assume *o*: *odd n*
have *th0*: *cmod* (*complex-of-real* (*cmod b*) / *b*) = *1*
using *b* **by** (*simp add*: *norm-divide*)
from *o* **have** $\exists\, m.\ n = Suc\ (2{*}m)$ **by** *presburger+*
then obtain *m* **where** *m*: $n = Suc\ (2{*}m)$ **by** *blast*
from *unimodular-reduce-norm*[*OF th0*] *o*
have $\exists\, v.$ *cmod* (*complex-of-real* (*cmod b*) / *b* + *v*ˆ*n*) < *1*
apply (*cases cmod* (*complex-of-real* (*cmod b*) / *b* + *1*) < *1*, *rule-tac x=1* **in** *exI*, *simp*)
apply (*cases cmod* (*complex-of-real* (*cmod b*) / *b* − *1*) < *1*, *rule-tac x=−1* **in** *exI*, *simp add*: *diff-def*)
apply (*cases cmod* (*complex-of-real* (*cmod b*) / *b* + *ii*) < *1*)
apply (*cases even m*, *rule-tac x=ii* **in** *exI*, *simp add*: *m power-mult*)
apply (*rule-tac x=− ii* **in** *exI*, *simp add*: *m power-mult*)
apply (*cases even m*, *rule-tac x=− ii* **in** *exI*, *simp add*: *m power-mult diff-def*)
apply (*rule-tac x=ii* **in** *exI*, *simp add*: *m power-mult diff-def*)
done
then obtain *v* **where** *v*: *cmod* (*complex-of-real* (*cmod b*) / *b* + *v*ˆ*n*) < *1* **by** *blast*
let *?w* = *v* / *complex-of-real* (*root n* (*cmod b*))
from *odd-real-root-pow*[*OF o*, *of cmod b*]
have *th1*: *?w* ˆ *n* = *v*ˆ*n* / *complex-of-real* (*cmod b*)
by (*simp add*: *power-divide complex-of-real-power*)
have *th2*:*cmod* (*complex-of-real* (*cmod b*) / *b*) = *1* **using** *b* **by** (*simp add*: *norm-divide*)
hence *th3*: *cmod* (*complex-of-real* (*cmod b*) / *b*) ≥ *0* **by** *simp*
have *th4*: *cmod* (*complex-of-real* (*cmod b*) / *b*) ∗
cmod (*1* + *b* ∗ (*v* ˆ *n* / *complex-of-real* (*cmod b*)))
< *cmod* (*complex-of-real* (*cmod b*) / *b*) ∗ *1*
apply (*simp only*: *norm-mult*[*symmetric*] *right-distrib*)
using *b v* **by** (*simp add*: *th2*)

from *mult-less-imp-less-left*[*OF th4 th3*]
have *?P ?w n* **unfolding** *th1* **.**
hence $\exists\, z.$ *?P z n* **.. }**
ultimately show $\exists\, z.$ *?P z n* **by** *blast*
qed

Bolzano-Weierstrass type property for closed disc in complex plane.

lemma *metric-bound-lemma*: *cmod* $(x - y) <= |Re\ x - Re\ y| + |Im\ x - Im\ y|$
using *real-sqrt-sum-squares-triangle-ineq*[*of Re x − Re y 0 0 Im x − Im y*]
unfolding *cmod-def* **by** *simp*

lemma *bolzano-weierstrass-complex-disc*:
assumes r: $\forall n.\ cmod\ (s\ n) \leq r$
shows $\exists f\ z.\ subseq\ f \wedge (\forall e > 0.\ \exists N.\ \forall n \geq N.\ cmod\ (s\ (f\ n)) - z) < e)$
proof−
from *seq-monosub*[*of Re o s*]
obtain *f g* **where** *f*: *subseq f monoseq* $(\lambda n.\ Re\ (s\ (f\ n)))$
unfolding *o-def* **by** *blast*
from *seq-monosub*[*of Im o s o f*]
obtain *g* **where** *g*: *subseq g monoseq* $(\lambda n.\ Im\ (s(f(g\ n))))$ **unfolding** *o-def* **by** *blast*
let *?h* = *f o g*
from *r*[*rule-format*, *of 0*] **have** *rp*: $r \geq 0$ **using** *norm-ge-zero*[*of s 0*] **by** *arith*
have *th*:$\forall n.\ r + 1 \geq |\ Re\ (s\ n)|$
proof
fix *n*
from *abs-Re-le-cmod*[*of s n*] *r*[*rule-format*, *of n*] **show** $|Re\ (s\ n)| \leq r + 1$ **by** *arith*
qed
have *conv1*: *convergent* $(\lambda n.\ Re\ (s\ (\ f\ n)))$
apply (*rule Bseq-monoseq-convergent*)
apply (*simp add*: *Bseq-def*)
apply (*rule exI*[**where** *x*= *r* + *1*])
using *th rp* **apply** *simp*
using *f*(*2*) **.**
have *th*:$\forall n.\ r + 1 \geq |\ Im\ (s\ n)|$
proof
fix *n*
from *abs-Im-le-cmod*[*of s n*] *r*[*rule-format*, *of n*] **show** $|Im\ (s\ n)| \leq r + 1$ **by** *arith*
qed

have *conv2*: *convergent* $(\lambda n.\ Im\ (s\ (f\ (g\ n))))$
apply (*rule Bseq-monoseq-convergent*)
apply (*simp add*: *Bseq-def*)
apply (*rule exI*[**where** *x*= *r* + *1*])
using *th rp* **apply** *simp*
using *g*(*2*) **.**

from *conv1*[*unfolded convergent-def*] **obtain** *x* **where** *LIMSEQ* $(\lambda n.\ Re\ (s\ (f\ n)))\ x$
by *blast*
hence *x*: $\forall r > 0.\ \exists n0.\ \forall n \geq n0.\ |\ Re\ (s\ (f\ n)) - x\ | < r$
unfolding *LIMSEQ-iff real-norm-def* **.**

from *conv2*[*unfolded convergent-def*] **obtain** *y* **where** *LIMSEQ* (λ*n*. *Im* (*s* (*f* (*g n*)))) *y*
by *blast*
hence *y*: ∀ *r*>*0*. ∃ *n0*. ∀ *n*≥*n0*. | *Im* (*s* (*f* (*g n*))) − *y* | < *r*
unfolding *LIMSEQ-iff real-norm-def* **.**
let *?w* = *Complex x y*
from *f*(*1*) *g*(*1*) **have** *hs*: *subseq ?h* **unfolding** *subseq-def* **by** *auto*
{fix *e* **assume** *ep*: *e* > (*0*::*real*)
hence *e2*: *e*/*2* > *0* **by** *simp*
from *x*[*rule-format*, *OF e2*] *y*[*rule-format*, *OF e2*]
obtain *N1 N2* **where** *N1*: ∀ *n*≥*N1*. |*Re* (*s* (*f n*)) − *x*| < *e* / *2* **and** *N2*: ∀ *n*≥*N2*. |*Im* (*s* (*f* (*g n*))) − *y*| < *e* / *2* **by** *blast*
{fix *n* **assume** *nN12*: *n* ≥ *N1* + *N2*
hence *nN1*: *g n* ≥ *N1* **and** *nN2*: *n* ≥ *N2* **using** *seq-suble*[*OF g*(*1*), *of n*] **by** *arith*+
from *add-strict-mono*[*OF N1*[*rule-format*, *OF nN1*] *N2*[*rule-format*, *OF nN2*]]
have *cmod* (*s* (*?h n*) − *?w*) < *e*
using *metric-bound-lemma*[*of s* (*f* (*g n*)) *?w*] **by** *simp* **}**
hence ∃ *N*. ∀ *n*≥*N*. *cmod* (*s* (*?h n*) − *?w*) < *e* **by** *blast* **}**
with *hs* **show** *?thesis* **by** *blast*
qed

Polynomial is continuous.

lemma *poly-cont*:
assumes *ep*: *e* > *0*
shows ∃ *d* >*0*. ∀ *w*. *0* < *cmod* (*w* − *z*) ∧ *cmod* (*w* − *z*) < *d* ⟶ *cmod* (*poly p w* − *poly p z*) < *e*
proof−
obtain *q* **where** *q*: *degree q* = *degree p* ⋀*x*. *poly q x* = *poly p* (*z* + *x*)
proof
show *degree* (*offset-poly p z*) = *degree p*
by (*rule degree-offset-poly*)
show ⋀*x*. *poly* (*offset-poly p z*) *x* = *poly p* (*z* + *x*)
by (*rule poly-offset-poly*)
qed
{fix *w*
note *q*(*2*)[*of w* − *z*, *simplified*]**}**
note *th* = *this*
show *?thesis* **unfolding** *th*[*symmetric*]
proof(*induct q*)
case *0* **thus** *?case* **using** *ep* **by** *auto*
next
case (*pCons c cs*)
from *poly-bound-exists*[*of 1 cs*]
obtain *m* **where** *m*: *m* > *0* ⋀*z*. *cmod z* ≤ *1* ⟹ *cmod* (*poly cs z*) ≤ *m* **by** *blast*
from *ep m*(*1*) **have** *em0*: *e*/*m* > *0* **by** (*simp add*: *field-simps*)
have *one0*: *1* > (*0*::*real*) **by** *arith*

```
  from real-lbound-gt-zero[OF one0 em0]
  obtain d where d: d >0 d < 1 d < e / m by blast
  from d(1,3) m(1) have dm: d*m > 0 d*m < e
   by (simp-all add: field-simps mult-pos-pos)
  show ?case
   proof(rule ex-forward[OF real-lbound-gt-zero[OF one0 em0]], clarsimp simp
add: norm-mult)
    fix d w
    assume H: d > 0 d < 1 d < e/m w≠z cmod (w−z) < d
    hence d1: cmod (w−z) ≤ 1 d ≥ 0 by simp-all
    from H(3) m(1) have dme: d*m < e by (simp add: field-simps)
    from H have th: cmod (w−z) ≤ d by simp
    from mult-mono[OF th m(2)[OF d1(1)] d1(2) norm-ge-zero] dme
    show cmod (w − z) * cmod (poly cs (w − z)) < e by simp
   qed
  qed
qed
```

Hence a polynomial attains minimum on a closed disc in the complex plane.

```
lemma poly-minimum-modulus-disc:
 ∃ z. ∀ w. cmod w ≤ r ⟶ cmod (poly p z) ≤ cmod (poly p w)
proof−
 {assume ¬ r ≥ 0 hence ?thesis unfolding linorder-not-le
   apply −
   apply (rule exI[where x=0])
   apply auto
   apply (subgoal-tac cmod w < 0)
   apply simp
   apply arith
   done }
 moreover
 {assume rp: r ≥ 0
  from rp have cmod 0 ≤ r ∧ cmod (poly p 0) = − (− cmod (poly p 0)) by
simp
  hence mth1: ∃ x z. cmod z ≤ r ∧ cmod (poly p z) = − x  by blast
  {fix x z
   assume H: cmod z ≤ r cmod (poly p z) = − x ¬x < 1
   hence − x < 0  by arith
   with H(2) norm-ge-zero[of poly p z]  have False by simp }
  then have mth2: ∃ z. ∀ x. (∃ z. cmod z ≤ r ∧ cmod (poly p z) = − x) ⟶ x
< z by blast
  from real-sup-exists[OF mth1 mth2] obtain s where
   s: ∀ y. (∃ x. (∃ z. cmod z ≤ r ∧ cmod (poly p z) = − x) ∧ y < x) ⟷(y <
s) by blast
  let ?m = −s
  {fix y
   from s[rule-format, of −y] have
  (∃ z x. cmod z ≤ r ∧ −(− cmod (poly p z)) < y) ⟷ ?m < y
```

unfolding *minus-less-iff*[*of y*] *equation-minus-iff* **by** *blast* **}**
note *s1* = *this*[*unfolded minus-minus*]
from *s1*[*of ?m*] **have** *s1m*: $\bigwedge z\ x.\ cmod\ z \leq r \Longrightarrow cmod\ (poly\ p\ z) \geq ?m$
by *auto*
{fix *n*::*nat*
from *s1*[*rule-format*, *of ?m* + *1*/*real* (*Suc n*)]
have $\exists z.\ cmod\ z \leq r \wedge cmod\ (poly\ p\ z) < -\ s + 1\ /\ real\ (Suc\ n)$
by *simp***}**
hence *th*: $\forall n.\ \exists z.\ cmod\ z \leq r \wedge cmod\ (poly\ p\ z) < -\ s + 1\ /\ real\ (Suc\ n)$ **..**
from *choice*[*OF th*] **obtain** *g* **where**
g: $\forall n.\ cmod\ (g\ n) \leq r\ \forall n.\ cmod\ (poly\ p\ (g\ n)) <$*?m+1 /real*(*Suc n*)
by *blast*
from *bolzano-weierstrass-complex-disc*[*OF g*(*1*)]
obtain *f z* **where** *fz*: *subseq f* $\forall e{>}0.\ \exists N.\ \forall n{\geq}N.\ cmod\ (g\ (f\ n) - z) < e$
by *blast*
{fix *w*
assume *wr*: *cmod w* ≤ *r*
let *?e* = |*cmod* (*poly p z*) − *?m*|
{assume *e*: *?e* > *0*
hence *e2*: *?e*/*2* > *0* **by** *simp*
from *poly-cont*[*OF e2*, *of z p*] **obtain** *d* **where**
d: $d{>}0\ \forall w.\ 0{<}cmod\ (w - z) \wedge cmod(w - z) < d \longrightarrow cmod(poly\ p\ w -$
poly p z) < *?e*/*2* **by** *blast*
{fix *w* **assume** *w*: *cmod* (*w* − *z*) < *d*
have *cmod*(*poly p w* − *poly p z*) < *?e* / *2*
using *d*(*2*)[*rule-format*, *of w*] *w e* **by** (*cases w*=*z*, *simp-all*)**}**
note *th1* = *this*

from *fz*(*2*)[*rule-format*, *OF d*(*1*)] **obtain** *N1* **where**
N1: $\forall n{\geq}N1.\ cmod\ (g\ (f\ n) - z) < d$ **by** *blast*
from *reals-Archimedean2*[*of 2*/*?e*] **obtain** *N2*::*nat* **where**
N2: *2*/*?e* < *real N2* **by** *blast*
have *th2*: *cmod*(*poly p* (*g*(*f*(*N1* + *N2*))) − *poly p z*) < *?e*/*2*
using *N1*[*rule-format*, *of N1* + *N2*] *th1* **by** *simp*
{fix *a b e2 m* :: *real*
have *a* < *e2* ⟹ *abs*(*b* − *m*) < *e2* ⟹ *2* ∗ *e2* <= *abs*(*b* − *m*) + *a*
==> *False* **by** *arith***}**
note *th0* = *this*
have *ath*:
$\bigwedge m\ x\ e.\ m <= x \Longrightarrow\ x < m + e$ ==> *abs*(*x* − *m*::*real*) < *e* **by** *arith*
from *s1m*[*OF g*(*1*)[*rule-format*]]
have *th31*: *?m* ≤ *cmod*(*poly p* (*g* (*f* (*N1* + *N2*)))) **.**
from *seq-suble*[*OF fz*(*1*), *of N1*+*N2*]
have *th00*: *real* (*Suc* (*N1*+*N2*)) ≤ *real* (*Suc* (*f* (*N1*+*N2*))) **by** *simp*
have *th000*: *0* ≤ (*1*::*real*) (*1*::*real*) ≤ *1 real* (*Suc* (*N1*+*N2*)) > *0*
using *N2* **by** *auto*
from *frac-le*[*OF th000 th00*] **have** *th00*: *?m* +*1* / *real* (*Suc* (*f* (*N1* + *N2*)))
≤ *?m* + *1* / *real* (*Suc* (*N1* + *N2*)) **by** *simp*
from *g*(*2*)[*rule-format*, *of f* (*N1* + *N2*)]

have *th01*:*cmod* (*poly p* (*g* (*f* (*N1* + *N2*)))) < − *s* + *1* / *real* (*Suc* (*f* (*N1* + *N2*))) **.**
from *order-less-le-trans*[*OF th01 th00*]
have *th32*: *cmod*(*poly p* (*g* (*f* (*N1* + *N2*)))) < *?m* + (*1*/ *real*(*Suc* (*N1* + *N2*))) **.**
from *N2* **have** *2*/*?e* < *real* (*Suc* (*N1* + *N2*)) **by** *arith*
with *e2 less-imp-inverse-less*[*of 2*/*?e real* (*Suc* (*N1* + *N2*))]
have *?e*/*2* > *1*/ *real* (*Suc* (*N1* + *N2*)) **by** (*simp add*: *inverse-eq-divide*)
with *ath*[*OF th31 th32*]
have *thc1*:|*cmod*(*poly p* (*g* (*f* (*N1* + *N2*)))) − *?m*|< *?e*/*2* **by** *arith*
have *ath2*: ⋀(*a*::*real*) *b c m*. |*a* − *b*| <= *c* ==> |*b* − *m*| <= |*a* − *m*| + *c*
by *arith*
have *th22*: |*cmod* (*poly p* (*g* (*f* (*N1* + *N2*)))) − *cmod* (*poly p z*)|
≤ *cmod* (*poly p* (*g* (*f* (*N1* + *N2*))) − *poly p z*)
by (*simp add*: *norm-triangle-ineq3*)
from *ath2*[*OF th22*, *of ?m*]
have *thc2*: *2*∗(*?e*/*2*) ≤ |*cmod*(*poly p* (*g* (*f* (*N1* + *N2*)))) − *?m*| + *cmod* (*poly p* (*g* (*f* (*N1* + *N2*))) − *poly p z*) **by** *simp*
from *th0*[*OF th2 thc1 thc2*] **have** *False* **.}**
hence *?e* = *0* **by** *auto*
then have *cmod* (*poly p z*) = *?m* **by** *simp*
with *s1m*[*OF wr*]
have *cmod* (*poly p z*) ≤ *cmod* (*poly p w*) **by** *simp* **}**
hence *?thesis* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

lemma (*rcis* (*sqrt* (*abs r*)) (*a*/*2*)) ^ *2* = *rcis* (*abs r*) *a*
unfolding *power2-eq-square*
apply (*simp add*: *rcis-mult*)
apply (*simp add*: *power2-eq-square*[*symmetric*])
done

lemma *cispi*: *cis pi* = −*1*
unfolding *cis-def*
by *simp*

lemma (*rcis* (*sqrt* (*abs r*)) ((*pi* + *a*)/*2*)) ^ *2* = *rcis* (− *abs r*) *a*
unfolding *power2-eq-square*
apply (*simp add*: *rcis-mult add-divide-distrib*)
apply (*simp add*: *power2-eq-square*[*symmetric*] *rcis-def cispi cis-mult*[*symmetric*])
done

Nonzero polynomial in z goes to infinity as z does.

lemma *poly-infinity*:
assumes *ex*: *p* ≠ *0*
shows ∃ *r*. ∀ *z*. *r* ≤ *cmod z* ⟶ *d* ≤ *cmod* (*poly* (*pCons a p*) *z*)
using *ex*
proof(*induct p arbitrary*: *a d*)

```
  case (pCons c cs a d)
  {assume H: cs ≠ 0
   with pCons.hyps obtain r where r: ∀ z. r ≤ cmod z ⟶ d + cmod a ≤ cmod
(poly (pCons c cs) z) by blast
    let ?r = 1 + |r|
    {fix z assume h: 1 + |r| ≤ cmod z
     have r0: r ≤ cmod z using h by arith
     from r[rule-format, OF r0]
     have th0: d + cmod a ≤ 1 * cmod(poly (pCons c cs) z) by arith
     from h have z1: cmod z ≥ 1 by arith
    from order-trans[OF th0 mult-right-mono[OF z1 norm-ge-zero[of poly (pCons
c cs) z]]]
     have th1: d ≤ cmod(z * poly (pCons c cs) z) - cmod a
       unfolding norm-mult by (simp add: algebra-simps)
     from complex-mod-triangle-sub[of z * poly (pCons c cs) z a]
     have th2: cmod(z * poly (pCons c cs) z) - cmod a ≤ cmod (poly (pCons a
(pCons c cs)) z)
       by (simp add: diff-le-eq algebra-simps)
     from th1 th2 have d ≤ cmod (poly (pCons a (pCons c cs)) z)  by arith}
   hence ?case by blast}
  moreover
  {assume cs0: ¬ (cs ≠ 0)
   with pCons.prems have c0: c ≠ 0 by simp
   from cs0 have cs0': cs = 0 by simp
   {fix z
    assume h: (|d| + cmod a) / cmod c ≤ cmod z
    from c0 have cmod c > 0 by simp
    from h c0 have th0: |d| + cmod a ≤ cmod (z*c)
      by (simp add: field-simps norm-mult)
    have ath: ⋀mzh mazh ma. mzh <= mazh + ma ==> abs(d) + ma <= mzh
==> d <= mazh by arith
    from complex-mod-triangle-sub[of z*c a ]
    have th1: cmod (z * c) ≤ cmod (a + z * c) + cmod a
      by (simp add: algebra-simps)
    from ath[OF th1 th0] have d ≤ cmod (poly (pCons a (pCons c cs)) z)
      using cs0' by simp}
   then have ?case  by blast}
  ultimately show ?case by blast
qed simp
```

Hence polynomial's modulus attains its minimum somewhere.

```
lemma poly-minimum-modulus:
  ∃ z.∀ w. cmod (poly p z) ≤ cmod (poly p w)
proof(induct p)
  case (pCons c cs)
  {assume cs0: cs ≠ 0
   from poly-infinity[OF cs0, of cmod (poly (pCons c cs) 0) c]
   obtain r where r: ⋀z. r ≤ cmod z ⟹ cmod (poly (pCons c cs) 0) ≤ cmod
(poly (pCons c cs) z) by blast
```

```
    have ath: ⋀z r. r ≤ cmod z ∨ cmod z ≤ |r| by arith
    from poly-minimum-modulus-disc[of |r| pCons c cs]
   obtain v where v: ⋀w. cmod w ≤ |r| ⟹ cmod (poly (pCons c cs) v) ≤ cmod
(poly (pCons c cs) w) by blast
    {fix z assume z: r ≤ cmod z
      from v[of 0] r[OF z]
      have cmod (poly (pCons c cs) v) ≤ cmod (poly (pCons c cs) z)
        by simp }
    note v0 = this
    from v0 v ath[of r] have ?case by blast}
  moreover
  {assume cs0: ¬ (cs ≠ 0)
    hence th:cs = 0 by simp
    from th pCons.hyps have ?case by simp}
  ultimately show ?case by blast
qed simp
```

Constant function (non-syntactic characterization).

```
definition constant f = (∀ x y. f x = f y)

lemma nonconstant-length: ¬ (constant (poly p)) ⟹ psize p ≥ 2
  unfolding constant-def psize-def
  apply (induct p, auto)
  done

lemma poly-replicate-append:
  poly (monom 1 n ∗ p) (x::'a::{comm-ring-1}) = x^n ∗ poly p x
  by (simp add: poly-monom)
```

Decomposition of polynomial, skipping zero coefficients after the first.

```
lemma poly-decompose-lemma:
 assumes nz: ¬(∀ z. z≠0 ⟶ poly p z = (0::'a::{idom}))
  shows ∃ k a q. a≠0 ∧ Suc (psize q + k) = psize p ∧
               (∀ z. poly p z = z^k ∗ poly (pCons a q) z)
unfolding psize-def
using nz
proof(induct p)
  case 0 thus ?case by simp
next
  case (pCons c cs)
  {assume c0: c = 0
    from pCons.hyps pCons.prems c0 have ?case
      apply (auto)
      apply (rule-tac x=k+1 in exI)
      apply (rule-tac x=a in exI, clarsimp)
      apply (rule-tac x=q in exI)
      by (auto)}
  moreover
  {assume c0: c≠0
```

```
    hence ?case apply−
      apply (rule exI[where x=0])
      apply (rule exI[where x=c], clarsimp)
      apply (rule exI[where x=cs])
      apply auto
      done}
  ultimately show ?case by blast
qed

lemma poly-decompose:
  assumes nc: ~constant(poly p)
  shows ∃k a q. a≠(0::'a::{idom}) ∧ k≠0 ∧
              psize q + k + 1 = psize p ∧
             (∀z. poly p z = poly p 0 + z^k * poly (pCons a q) z)
using nc
proof(induct p)
  case 0 thus ?case by (simp add: constant-def)
next
  case (pCons c cs)
  {assume C:∀z. z ≠ 0 ⟶ poly cs z = 0
    {fix x y
      from C have poly (pCons c cs) x = poly (pCons c cs) y by (cases x=0,
auto)}
    with pCons.prems have False by (auto simp add: constant-def)}
  hence th: ¬ (∀z. z ≠ 0 ⟶ poly cs z = 0) ..
  from poly-decompose-lemma[OF th]
  show ?case
    apply clarsimp
    apply (rule-tac x=k+1 in exI)
    apply (rule-tac x=a in exI)
    apply simp
    apply (rule-tac x=q in exI)
    apply (auto simp add: power-Suc)
    apply (auto simp add: psize-def split: if-splits)
    done
qed
```

Fundamental theorem of algebra

```
lemma fundamental-theorem-of-algebra:
  assumes nc: ~constant(poly p)
  shows ∃z::complex. poly p z = 0
using nc
proof(induct psize p arbitrary: p rule: less-induct)
  case less
  let ?p = poly p
  let ?ths = ∃z. ?p z = 0

  from nonconstant-length[OF less(2)] have n2: psize p ≥ 2 .
  from poly-minimum-modulus obtain c where
```

```
  c: ∀ w. cmod (?p c) ≤ cmod (?p w) by blast
 {assume pc: ?p c = 0 hence ?ths by blast}
 moreover
 {assume pc0: ?p c ≠ 0
  from poly-offset[of p c] obtain q where
    q: psize q = psize p ∀ x. poly q x = ?p (c+x) by blast
  {assume h: constant (poly q)
   from q(2) have th: ∀ x. poly q (x − c) = ?p x by auto
   {fix x y
    from th have ?p x = poly q (x − c) by auto
    also have ... = poly q (y − c)
      using h unfolding constant-def by blast
    also have ... = ?p y using th by auto
    finally have ?p x = ?p y .}
   with less(2) have False unfolding constant-def by blast }
  hence qnc: ¬ constant (poly q) by blast
  from q(2) have pqc0: ?p c = poly q 0 by simp
  from c pqc0 have cq0: ∀ w. cmod (poly q 0) ≤ cmod (?p w) by simp
  let ?a0 = poly q 0
  from pc0 pqc0 have a00: ?a0 ≠ 0 by simp
  from a00
  have qr: ∀ z. poly q z = poly (smult (inverse ?a0) q) z * ?a0
    by simp
  let ?r = smult (inverse ?a0) q
  have lgqr: psize q = psize ?r
    using a00 unfolding psize-def degree-def
    by (simp add: expand-poly-eq)
  {assume h: ⋀x y. poly ?r x = poly ?r y
    {fix x y
     from qr[rule-format, of x]
     have poly q x = poly ?r x * ?a0 by auto
     also have ... = poly ?r y * ?a0 using h by simp
     also have ... = poly q y using qr[rule-format, of y] by simp
     finally have poly q x = poly q y .}
    with qnc have False unfolding constant-def by blast}
  hence rnc: ¬ constant (poly ?r) unfolding constant-def by blast
  from qr[rule-format, of 0] a00  have r01: poly ?r 0 = 1 by auto
  {fix w
   have cmod (poly ?r w) < 1 ⟷ cmod (poly q w / ?a0) < 1
     using qr[rule-format, of w] a00 by (simp add: divide-inverse mult-ac)
   also have ... ⟷ cmod (poly q w) < cmod ?a0
     using a00 unfolding norm-divide by (simp add: field-simps)
   finally have cmod (poly ?r w) < 1 ⟷ cmod (poly q w) < cmod ?a0 .}
  note mrmq-eq = this
  from poly-decompose[OF rnc] obtain k a s where
    kas: a≠0 k≠0 psize s + k + 1 = psize ?r
    ∀ z. poly ?r z = poly ?r 0 + z^k* poly (pCons a s) z by blast
  {assume psize p = k + 1
   with kas(3) lgqr[symmetric] q(1) have s0:s=0 by auto
```

{fix *w*
have *cmod (poly ?r w) = cmod (1 + a * w ^ k)*
using *kas(4)[rule-format, of w] s0 r01* **by** (*simp add*: *algebra-simps*)**}**
note *hth = this* [*symmetric*]
from *reduce-poly-simple*[*OF kas(1,2)*]
have ∃ *w*. *cmod (poly ?r w) < 1* **unfolding** *hth* **by** *blast***}**
moreover
{assume *kn*: *psize p ≠ k+1*
from *kn kas(3) q(1) lgqr* **have** *k1n*: *k + 1 < psize p* **by** *simp*
have *th01*: ¬ *constant (poly (pCons 1 (monom a (k − 1))))*
unfolding *constant-def poly-pCons poly-monom*
using *kas(1)* **apply** *simp*
by (*rule exI*[**where** *x=0*], *rule exI*[**where** *x=1*], *simp*)
from *kas(1) kas(2)* **have** *th02*: *k+1 = psize (pCons 1 (monom a (k − 1)))*
by (*simp add*: *psize-def degree-monom-eq*)
from *less(1)* [*OF k1n* [*simplified th02*] *th01*]
obtain *w* **where** *w*: *1 + w^k * a = 0*
unfolding *poly-pCons poly-monom*
using *kas(2)* **by** (*cases k*, *auto simp add*: *algebra-simps*)
from *poly-bound-exists*[*of cmod w s*] **obtain** *m* **where**
m: *m > 0* ∀ *z*. *cmod z ≤ cmod w ⟶ cmod (poly s z) ≤ m* **by** *blast*
have *w0*: *w≠0* **using** *kas(2) w* **by** (*auto simp add*: *power-0-left*)
from *w* **have** *(1 + w ^ k * a) − 1 = 0 − 1* **by** *simp*
then have *wm1*: *w^k * a = − 1* **by** *simp*
have *inv0*: *0 < inverse (cmod w ^ (k + 1) * m)*
using *norm-ge-zero*[*of w*] *w0 m(1)*
by (*simp add*: *inverse-eq-divide zero-less-mult-iff*)
with *real-down2*[*OF zero-less-one*] **obtain** *t* **where**
t: *t > 0 t < 1 t < inverse (cmod w ^ (k + 1) * m)* **by** *blast*
let *?ct = complex-of-real t*
let *?w = ?ct * w*
have *1 + ?w^k * (a + ?w * poly s ?w) = 1 + ?ct^k * (w^k * a) + ?w^k * ?w * poly s ?w* **using** *kas(1)* **by** (*simp add*: *algebra-simps power-mult-distrib*)
also have ... = *complex-of-real (1 − t^k) + ?w^k * ?w * poly s ?w*
unfolding *wm1* **by** (*simp*)
finally have *cmod (1 + ?w^k * (a + ?w * poly s ?w)) = cmod (complex-of-real (1 − t^k) + ?w^k * ?w * poly s ?w)*
apply −
apply (*rule cong*[*OF refl*[*of cmod*]])
apply *assumption*
done
with *norm-triangle-ineq*[*of complex-of-real (1 − t^k) ?w^k * ?w * poly s ?w*]
have *th11*: *cmod (1 + ?w^k * (a + ?w * poly s ?w)) ≤ |1 − t^k| + cmod (?w^k * ?w * poly s ?w)* **unfolding** *norm-of-real* **by** *simp*
have *ath*: ⋀*x* (*t*::*real*). *0≤ x ⟹ x < t ⟹ t≤1 ⟹ |1 − t| + x < 1* **by** *arith*
have *t *cmod w ≤ 1 * cmod w* **apply** (*rule mult-mono*) **using** *t(1,2)* **by** *auto*
then have *tw*: *cmod ?w ≤ cmod w* **using** *t(1)* **by** (*simp add*: *norm-mult*)

```
    from t inv0 have t* (cmod w ^ (k + 1) * m) < 1
      by (simp add: inverse-eq-divide field-simps)
    with zero-less-power[OF t(1), of k]
    have th30: t^k * (t* (cmod w ^ (k + 1) * m)) < t^k * 1
      apply − apply (rule mult-strict-left-mono) by simp-all
     have cmod (?w^k * ?w * poly s ?w) = t^k * (t* (cmod w ^ (k+1) * cmod
(poly s ?w))) using w0 t(1)
       by (simp add: algebra-simps power-mult-distrib norm-of-real norm-power
norm-mult)
    then have cmod (?w^k * ?w * poly s ?w) ≤ t^k * (t* (cmod w ^ (k + 1) *
m))
      using t(1,2) m(2)[rule-format, OF tw] w0
      apply (simp only: )
      apply auto
      apply (rule mult-mono, simp-all add: norm-ge-zero)+
      apply (simp add: zero-le-mult-iff zero-le-power)
      done
    with th30 have th120: cmod (?w^k * ?w * poly s ?w) < t^k by simp
    from power-strict-mono[OF t(2), of k] t(1) kas(2) have th121: t^k ≤ 1
      by auto
    from ath[OF norm-ge-zero[of ?w^k * ?w * poly s ?w] th120 th121]
    have th12: |1 − t^k| + cmod (?w^k * ?w * poly s ?w) < 1 .
    from th11 th12
    have cmod (1 + ?w^k * (a + ?w * poly s ?w)) < 1 by arith
    then have cmod (poly ?r ?w) < 1
      unfolding kas(4)[rule-format, of ?w] r01 by simp
    then have ∃ w. cmod (poly ?r w) < 1 by blast}
  ultimately have cr0-contr: ∃ w. cmod (poly ?r w) < 1 by blast
  from cr0-contr cq0 q(2)
  have ?ths unfolding mrmq-eq not-less[symmetric] by auto}
 ultimately show ?ths by blast
qed
```

Alternative version with a syntactic notion of constant polynomial.

```
lemma fundamental-theorem-of-algebra-alt:
 assumes nc: ~(∃ a l. a≠ 0 ∧ l = 0 ∧ p = pCons a l)
 shows ∃ z. poly p z = (0::complex)
using nc
proof(induct p)
 case (pCons c cs)
 {assume c=0 hence ?case by auto}
 moreover
 {assume c0: c≠0
  {assume nc: constant (poly (pCons c cs))
   from nc[unfolded constant-def, rule-format, of 0]
   have ∀ w. w ≠ 0 ⟶ poly cs w = 0 by auto
   hence cs = 0
    proof(induct cs)
     case (pCons d ds)
```

```
    {assume d=0 hence ?case using pCons.prems pCons.hyps by simp}
    moreover
    {assume d0: d≠0
     from poly-bound-exists[of 1 ds] obtain m where
       m: m > 0 ∀ z. ∀ z. cmod z ≤ 1 ⟶ cmod (poly ds z) ≤ m by blast
     have dm: cmod d / m > 0 using d0 m(1) by (simp add: field-simps)
     from real-down2[OF dm zero-less-one] obtain x where
       x: x > 0 x < cmod d / m x < 1 by blast
     let ?x = complex-of-real x
     from x have cx: ?x ≠ 0  cmod ?x ≤ 1 by simp-all
     from pCons.prems[rule-format, OF cx(1)]
   have cth: cmod (?x*poly ds ?x) = cmod d by (simp add: eq-diff-eq[symmetric])
     from m(2)[rule-format, OF cx(2)] x(1)
     have th0: cmod (?x*poly ds ?x) ≤ x*m
       by (simp add: norm-mult)
     from x(2) m(1) have x*m < cmod d by (simp add: field-simps)
     with th0 have cmod (?x*poly ds ?x) ≠ cmod d by auto
     with cth  have ?case by blast}
    ultimately show ?case by blast
   qed simp}
  then have nc: ¬ constant (poly (pCons c cs)) using pCons.prems c0
   by blast
  from fundamental-theorem-of-algebra[OF nc] have ?case .}
 ultimately show ?case by blast
qed simp
```

43.5 Nullstellensatz, degrees and divisibility of polynomials

```
lemma nullstellensatz-lemma:
 fixes p :: complex poly
 assumes ∀ x. poly p x = 0 ⟶ poly q x = 0
 and degree p = n and n ≠ 0
 shows p dvd (q ^ n)
using prems
proof(induct n arbitrary: p q rule: nat-less-induct)
 fix n::nat fix p q :: complex poly
 assume IH: ∀ m<n. ∀ p q.
              (∀ x. poly p x = (0::complex) ⟶ poly q x = 0) ⟶
              degree p = m ⟶ m ≠ 0 ⟶ p dvd (q ^ m)
   and pq0: ∀ x. poly p x = 0 ⟶ poly q x = 0
   and dpn: degree p = n and n0: n ≠ 0
 from dpn n0 have pne: p ≠ 0 by auto
 let ?ths = p dvd (q ^ n)
 {fix a assume a: poly p a = 0
  {assume oa: order a p ≠ 0
   let ?op = order a p
   from pne have ap: ([:- a, 1:] ^ ?op) dvd p
     ¬ [:- a, 1:] ^ (Suc ?op) dvd p using order by blast+
   note oop = order-degree[OF pne, unfolded dpn]
```

```
{assume q0: q = 0
 hence ?ths using n0
  by (simp add: power-0-left)}
moreover
{assume q0: q ≠ 0
 from pq0[rule-format, OF a, unfolded poly-eq-0-iff-dvd]
 obtain r where r: q = [:- a, 1:] * r by (rule dvdE)
 from ap(1) obtain s where
  s: p = [:- a, 1:] ^ ?op * s by (rule dvdE)
 have sne: s ≠ 0
  using s pne by auto
 {assume ds0: degree s = 0
  from ds0 have ∃k. s = [:k:]
   by (cases s, simp split: if-splits)
  then obtain k where kpn: s = [:k:] by blast
  from sne kpn have k: k ≠ 0 by simp
  let ?w = ([:1/k:] * ([:-a,1:] ^ (n - ?op))) * (r ^ n)
  from k oop [of a] have q ^ n = p * ?w
   apply -
   apply (subst r, subst s, subst kpn)
   apply (subst power-mult-distrib, simp)
   apply (subst power-add [symmetric], simp)
   done
  hence ?ths unfolding dvd-def by blast}
 moreover
 {assume ds0: degree s ≠ 0
  from ds0 sne dpn s oa
   have dsn: degree s < n apply auto
    apply (erule ssubst)
    apply (simp add: degree-mult-eq degree-linear-power)
    done
   {fix x assume h: poly s x = 0
    {assume xa: x = a
     from h[unfolded xa poly-eq-0-iff-dvd] obtain u where
       u: s = [:- a, 1:] * u by (rule dvdE)
     have p = [:- a, 1:] ^ (Suc ?op) * u
      by (subst s, subst u, simp only: power-Suc mult-ac)
     with ap(2)[unfolded dvd-def] have False by blast}
    note xa = this
    from h have poly p x = 0 by (subst s, simp)
    with pq0 have poly q x = 0 by blast
    with r xa have poly r x = 0
     by (auto simp add: uminus-add-conv-diff)}
   note impth = this
   from IH[rule-format, OF dsn, of s r] impth ds0
   have s dvd (r ^ (degree s)) by blast
   then obtain u where u: r ^ (degree s) = s * u ..
   hence u': ⋀x. poly s x * poly u x = poly r x ^ degree s
    by (simp only: poly-mult[symmetric] poly-power[symmetric])
```

```
        let ?w = (u * ([:-a,1:] ^ (n - ?op))) * (r ^ (n - degree s))
        from oop[of a] dsn have q ^ n = p * ?w
          apply -
          apply (subst s, subst r)
          apply (simp only: power-mult-distrib)
          apply (subst mult-assoc [where b=s])
          apply (subst mult-assoc [where a=u])
          apply (subst mult-assoc [where b=u, symmetric])
          apply (subst u [symmetric])
          apply (simp add: mult-ac power-add [symmetric])
          done
        hence ?ths unfolding dvd-def by blast}
    ultimately have ?ths by blast }
    ultimately have ?ths by blast}
  then have ?ths using a order-root pne by blast}
 moreover
 {assume exa: ¬ (∃ a. poly p a = 0)
  from fundamental-theorem-of-algebra-alt[of p] exa obtain c where
    ccs: c≠0 p = pCons c 0 by blast

  then have pp: ⋀x. poly p x =  c by simp
  let ?w = [:1/c:] * (q ^ n)
  from ccs
  have (q ^ n) = (p * ?w)
    by (simp add: smult-smult)
  hence ?ths unfolding dvd-def by blast}
 ultimately show ?ths by blast
qed

lemma nullstellensatz-univariate:
  (∀ x. poly p x = (0::complex) ⟶ poly q x = 0) ⟷
    p dvd (q ^ (degree p)) ∨ (p = 0 ∧ q = 0)
proof−
 {assume pe: p = 0
  hence eq: (∀ x. poly p x = (0::complex) ⟶ poly q x = 0) ⟷ q = 0
    apply auto
    apply (rule poly-zero [THEN iffD1])
    by (rule ext, simp)
  {assume p dvd (q ^ (degree p))
    then obtain r where r: q ^ (degree p) = p * r ..
    from r pe have False by simp}
  with eq pe have ?thesis by blast}
 moreover
 {assume pe: p ≠ 0
  {assume dp: degree p = 0
    then obtain k where k: p = [:k:] k≠0 using pe
      by (cases p, simp split: if-splits)
    hence th1: ∀ x. poly p x ≠ 0 by simp
    from k dp have q ^ (degree p) = p * [:1/k:]
```

```
      by (simp add: one-poly-def)
     hence th2: p dvd (q ^ (degree p)) ..
     from th1 th2 pe have ?thesis by blast}
   moreover
   {assume dp: degree p ≠ 0
    then obtain n where n: degree p = Suc n  by (cases degree p, auto)
    {assume p dvd (q ^ (Suc n))
     then obtain u where u: q ^ (Suc n) = p * u ..
     {fix x assume h: poly p x = 0 poly q x ≠ 0
      hence poly (q ^ (Suc n)) x ≠ 0 by simp
      hence False using u h(1) by (simp only: poly-mult) simp}}
     with n nullstellensatz-lemma[of p q degree p] dp
     have ?thesis by auto}
   ultimately have ?thesis by blast}
 ultimately show ?thesis by blast
qed
```

Useful lemma

```
lemma constant-degree:
 fixes p :: 'a::{idom,ring-char-0} poly
 shows constant (poly p) ⟷ degree p = 0 (is ?lhs = ?rhs)
proof
 assume l: ?lhs
 from l[unfolded constant-def, rule-format, of - 0]
 have th: poly p = poly [:poly p 0:] apply - by (rule ext, simp)
 then have p = [:poly p 0:] by (simp add: poly-eq-iff)
 then have degree p = degree [:poly p 0:] by simp
 then show ?rhs by simp
next
 assume r: ?rhs
 then obtain k where p = [:k:]
  by (cases p, simp split: if-splits)
 then show ?lhs unfolding constant-def by auto
qed

lemma divides-degree: assumes pq: p dvd (q:: complex poly)
 shows degree p ≤ degree q ∨ q = 0
apply (cases q = 0, simp-all)
apply (erule dvd-imp-degree-le [OF pq])
done



lemma mpoly-base-conv:
 (0::complex) ≡ poly 0 x c ≡ poly [:c:] x x ≡ poly [:0,1:] x by simp-all

lemma mpoly-norm-conv:
 poly [:0:] (x::complex) ≡ poly 0 x poly [:poly 0 y:] x ≡ poly 0 x by simp-all
```

lemma *mpoly-sub-conv*:
poly p (*x*::*complex*) − *poly q x* ≡ *poly p x* + −*1* ∗ *poly q x*
by (*simp add*: *diff-def*)

lemma *poly-pad-rule*: *poly p x* = *0* ==> *poly* (*pCons 0 p*) *x* = (*0*::*complex*) **by** *simp*

lemma *poly-cancel-eq-conv*: *p* = (*0*::*complex*) ⟹ *a* ≠ *0* ⟹ (*q* = *0*) ≡ (*a* ∗ *q* − *b* ∗ *p* = *0*) **apply** (*atomize* (*full*)) **by** *auto*

lemma *resolve-eq-raw*: *poly 0 x* ≡ *0 poly* [:*c*:] *x* ≡ (*c*::*complex*) **by** *auto*
lemma *resolve-eq-then*: (*P* ⟹ (*Q* ≡ *Q1*)) ⟹ (¬*P* ⟹ (*Q* ≡ *Q2*))
⟹ *Q* ≡ *P* ∧ *Q1* ∨ ¬*P*∧ *Q2* **apply** (*atomize* (*full*)) **by** *blast*

lemma *poly-divides-pad-rule*:
fixes *p q* :: *complex poly*
assumes *pq*: *p dvd q*
shows *p dvd* (*pCons* (*0*::*complex*) *q*)
proof−
have *pCons 0 q* = *q* ∗ [:*0*,*1*:] **by** *simp*
then have *q dvd* (*pCons 0 q*) **..**
with *pq* **show** *?thesis* **by** (*rule dvd-trans*)
qed

lemma *poly-divides-pad-const-rule*:
fixes *p q* :: *complex poly*
assumes *pq*: *p dvd q*
shows *p dvd* (*smult a q*)
proof−
have *smult a q* = *q* ∗ [:*a*:] **by** *simp*
then have *q dvd smult a q* **..**
with *pq* **show** *?thesis* **by** (*rule dvd-trans*)
qed

lemma *poly-divides-conv0*:
fixes *p* :: *complex poly*
assumes *lgpq*: *degree q* < *degree p* **and** *lq*:*p* ≠ *0*
shows *p dvd q* ≡ *q* = *0* (**is** *?lhs* ≡ *?rhs*)
proof−
{assume *r*: *?rhs*
hence *q* = *p* ∗ *0* **by** *simp*
hence *?lhs* **..}**
moreover
{assume *l*: *?lhs*
{assume *q0*: *q* = *0*
hence *?rhs* **by** *simp***}**
moreover
{assume *q0*: *q* ≠ *0*

```
    from l q0 have degree p ≤ degree q
      by (rule dvd-imp-degree-le)
    with lgpq have ?rhs by simp }
  ultimately have ?rhs by blast }
 ultimately show ?lhs ≡ ?rhs by − (atomize (full), blast)
qed

lemma poly-divides-conv1:
 assumes a0: a≠ (0::complex) and pp': (p::complex poly) dvd p'
 and qrp': smult a q − p' ≡ r
 shows p dvd q ≡ p dvd (r::complex poly) (is ?lhs ≡ ?rhs)
proof−
 {
 from pp' obtain t where t: p' = p * t ..
 {assume l: ?lhs
  then obtain u where u: q = p * u ..
   have r = p * (smult a u − t)
    using u qrp' [symmetric] t by (simp add: algebra-simps mult-smult-right)
   then have ?rhs ..}
 moreover
 {assume r: ?rhs
  then obtain u where u: r = p * u ..
  from u [symmetric] t qrp' [symmetric] a0
  have q = p * smult (1/a) (u + t)
    by (simp add: algebra-simps mult-smult-right smult-smult)
  hence ?lhs ..}
 ultimately have ?lhs = ?rhs by blast }
thus ?lhs ≡ ?rhs  by − (atomize(full), blast)
qed

lemma basic-cqe-conv1:
 (∃x. poly p x = 0 ∧ poly 0 x ≠ 0) ≡ False
 (∃x. poly 0 x ≠ 0) ≡ False
 (∃x. poly [:c:] x ≠ 0) ≡ c≠0
 (∃x. poly 0 x = 0) ≡ True
 (∃x. poly [:c:] x = 0) ≡ c = 0 by simp-all

lemma basic-cqe-conv2:
 assumes l:p ≠ 0
 shows (∃x. poly (pCons a (pCons b p)) x = (0::complex)) ≡ True
proof−
 {fix h t
  assume h: h≠0 t=0  pCons a (pCons b p) = pCons h t
  with l have False by simp}
 hence th: ¬ (∃ h t. h≠0 ∧ t=0 ∧ pCons a (pCons b p) = pCons h t)
  by blast
 from fundamental-theorem-of-algebra-alt[OF th]
 show (∃x. poly (pCons a (pCons b p)) x = (0::complex)) ≡ True by auto
qed
```

lemma *basic-cqe-conv-2b*: $(\exists\, x.\ poly\ p\ x \neq (0::complex)) \equiv (p \neq 0)$
proof −
have $p = 0 \longleftrightarrow poly\ p = poly\ 0$
by (*simp add*: *poly-zero*)
also have $\ldots \longleftrightarrow (\neg\ (\exists\, x.\ poly\ p\ x \neq 0))$ **by** (*auto intro*: *ext*)
finally show $(\exists\, x.\ poly\ p\ x \neq (0::complex)) \equiv p \neq 0$
by − (*atomize* (*full*), *blast*)
qed

lemma *basic-cqe-conv3*:
fixes $p\ q$ :: *complex poly*
assumes *l*: $p \neq 0$
shows $(\exists\, x.\ poly\ (pCons\ a\ p)\ x = 0 \wedge poly\ q\ x \neq 0) \equiv \neg\ ((pCons\ a\ p)\ dvd\ (q\ \hat{}\ (psize\ p)))$
proof −
from *l* **have** *dp*:*degree* $(pCons\ a\ p) = psize\ p$ **by** (*simp add*: *psize-def*)
from *nullstellensatz-univariate*[*of pCons a p q*] *l*
show $(\exists\, x.\ poly\ (pCons\ a\ p)\ x = 0 \wedge poly\ q\ x \neq 0) \equiv \neg\ ((pCons\ a\ p)\ dvd\ (q\ \hat{}\ (psize\ p)))$
unfolding *dp*
by − (*atomize* (*full*), *auto*)
qed

lemma *basic-cqe-conv4*:
fixes $p\ q$ :: *complex poly*
assumes *h*: $\bigwedge x.\ poly\ (q\ \hat{}\ n)\ x \equiv poly\ r\ x$
shows $p\ dvd\ (q\ \hat{}\ n) \equiv p\ dvd\ r$
proof −
from *h* **have** $poly\ (q\ \hat{}\ n) = poly\ r$ **by** (*auto intro*: *ext*)
then have $(q\ \hat{}\ n) = r$ **by** (*simp add*: *poly-eq-iff*)
thus $p\ dvd\ (q\ \hat{}\ n) \equiv p\ dvd\ r$ **by** *simp*
qed

lemma *pmult-Cons-Cons*: $(pCons\ (a::complex)\ (pCons\ b\ p) * q = (smult\ a\ q) + (pCons\ 0\ (pCons\ b\ p * q)))$
by *simp*

lemma *elim-neg-conv*: $-\ z \equiv (-1) * (z::complex)$ **by** *simp*
lemma *eqT-intr*: $PROP\ P \Longrightarrow (True \Longrightarrow PROP\ P\ )\ PROP\ P \Longrightarrow True$ **by** *blast*+
lemma *negate-negate-rule*: $Trueprop\ P \equiv \neg\ P \equiv False$ **by** (*atomize* (*full*), *auto*)

lemma *complex-entire*: $(z::complex) \neq 0 \wedge w \neq 0 \equiv z*w \neq 0$ **by** *simp*
lemma *resolve-eq-ne*: $(P \equiv True) \equiv (\neg P \equiv False)\ (P \equiv False) \equiv (\neg P \equiv True)$
by (*atomize* (*full*)) *simp-all*
lemma *cqe-conv1*: $poly\ 0\ x = 0 \longleftrightarrow True$ **by** *simp*
lemma *cqe-conv2*: $(p \Longrightarrow (q \equiv r)) \equiv ((p \wedge q) \equiv (p \wedge r))$ (**is** *?l* ≡ *?r*)
proof

assume $p \Longrightarrow q \equiv r$ **thus** $p \wedge q \equiv p \wedge r$ **apply** $-$ **apply** (*atomize* (*full*)) **by** *blast*
next
assume $p \wedge q \equiv p \wedge r$ p
thus $q \equiv r$ **apply** $-$ **apply** (*atomize* (*full*)) **apply** *blast* **done**
qed
lemma *poly-const-conv*: *poly* [:c:] (x::*complex*) $= y \longleftrightarrow c = y$ **by** *simp*

end

44 Infinite-Set: Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

44.1 Infinite Sets

Some elementary facts about infinite sets, mostly by Stefan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

abbreviation
infinite :: $'a$ *set* $\Rightarrow$ *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-Un: infinite (S ∪ T) ⟷ infinite S ∨ infinite 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** $S \subseteq \{..<k\}$ **..**
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 $\neg$ *?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*) = (∀ *m*. ∃ *n*. *m*≤*n* ∧ *n*∈*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*: *infinite* (*UNIV* :: *nat set*)
by (*auto simp add*: *infinite-nat-iff-unbounded*)

lemma *nat-not-finite*: *finite* (*UNIV*::*nat set*) $\Longrightarrow$ *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* $\Rightarrow$ *'a*) $\Longrightarrow$ *infinite* (*range f*)
proof
assume *finite* (*range f*) **and** *inj f*
then have *finite* (*UNIV*::*nat set*)
by (*rule finite-imageD*)
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* $\subseteq$ (*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 ≡ nat-rec S (λn T. T − {SOME e. e ∈ T})
  def pick ≡ λn. (SOME e. e ∈ Sseq n)
  have Sseq-inf: ⋀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: ⋀n. Sseq n ⊆ S
  proof −
    fix n
    show Sseq n ⊆ S
      by (induct n) (auto simp add: Sseq-def)
  qed
  have Sseq-pick: ⋀n. pick n ∈ Sseq n
  proof −
    fix n
    show pick n ∈ Sseq n
    proof (unfold pick-def, rule someI-ex)
      from Sseq-inf have infinite (Sseq n) .
      then have Sseq n ≠ {} by auto
      then show ∃x. x ∈ Sseq n by auto
    qed
  qed
  with Sseq-S have rng: range pick ⊆ S
    by auto
  have pick-Sseq-gt: ⋀n m. pick n ∉ Sseq (n + Suc m)
  proof −
    fix n m
    show pick n ∉ Sseq (n + Suc m)
      by (induct m) (auto simp add: Sseq-def pick-def)
  qed
  have pick-pick: ⋀n m. pick n ≠ pick (n + Suc m)
  proof −
    fix n m
```

```
    from Sseq-pick have pick (n + Suc m) ∈ Sseq (n + Suc m) .
    moreover from pick-Sseq-gt
    have pick n ∉ Sseq (n + Suc m) .
    ultimately show pick n ≠ 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 ≠ pick j
    proof
      assume eq: pick i = pick j
      from ‹i < j› obtain k where j = i + Suc k
        by (auto simp add: less-iff-Suc-add)
      with pick-pick have pick i ≠ 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 = (∃f. inj (f::nat ⇒ 'a) ∧ range f ⊆ 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 ∃ y ∈ f‘A. infinite (f −‘ {y})
proof (rule ccontr)
  assume ¬ ?thesis
  with img have finite (UN y:f‘A. f −‘ {y}) by (blast intro: finite-UN-I)
  moreover have A ⊆ (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 ∈ f‘A and infinite (f −‘ {y})
  using assms by (blast dest: inf-img-fin-dom)
```

44.2 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

definition
Inf-many :: (*'a* ⇒ *bool*) ⇒ *bool* (**binder** *INFM* *10*) **where**
Inf-many P = *infinite* {*x*. *P x*}

definition
Alm-all :: (*'a* ⇒ *bool*) ⇒ *bool* (**binder** *MOST* *10*) **where**
Alm-all P = (¬ (*INFM x*. ¬ *P x*))

notation (*xsymbols*)
Inf-many (**binder** ∃$_\infty$ *10*) **and**
Alm-all (**binder** ∀$_\infty$ *10*)

notation (*HTML* **output**)
Inf-many (**binder** ∃$_\infty$ *10*) **and**
Alm-all (**binder** ∀$_\infty$ *10*)

lemma *INFM-iff-infinite*: (*INFM x*. *P x*) ⟷ *infinite* {*x*. *P x*}
unfolding *Inf-many-def* **..**

lemma *MOST-iff-cofinite*: (*MOST x*. *P x*) ⟷ *finite* {*x*. ¬ *P x*}
unfolding *Alm-all-def Inf-many-def* **by** *simp*

lemmas *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

lemma *not-INFM* [*simp*]: ¬ (*INFM x*. *P x*) ⟷ (*MOST x*. ¬ *P x*)
unfolding *Alm-all-def not-not* **..**

lemma *not-MOST* [*simp*]: ¬ (*MOST x*. *P x*) ⟷ (*INFM x*. ¬ *P x*)
unfolding *Alm-all-def not-not* **..**

lemma *INFM-const* [*simp*]: (*INFM x*::*'a*. *P*) ⟷ *P* ∧ *infinite* (*UNIV*::*'a set*)
unfolding *Inf-many-def* **by** *simp*

lemma *MOST-const* [*simp*]: (*MOST x*::*'a*. *P*) ⟷ *P* ∨ *finite* (*UNIV*::*'a set*)
unfolding *Alm-all-def* **by** *simp*

lemma *INFM-EX*: (∃$_\infty$*x*. *P x*) ⟹ (∃ *x*. *P x*)
by (*erule contrapos-pp*, *simp*)

lemma *ALL-MOST*: ∀ *x*. *P x* ⟹ ∀$_\infty$*x*. *P x*
by *simp*

lemma *INFM-E*: **assumes** *INFM x*. *P x* **obtains** *x* **where** *P x*

using *INFM-EX* [*OF assms*] **by** (*rule exE*)

lemma *MOST-I*: **assumes** $\bigwedge x.\ P\ x$ **shows** *MOST* $x.\ P\ x$
using *assms* **by** *simp*

lemma *INFM-mono*:
assumes *inf*: $\exists_\infty x.\ P\ x$ **and** *q*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\exists_\infty x.\ Q\ x$
proof −
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*: *INFM-mono*)

lemma *INFM-disj-distrib*:
$(\exists_\infty x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_\infty x.\ P\ x) \vee (\exists_\infty x.\ Q\ x)$
unfolding *Inf-many-def* **by** (*simp add*: *Collect-disj-eq*)

lemma *INFM-imp-distrib*:
$(INFM\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow ((MOST\ x.\ P\ x) \longrightarrow (INFM\ x.\ Q\ x))$
by (*simp only*: *imp-conv-disj INFM-disj-distrib not-MOST*)

lemma *MOST-conj-distrib*:
$(\forall_\infty x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall_\infty x.\ P\ x) \wedge (\forall_\infty x.\ Q\ x)$
unfolding *Alm-all-def* **by** (*simp add*: *INFM-disj-distrib del*: *disj-not1*)

lemma *MOST-conjI*:
$MOST\ x.\ P\ x \Longrightarrow MOST\ x.\ Q\ x \Longrightarrow MOST\ x.\ P\ x \wedge Q\ x$
by (*simp add*: *MOST-conj-distrib*)

lemma *INFM-conjI*:
$INFM\ x.\ P\ x \Longrightarrow MOST\ x.\ Q\ x \Longrightarrow INFM\ x.\ P\ x \wedge Q\ x$
unfolding *MOST-iff-cofinite INFM-iff-infinite*
apply (*drule* (*1*) *Diff-infinite-finite*)
apply (*simp add*: *Collect-conj-eq Collect-neg-eq*)
done

lemma *MOST-rev-mp*:
assumes $\forall_\infty x.\ P\ x$ **and** $\forall_\infty x.\ P\ x \longrightarrow Q\ x$
shows $\forall_\infty x.\ Q\ x$
proof −
have $\forall_\infty x.\ P\ x \wedge (P\ x \longrightarrow Q\ x)$
using *assms* **by** (*rule MOST-conjI*)
thus *?thesis* **by** (*rule MOST-mono*) *simp*
qed

lemma *MOST-imp-iff*:
assumes $MOST\ x.\ P\ x$
shows $(MOST\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow (MOST\ x.\ Q\ x)$
proof
assume $MOST\ x.\ P\ x \longrightarrow Q\ x$
with *assms* **show** $MOST\ x.\ Q\ x$ **by** (*rule MOST-rev-mp*)
next
assume $MOST\ x.\ Q\ x$
then show $MOST\ x.\ P\ x \longrightarrow Q\ x$ **by** (*rule MOST-mono*) *simp*
qed

lemma *INFM-MOST-simps* [*simp*]:
$\bigwedge P\ Q.\ (INFM\ x.\ P\ x \wedge Q) \longleftrightarrow (INFM\ x.\ P\ x) \wedge Q$
$\bigwedge P\ Q.\ (INFM\ x.\ P \wedge Q\ x) \longleftrightarrow P \wedge (INFM\ x.\ Q\ x)$
$\bigwedge P\ Q.\ (MOST\ x.\ P\ x \vee Q) \longleftrightarrow (MOST\ x.\ P\ x) \vee Q$
$\bigwedge P\ Q.\ (MOST\ x.\ P \vee Q\ x) \longleftrightarrow P \vee (MOST\ x.\ Q\ x)$
$\bigwedge P\ Q.\ (MOST\ x.\ P\ x \longrightarrow Q) \longleftrightarrow ((INFM\ x.\ P\ x) \longrightarrow Q)$
$\bigwedge P\ Q.\ (MOST\ x.\ P \longrightarrow Q\ x) \longleftrightarrow (P \longrightarrow (MOST\ x.\ Q\ x))$
unfolding *Alm-all-def Inf-many-def*
by (*simp-all add*: *Collect-conj-eq*)

Properties of quantifiers with injective functions.

lemma *INFM-inj*:
$INFM\ x.\ P\ (f\ x) \Longrightarrow inj\ f \Longrightarrow INFM\ x.\ P\ x$
unfolding *INFM-iff-infinite*
by (*clarify*, *drule* (*1*) *finite-vimageI*, *simp*)

lemma *MOST-inj*:
$MOST\ x.\ P\ x \Longrightarrow inj\ f \Longrightarrow MOST\ x.\ P\ (f\ x)$
unfolding *MOST-iff-cofinite*
by (*drule* (*1*) *finite-vimageI*, *simp*)

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
$\neg\ (INFM\ x.\ x = a)$
$\neg\ (INFM\ x.\ a = x)$
unfolding *INFM-iff-infinite* **by** *simp-all*

lemma *MOST-neq* [*simp*]:
$MOST\ x.\ x \neq a$
$MOST\ x.\ a \neq x$
unfolding *MOST-iff-cofinite* **by** *simp-all*

lemma *INFM-neq* [*simp*]:
$(INFM\ x{::}'a.\ x \neq a) \longleftrightarrow infinite\ (UNIV{::}'a\ set)$
$(INFM\ x{::}'a.\ a \neq x) \longleftrightarrow infinite\ (UNIV{::}'a\ set)$
unfolding *INFM-iff-infinite* **by** *simp-all*

lemma *MOST-eq* [*simp*]:
(*MOST x*::'*a*. *x* = *a*) ⟷ *finite* (*UNIV*::'*a set*)
(*MOST x*::'*a*. *a* = *x*) ⟷ *finite* (*UNIV*::'*a set*)
unfolding *MOST-iff-cofinite* **by** *simp-all*

lemma *MOST-eq-imp*:
MOST x. *x* = *a* ⟶ *P x*
MOST x. *a* = *x* ⟶ *P x*
unfolding *MOST-iff-cofinite* **by** *simp-all*

Properties of quantifiers over the naturals.

lemma *INFM-nat*: (∃ ∞*n*. *P* (*n*::*nat*)) = (∀ *m*. ∃ *n*. *m*<*n* ∧ *P n*)
by (*simp add*: *Inf-many-def infinite-nat-iff-unbounded*)

lemma *INFM-nat-le*: (∃ ∞*n*. *P* (*n*::*nat*)) = (∀ *m*. ∃ *n*. *m*≤*n* ∧ *P n*)
by (*simp add*: *Inf-many-def infinite-nat-iff-unbounded-le*)

lemma *MOST-nat*: (∀ ∞*n*. *P* (*n*::*nat*)) = (∃ *m*. ∀ *n*. *m*<*n* ⟶ *P n*)
by (*simp add*: *Alm-all-def INFM-nat*)

lemma *MOST-nat-le*: (∀ ∞*n*. *P* (*n*::*nat*)) = (∃ *m*. ∀ *n*. *m*≤*n* ⟶ *P n*)
by (*simp add*: *Alm-all-def INFM-nat-le*)

44.3 Enumeration of an Infinite Set

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

primrec (**in** *wellorder*) *enumerate* :: '*a set* ⇒ *nat* ⇒ '*a* **where**
enumerate-0: *enumerate S 0* = (*LEAST n*. *n* ∈ *S*)
| *enumerate-Suc*: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*LEAST n*. *n* ∈ *S*}) *n*

lemma *enumerate-Suc'*:
enumerate S (*Suc n*) = *enumerate* (*S* − {*enumerate S 0*}) *n*
by *simp*

lemma *enumerate-in-set*: *infinite S* ⟹ *enumerate S n* : *S*
apply (*induct n arbitrary*: *S*)
apply (*fastsimp intro*: *LeastI dest*!: *infinite-imp-nonempty*)
apply *simp*
apply (*metis Collect-def Collect-mem-eq DiffE infinite-remove*)
done

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

lemma *enumerate-step*: *infinite S* ⟹ *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* ⟹ *infinite S* ⟹ *enumerate S m* < *enumerate S n*
apply (*erule less-Suc-induct*)
apply (*auto intro*: *enumerate-step*)
done

44.4 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

definition
atmost-one :: *'a set* ⇒ *bool* **where**
atmost-one S = (∀ *x y*. *x*∈*S* ∧ *y*∈*S* ⟶ *x*=*y*)

lemma *atmost-one-empty*: *S* = {} ⟹ *atmost-one S*
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

45 Lattice-Syntax: Pretty syntax for lattice operations

theory *Lattice-Syntax*
imports *Complete-Lattice*
begin

notation
top (⊤) **and**
bot (⊥) **and**
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅- [*900*] *900*) **and**
Sup (⨆- [*900*] *900*)

end

46 ListVector: Lists as vectors

theory *ListVector*
imports *List Main*
begin

A vector-space like structure of lists and arithmetic operations on them. Is only a vector space if restricted to lists of the same length.

Multiplication with a scalar:

abbreviation *scale* :: ('*a*::*times*) ⇒ '*a list* ⇒ '*a list* (**infix** $*_s$ *70*)
where $x *_s xs \equiv map\ (op * x)\ xs$

lemma *scale1*[*simp*]: (*1*::'*a*::*monoid-mult*) $*_s$ *xs* = *xs*
by (*induct xs*) *simp-all*

46.1 + and −

fun *zipwith0* :: ('*a*::*zero* ⇒ '*b*::*zero* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list*
where
zipwith0 f [] [] = [] |
zipwith0 f (*x*#*xs*) (*y*#*ys*) = *f x y* # *zipwith0 f xs ys* |
zipwith0 f (*x*#*xs*) [] = *f x 0* # *zipwith0 f xs* [] |
zipwith0 f [] (*y*#*ys*) = *f 0 y* # *zipwith0 f* [] *ys*

instantiation *list* :: ({*zero*, *plus*}) *plus*
begin

definition
list-add-def: *op* + = *zipwith0* (*op* +)

instance ..

end

instantiation *list* :: ({*zero*, *uminus*}) *uminus*
begin

definition
list-uminus-def: *uminus* = *map uminus*

instance ..

end

instantiation *list* :: ({*zero*,*minus*}) *minus*

begin

definition
list-diff-def: *op −* = *zipwith0* (*op −*)

instance ..

end

lemma *zipwith0-Nil*[*simp*]: *zipwith0 f* [] *ys* = *map* (*f 0*) *ys*
by(*induct ys*) *simp-all*

lemma *list-add-Nil*[*simp*]: [] + *xs* = (*xs*::*'a*::*monoid-add list*)
by (*induct xs*) (*auto simp*:*list-add-def*)

lemma *list-add-Nil2*[*simp*]: *xs* + [] = (*xs*::*'a*::*monoid-add list*)
by (*induct xs*) (*auto simp*:*list-add-def*)

lemma *list-add-Cons*[*simp*]: (*x*#*xs*) + (*y*#*ys*) = (*x*+*y*)#(*xs*+*ys*)
by(*auto simp*:*list-add-def*)

lemma *list-diff-Nil*[*simp*]: [] − *xs* = −(*xs*::*'a*::*group-add list*)
by (*induct xs*) (*auto simp*:*list-diff-def list-uminus-def*)

lemma *list-diff-Nil2*[*simp*]: *xs* − [] = (*xs*::*'a*::*group-add list*)
by (*induct xs*) (*auto simp*:*list-diff-def*)

lemma *list-diff-Cons-Cons*[*simp*]: (*x*#*xs*) − (*y*#*ys*) = (*x*−*y*)#(*xs*−*ys*)
by (*induct xs*) (*auto simp*:*list-diff-def*)

lemma *list-uminus-Cons*[*simp*]: −(*x*#*xs*) = (−*x*)#(−*xs*)
by (*induct xs*) (*auto simp*:*list-uminus-def*)

lemma *self-list-diff*:
xs − *xs* = *replicate* (*length*(*xs*::*'a*::*group-add list*)) *0*
by(*induct xs*) *simp-all*

lemma *list-add-assoc*: **fixes** *xs* :: *'a*::*monoid-add list*
shows (*xs*+*ys*)+*zs* = *xs*+(*ys*+*zs*)
apply(*induct xs arbitrary*: *ys zs*)
apply *simp*
apply(*case-tac ys*)
apply(*simp*)
apply(*simp*)
apply(*case-tac zs*)
apply(*simp*)
apply(*simp add*:*add-assoc*)
done

46.2 Inner product

definition *iprod* :: *'a::ring list* ⇒ *'a list* ⇒ *'a* (⟨-,-⟩) **where**
⟨*xs*,*ys*⟩ = (∑ (*x*,*y*) ← *zip xs ys*. *x*∗*y*)

lemma *iprod-Nil*[*simp*]: ⟨[],*ys*⟩ = *0*
by(*simp add*:*iprod-def*)

lemma *iprod-Nil2*[*simp*]: ⟨*xs*,[]⟩ = *0*
by(*simp add*:*iprod-def*)

lemma *iprod-Cons*[*simp*]: ⟨*x*#*xs*,*y*#*ys*⟩ = *x*∗*y* + ⟨*xs*,*ys*⟩
by(*simp add*:*iprod-def*)

lemma *iprod0-if-coeffs0*: ∀ *c*∈*set cs*. *c* = *0* ⟹ ⟨*cs*,*xs*⟩ = *0*
apply(*induct cs arbitrary*:*xs*)
apply *simp*
apply(*case-tac xs*) **apply** *simp*
apply *auto*
done

lemma *iprod-uminus*[*simp*]: ⟨−*xs*,*ys*⟩ = −⟨*xs*,*ys*⟩
by(*simp add*: *iprod-def uminus-listsum-map o-def split-def map-zip-map list-uminus-def*)

lemma *iprod-left-add-distrib*: ⟨*xs* + *ys*,*zs*⟩ = ⟨*xs*,*zs*⟩ + ⟨*ys*,*zs*⟩
apply(*induct xs arbitrary*: *ys zs*)
apply (*simp add*: *o-def split-def*)
apply(*case-tac ys*)
apply *simp*
apply(*case-tac zs*)
apply (*simp*)
apply(*simp add*:*left-distrib*)
done

lemma *iprod-left-diff-distrib*: ⟨*xs* − *ys*, *zs*⟩ = ⟨*xs*,*zs*⟩ − ⟨*ys*,*zs*⟩
apply(*induct xs arbitrary*: *ys zs*)
apply (*simp add*: *o-def split-def*)
apply(*case-tac ys*)
apply *simp*
apply(*case-tac zs*)
apply (*simp*)
apply(*simp add*:*left-diff-distrib*)
done

lemma *iprod-assoc*: ⟨*x* ∗$_s$ *xs*, *ys*⟩ = *x* ∗ ⟨*xs*,*ys*⟩
apply(*induct xs arbitrary*: *ys*)
apply *simp*
apply(*case-tac ys*)
apply (*simp*)
apply (*simp add*:*right-distrib mult-assoc*)

done

end

47 Kleene-Algebra: Kleene Algebras

theory *Kleene-Algebra*
imports *Main*
begin

WARNING: This is work in progress. Expect changes in the future.

Various lemmas correspond to entries in a database of theorems about Kleene algebras and related structures maintained by Peter Höfner: see http://www.informatik.uni-augsburg.de/~hoefnepe/kleene_db/lemmas/index.html.

47.1 Preliminaries

A class where addition is idempotent.

class *idem-add* = *plus* +
  assumes *add-idem* [*simp*]: $x + x = x$

A class of idempotent abelian semigroups (written additively).

class *idem-ab-semigroup-add* = *ab-semigroup-add* + *idem-add*
begin

lemma *add-idem2* [*simp*]: $x + (x + y) = x + y$
unfolding *add-assoc*[*symmetric*] **by** *simp*

lemma *add-idem3* [*simp*]: $x + (y + x) = x + y$
by (*simp add*: *add-commute*)

end

A class where order is defined in terms of addition.

class *order-by-add* = *plus* + *ord* +
  assumes *order-def*: $x \leq y \longleftrightarrow x + y = y$
  assumes *strict-order-def*: $x < y \longleftrightarrow x \leq y \wedge \neg\, y \leq x$
begin

lemma *ord-simp* [*simp*]: $x \leq y \Longrightarrow x + y = y$
  unfolding *order-def* **.**

lemma *ord-intro*: $x + y = y \Longrightarrow x \leq y$
  unfolding *order-def* **.**

end

A class of idempotent abelian semigroups (written additively) where order is defined in terms of addition.

class *ordered-idem-ab-semigroup-add* = *idem-ab-semigroup-add* + *order-by-add*
begin

lemma *ord-simp2* [*simp*]: $x \leq y \Longrightarrow y + x = y$
unfolding *order-def add-commute* **.**

subclass *order* **proof**
fix $x\ y\ z :: 'a$
show $x \leq x$
unfolding *order-def* **by** *simp*
show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
unfolding *order-def* **by** (*metis add-assoc*)
show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
unfolding *order-def* **by** (*simp add*: *add-commute*)
show $x < y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$
by (*fact strict-order-def*)
qed

subclass *ordered-ab-semigroup-add* **proof**
fix $a\ b\ c :: 'a$
assume $a \leq b$ **show** $c + a \leq c + b$
proof (*rule ord-intro*)
have $c + a + (c + b) = a + b + c$ **by** (*simp add*: *add-ac*)
also have $\ldots = c + b$ **by** (*simp add*: ‹$a \leq b$› *add-ac*)
finally show $c + a + (c + b) = c + b$ **.**
qed
qed

lemma *plus-leI* [*simp*]:
$x \leq z \Longrightarrow y \leq z \Longrightarrow x + y \leq z$
unfolding *order-def* **by** (*simp add*: *add-assoc*)

lemma *less-add* [*simp*]: $x \leq x + y\ y \leq x + y$
unfolding *order-def* **by** (*auto simp*: *add-ac*)

lemma *add-est1* [*elim*]: $x + y \leq z \Longrightarrow x \leq z$
using *less-add*(*1*) **by** (*rule order-trans*)

lemma *add-est2* [*elim*]: $x + y \leq z \Longrightarrow y \leq z$
using *less-add*(*2*) **by** (*rule order-trans*)

lemma *add-supremum*: $(x + y \leq z) = (x \leq z \wedge y \leq z)$
by *auto*

end

A class of commutative monoids (written additively) where order is de-

fined in terms of addition.

class *ordered-comm-monoid-add* = *comm-monoid-add* + *order-by-add*
begin

lemma *zero-minimum* [*simp*]: $0 \leq x$
unfolding *order-def* **by** *simp*

end

A class of idempotent commutative monoids (written additively) where order is defined in terms of addition.

class *ordered-idem-comm-monoid-add* = *ordered-comm-monoid-add* + *idem-add*
begin

subclass *ordered-idem-ab-semigroup-add* **..**

lemma *sum-is-zero*: $(x + y = 0) = (x = 0 \wedge y = 0)$
by (*simp add*: *add-supremum eq-iff*)

end

47.2 A class of Kleene algebras

Class *pre-kleene* provides all operations of Kleene algebras except for the Kleene star.

class *pre-kleene* = *semiring-1* + *idem-add* + *order-by-add*
begin

subclass *ordered-idem-comm-monoid-add* **..**

subclass *ordered-semiring* **proof**
  fix a b c :: $'a$
  assume $a \leq b$

  show $c * a \leq c * b$
  proof (*rule ord-intro*)
    from ‹$a \leq b$› **have** $c * (a + b) = c * b$ **by** *simp*
    thus $c * a + c * b = c * b$ **by** (*simp add*: *right-distrib*)
  qed

  show $a * c \leq b * c$
  proof (*rule ord-intro*)
    from ‹$a \leq b$› **have** $(a + b) * c = b * c$ **by** *simp*
    thus $a * c + b * c = b * c$ **by** (*simp add*: *left-distrib*)
  qed
qed

end

A class that provides a star operator.

class *star* =
fixes *star* :: $'a \Rightarrow 'a$

Finally, a class of Kleene algebras.

class *kleene* = *pre-kleene* + *star* +
assumes *star1*: $1 + a * star\ a \leq star\ a$
and *star2*: $1 + star\ a * a \leq star\ a$
and *star3*: $a * x \leq x \Longrightarrow star\ a * x \leq x$
and *star4*: $x * a \leq x \Longrightarrow x * star\ a \leq x$
begin

lemma *star3'* [*simp*]:
assumes *a*: $b + a * x \leq x$
shows $star\ a * b \leq x$
by (*metis assms less-add mult-left-mono order-trans star3 zero-minimum*)

lemma *star4'* [*simp*]:
assumes *a*: $b + x * a \leq x$
shows $b * star\ a \leq x$
by (*metis assms less-add mult-right-mono order-trans star4 zero-minimum*)

lemma *star-unfold-left*: $1 + a * star\ a = star\ a$
proof (*rule antisym*, *rule star1*)
have $1 + a * (1 + a * star\ a) \leq 1 + a * star\ a$
by (*metis add-left-mono mult-left-mono star1 zero-minimum*)
with *star3'* **have** $star\ a * 1 \leq 1 + a * star\ a$ **.**
thus $star\ a \leq 1 + a * star\ a$ **by** *simp*
qed

lemma *star-unfold-right*: $1 + star\ a * a = star\ a$
proof (*rule antisym*, *rule star2*)
have $1 + (1 + star\ a * a) * a \leq 1 + star\ a * a$
by (*metis add-left-mono mult-right-mono star2 zero-minimum*)
with *star4'* **have** $1 * star\ a \leq 1 + star\ a * a$ **.**
thus $star\ a \leq 1 + star\ a * a$ **by** *simp*
qed

lemma *star-zero* [*simp*]: $star\ 0 = 1$
by (*fact star-unfold-left*[*of 0*, *simplified*, *symmetric*])

lemma *star-one* [*simp*]: $star\ 1 = 1$
by (*metis add-idem2 eq-iff mult-1-right ord-simp2 star3 star-unfold-left*)

lemma *one-less-star* [*simp*]: $1 \leq star\ x$
by (*metis less-add*(*1*) *star-unfold-left*)

lemma *ka1* [*simp*]: $x * star\ x \leq star\ x$
by (*metis less-add*(*2*) *star-unfold-left*)

lemma *star-mult-idem* [*simp*]: *star x ∗ star x = star x*
by (*metis add-commute add-est1 eq-iff mult-1-right right-distrib star3 star-unfold-left*)

lemma *less-star* [*simp*]: *x ≤ star x*
by (*metis less-add*(*2*) *mult-1-right mult-left-mono one-less-star order-trans star-unfold-left zero-minimum*)

lemma *star-simulation-leq-1*:
assumes *a*: *a ∗ x ≤ x ∗ b*
shows *star a ∗ x ≤ x ∗ star b*
proof (*rule star3'*, *rule order-trans*)
from *a* **have** *a ∗ x ∗ star b ≤ x ∗ b ∗ star b*
by (*rule mult-right-mono*) *simp*
thus *x + a ∗ (x ∗ star b) ≤ x + x ∗ b ∗ star b*
using *add-left-mono* **by** (*auto simp*: *mult-assoc*)
show *. . . ≤ x ∗ star b*
by (*metis add-supremum ka1 mult.right-neutral mult-assoc mult-left-mono one-less-star zero-minimum*)
qed

lemma *star-simulation-leq-2*:
assumes *a*: *x ∗ a ≤ b ∗ x*
shows *x ∗ star a ≤ star b ∗ x*
proof (*rule star4'*, *rule order-trans*)
from *a* **have** *star b ∗ x ∗ a ≤ star b ∗ b ∗ x*
by (*metis mult-assoc mult-left-mono zero-minimum*)
thus *x + star b ∗ x ∗ a ≤ x + star b ∗ b ∗ x*
using *add-mono* **by** *auto*
show *. . . ≤ star b ∗ x*
by (*metis add-supremum left-distrib less-add mult.left-neutral mult-assoc mult-right-mono star-unfold-right zero-minimum*)
qed

lemma *star-simulation* [*simp*]:
assumes *a*: *a ∗ x = x ∗ b*
shows *star a ∗ x = x ∗ star b*
by (*metis antisym assms order-refl star-simulation-leq-1 star-simulation-leq-2*)

lemma *star-slide2* [*simp*]: *star x ∗ x = x ∗ star x*
by (*metis star-simulation*)

lemma *star-idemp* [*simp*]: *star* (*star x*) = *star x*
by (*metis add-idem2 eq-iff less-star mult-1-right star3' star-mult-idem star-unfold-left*)

lemma *star-slide* [*simp*]: *star* (*x ∗ y*) *∗ x = x ∗ star* (*y ∗ x*)
by (*metis mult-assoc star-simulation*)

lemma *star-one'*:

assumes $p * p' = 1$ $p' * p = 1$
shows $p' * star\ a * p = star\ (p' * a * p)$
proof −
from *assms*
have $p' * star\ a * p = p' * star\ (p * p' * a) * p$
by *simp*
also have $\ldots = p' * p * star\ (p' * a * p)$
by (*simp add*: *mult-assoc*)
also have $\ldots = star\ (p' * a * p)$
by (*simp add*: *assms*)
finally show *?thesis* **.**
qed

lemma *x-less-star* [*simp*]: $x \leq x * star\ a$
by (*metis mult.right-neutral mult-left-mono one-less-star zero-minimum*)

lemma *star-mono* [*simp*]: $x \leq y \Longrightarrow star\ x \leq star\ y$
by (*metis add-commute eq-iff less-star ord-simp2 order-trans star3 star4' star-idemp star-mult-idem x-less-star*)

lemma *star-sub*: $x \leq 1 \Longrightarrow star\ x = 1$
by (*metis add-commute ord-simp star-idemp star-mono star-mult-idem star-one star-unfold-left*)

lemma *star-unfold2*: $star\ x * y = y + x * star\ x * y$
by (*subst star-unfold-right*[*symmetric*]) (*simp add*: *mult-assoc left-distrib*)

lemma *star-absorb-one* [*simp*]: $star\ (x + 1) = star\ x$
by (*metis add-commute eq-iff left-distrib less-add mult-1-left mult-assoc star3 star-mono star-mult-idem star-unfold2 x-less-star*)

lemma *star-absorb-one'* [*simp*]: $star\ (1 + x) = star\ x$
by (*subst add-commute*) (*fact star-absorb-one*)

lemma *ka16*: $(y * star\ x) * star\ (y * star\ x) \leq star\ x * star\ (y * star\ x)$
by (*metis ka1 less-add*(*1*) *mult-assoc order-trans star-unfold2*)

lemma *ka16'*: $(star\ x * y) * star\ (star\ x * y) \leq star\ (star\ x * y) * star\ x$
by (*metis ka1 mult-assoc order-trans star-slide x-less-star*)

lemma *ka17*: $(x * star\ x) * star\ (y * star\ x) \leq star\ x * star\ (y * star\ x)$
by (*metis ka1 mult-assoc mult-right-mono zero-minimum*)

lemma *ka18*: $(x * star\ x) * star\ (y * star\ x) + (y * star\ x) * star\ (y * star\ x)$
$\leq star\ x * star\ (y * star\ x)$
by (*metis ka16 ka17 left-distrib mult-assoc plus-leI*)

lemma *star-decomp*: $star\ (x + y) = star\ x * star\ (y * star\ x)$
proof (*rule antisym*)

have $1 + (x + y) * star\ x * star\ (y * star\ x) \leq$
$1 + x * star\ x * star\ (y * star\ x) + y * star\ x * star\ (y * star\ x)$
by (*metis add-commute add-left-commute eq-iff left-distrib mult-assoc*)
also have $\ldots \leq star\ x * star\ (y * star\ x)$
by (*metis add-commute add-est1 add-left-commute ka18 plus-leI star-unfold-left x-less-star*)
finally show $star\ (x + y) \leq star\ x * star\ (y * star\ x)$
by (*metis mult-1-right mult-assoc star3'*)
next
show $star\ x * star\ (y * star\ x) \leq star\ (x + y)$
by (*metis add-assoc add-est1 add-est2 add-left-commute less-star mult-mono' star-absorb-one star-absorb-one' star-idemp star-mono star-mult-idem zero-minimum*)
qed

lemma *ka22*: $y * star\ x \leq star\ x * star\ y \Longrightarrow star\ y * star\ x \leq star\ x * star\ y$
by (*metis mult-assoc mult-right-mono plus-leI star3' star-mult-idem x-less-star zero-minimum*)

lemma *ka23*: $star\ y * star\ x \leq star\ x * star\ y \Longrightarrow y * star\ x \leq star\ x * star\ y$
by (*metis less-star mult-right-mono order-trans zero-minimum*)

lemma *ka24*: $star\ (x + y) \leq star\ (star\ x * star\ y)$
by (*metis add-est1 add-est2 less-add(1) mult-assoc order-def plus-leI star-absorb-one star-mono star-slide2 star-unfold2 star-unfold-left x-less-star*)

lemma *ka25*: $star\ y * star\ x \leq star\ x * star\ y \Longrightarrow star\ (star\ y * star\ x) \leq star\ x * star\ y$
proof −
assume $star\ y * star\ x \leq star\ x * star\ y$
hence $\forall\, x_1.\ star\ y * (star\ x * x_1) \leq star\ x * (star\ y * x_1)$ **by** (*metis mult-assoc mult-right-mono zero-minimum*)
hence $star\ y * (star\ x * star\ y) \leq star\ x * star\ y$ **by** (*metis star-mult-idem*)
hence $\exists\, x_1.\ star\ (star\ y * star\ x) * star\ x_1 \leq star\ x * star\ y$ **by** (*metis star-decomp star-idemp star-simulation-leq-2 star-slide*)
hence $\exists\, x_1 \geq star\ (star\ y * star\ x).\ x_1 \leq star\ x * star\ y$ **by** (*metis x-less-star*)
thus $star\ (star\ y * star\ x) \leq star\ x * star\ y$ **by** (*metis order-trans*)
qed

lemma *church-rosser*:
$star\ y * star\ x \leq star\ x * star\ y \Longrightarrow star\ (x + y) \leq star\ x * star\ y$
by (*metis add-commute ka24 ka25 order-trans*)

lemma *kleene-bubblesort*: $y * x \leq x * y \Longrightarrow star\ (x + y) \leq star\ x * star\ y$
by (*metis church-rosser star-simulation-leq-1 star-simulation-leq-2*)

lemma *ka27*: $star\ (x + star\ y) = star\ (x + y)$
by (*metis add-commute star-decomp star-idemp*)

lemma *ka28*: $star\ (star\ x + star\ y) = star\ (x + y)$

by (*metis add-commute ka27*)

lemma *ka29*: $(y * (1 + x) \leq (1 + x) * star\ y) = (y * x \leq (1 + x) * star\ y)$
by (*metis add-supremum left-distrib less-add(1) less-star mult.left-neutral mult.right-neutral order-trans right-distrib*)

lemma *ka30*: $star\ x * star\ y \leq star\ (x + y)$
by (*metis mult-left-mono star-decomp star-mono x-less-star zero-minimum*)

lemma *simple-simulation*: $x * y = 0 \Longrightarrow star\ x * y = y$
by (*metis mult.right-neutral mult-zero-right star-simulation star-zero*)

lemma *ka32*: $star\ (x * y) = 1 + x * star\ (y * x) * y$
by (*metis mult-assoc star-slide star-unfold-left*)

lemma *ka33*: $x * y + 1 \leq y \Longrightarrow star\ x \leq y$
by (*metis add-commute mult.right-neutral star3'*)

end

47.3 Complete lattices are Kleene algebras

lemma (**in** *complete-lattice*) *le-SUPI'*:
  assumes $l \leq M\ i$
  shows $l \leq (SUP\ i.\ M\ i)$
  using *assms* **by** (*rule order-trans*) (*rule le-SUPI* [*OF UNIV-I*])

class *kleene-by-complete-lattice* = *pre-kleene*
  + *complete-lattice* + *power* + *star* +
  assumes *star-cont*: $a * star\ b * c = SUPR\ UNIV\ (\lambda n.\ a * b\ \hat{}\ n * c)$
begin

subclass *kleene*
proof
  fix $a\ x :: 'a$

  have [*simp*]: $1 \leq star\ a$
    unfolding *star-cont*[*of 1 a 1*, *simplified*]
    by (*subst power-0*[*symmetric*]) (*rule le-SUPI* [*OF UNIV-I*])

  show $1 + a * star\ a \leq star\ a$
    apply (*rule plus-leI*, *simp*)
    apply (*simp add*:*star-cont*[*of a a 1*, *simplified*])
    apply (*simp add*:*star-cont*[*of 1 a 1*, *simplified*])
    apply (*subst power-Suc*[*symmetric*])
    by (*intro SUP-leI le-SUPI UNIV-I*)

  show $1 + star\ a * a \leq star\ a$
    apply (*rule plus-leI*, *simp*)

```
    apply (simp add:star-cont[of 1 a a, simplified])
    apply (simp add:star-cont[of 1 a 1, simplified])
   by (auto intro: SUP-leI le-SUPI simp add: power-Suc[symmetric] power-commutes
simp del: power-Suc)

  show a * x ≤ x ⟹ star a * x ≤ x
  proof −
    assume a: a * x ≤ x

    {
      fix n
      have a ^ (Suc n) * x ≤ a ^ n * x
      proof (induct n)
        case 0 thus ?case by (simp add: a)
      next
        case (Suc n)
        hence a * (a ^ Suc n * x) ≤ a * (a ^ n * x)
          by (auto intro: mult-mono)
        thus ?case
          by (simp add: mult-assoc)
      qed
    }
    note a = this

    {
      fix n have a ^ n * x ≤ x
      proof (induct n)
        case 0 show ?case by simp
      next
        case (Suc n) with a[of n]
        show ?case by simp
      qed
    }
    note b = this

    show star a * x ≤ x
      unfolding star-cont[of 1 a x, simplified]
      by (rule SUP-leI) (rule b)
  qed

  show x * a ≤ x ⟹ x * star a ≤ x
  proof −
    assume a: x * a ≤ x

    {
      fix n
      have x * a ^ (Suc n) ≤ x * a ^ n
      proof (induct n)
        case 0 thus ?case by (simp add: a)
```

```
    next
      case (Suc n)
      hence (x * a ^ Suc n) * a ≤ (x * a ^ n) * a
        by (auto intro: mult-mono)
      thus ?case
        by (simp add: power-commutes mult-assoc)
    qed
  }
  note a = this

  {
    fix n have x * a ^ n ≤ x
    proof (induct n)
      case 0 show ?case by simp
    next
      case (Suc n) with a[of n]
      show ?case by simp
    qed
  }
  note b = this

  show x * star a ≤ x
    unfolding star-cont[of x a 1, simplified]
    by (rule SUP-leI) (rule b)
 qed
qed

end
```

47.4 Transitive closure

```
context kleene
begin

definition
  tcl-def: tcl x = star x * x

lemma tcl-zero: tcl 0 = 0
unfolding tcl-def by simp

lemma tcl-unfold-right: tcl a = a + tcl a * a
by (metis star-slide2 star-unfold2 tcl-def)

lemma less-tcl: a ≤ tcl a
by (metis star-slide2 tcl-def x-less-star)

end
```

47.5 A naive algorithm to generate the transitive closure

function (*default* $\lambda x.\ 0$, *tailrec*, *domintros*)
  mk-tcl :: ('*a*::{*plus*,*times*,*ord*,*zero*}) ⇒ '*a* ⇒ '*a*
where
  mk-tcl A X = (*if X* ∗ *A* ≤ *X then X else mk-tcl A* (*X* + *X* ∗ *A*))
  by *pat-completeness simp*

declare *mk-tcl.simps*[*simp del*]

lemma *mk-tcl-code*[*code*]:
  mk-tcl A X =
  (*let XA* = *X* ∗ *A*
  in if XA ≤ *X then X else mk-tcl A* (*X* + *XA*))
  unfolding *mk-tcl.simps*[*of A X*] *Let-def* **..**

context *kleene*
begin

lemma *mk-tcl-lemma1*: (*X* + *X* ∗ *A*) ∗ *star A* = *X* ∗ *star A*
by (*metis ka1 left-distrib mult-assoc mult-left-mono ord-simp2 zero-minimum*)

lemma *mk-tcl-lemma2*: *X* ∗ *A* ≤ *X* ⟹ *X* ∗ *star A* = *X*
by (*rule antisym*) (*auto simp*: *star4*)

end

lemma *mk-tcl-correctness*:
  fixes *X* :: '*a*::*kleene*
  assumes *mk-tcl-dom* (*A*, *X*)
  shows *mk-tcl A X* = *X* ∗ *star A*
  using *assms*
  by *induct* (*auto simp*: *mk-tcl-lemma1 mk-tcl-lemma2*)

lemma *graph-implies-dom*: *mk-tcl-graph x y* ⟹ *mk-tcl-dom x*
  by (*rule mk-tcl-graph.induct*) (*auto intro*:*accp.accI elim*:*mk-tcl-rel.cases*)

lemma *mk-tcl-default*: ¬ *mk-tcl-dom* (*a*,*x*) ⟹ *mk-tcl a x* = *0*
  unfolding *mk-tcl-def*
  by (*rule fundef-default-value*[*OF mk-tcl-sumC-def graph-implies-dom*])

We can replace the dom-Condition of the correctness theorem with something executable:

lemma *mk-tcl-correctness2*:
  fixes *A X* :: '*a* :: {*kleene*}
  assumes *mk-tcl A A* ≠ *0*
  shows *mk-tcl A A* = *tcl A*
  using *assms mk-tcl-default mk-tcl-correctness*
  unfolding *tcl-def*
  by *auto*

end

48 Multiset: (Finite) multisets

theory *Multiset*
imports *Main*
begin

48.1 The type of multisets

typedef *'a multiset* = {*f* :: *'a* => *nat*. *finite* {*x*. *f x* > *0*}}
  morphisms *count Abs-multiset*
proof
  show (λ*x*. *0*::*nat*) ∈ *?multiset* **by** *simp*
qed

lemmas *multiset-typedef* = *Abs-multiset-inverse count-inverse count*

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

notation (*xsymbols*)
  Melem (**infix** ∈# *50*)

lemma *multiset-ext-iff*:
  M = *N* ⟷ (∀ *a*. *count M a* = *count N a*)
  by (*simp only*: *count-inject* [*symmetric*] *expand-fun-eq*)

lemma *multiset-ext*:
  (⋀*x*. *count A x* = *count B x*) ⟹ *A* = *B*
  using *multiset-ext-iff* **by** *auto*

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*:
  (λ*a*. *0*) ∈ *multiset*
  by (*simp add*: *multiset-def*)

lemma *only1-in-multiset*:
  (λ*b*. *if b* = *a then n else 0*) ∈ *multiset*
  by (*simp add*: *multiset-def*)

lemma *union-preserves-multiset*:
  M ∈ *multiset* ⟹ *N* ∈ *multiset* ⟹ (λ*a*. *M a* + *N a*) ∈ *multiset*
  by (*simp add*: *multiset-def*)

lemma *diff-preserves-multiset*:

assumes $M \in multiset$
shows $(\lambda a.\ M\ a - N\ a) \in multiset$
proof −
have $\{x.\ N\ x < M\ x\} \subseteq \{x.\ 0 < M\ x\}$
by *auto*
with *assms* **show** *?thesis*
by (*auto simp add*: *multiset-def intro*: *finite-subset*)
qed

lemma *MCollect-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
proof −
have $\{x.\ (P\ x \longrightarrow 0 < M\ x) \wedge P\ x\} \subseteq \{x.\ 0 < M\ x\}$
by *auto*
with *assms* **show** *?thesis*
by (*auto simp add*: *multiset-def intro*: *finite-subset*)
qed

lemmas *in-multiset* = *const0-in-multiset only1-in-multiset*
union-preserves-multiset diff-preserves-multiset MCollect-preserves-multiset

48.2 Representing multisets

Multiset comprehension

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

syntax
-MCollect :: *pttrn* => *'a multiset* => *bool* => *'a multiset* ((*1*{# - :# -./ -#}))
translations
{#*x* :# *M*. *P*#} == *CONST MCollect M* (λ*x*. *P*)

Multiset enumeration

instantiation *multiset* :: (*type*) {*zero*, *plus*}
begin

definition *Mempty-def*:
0 = *Abs-multiset* (λ*a*. *0*)

abbreviation *Mempty* :: *'a multiset* ({#}) **where**
Mempty ≡ *0*

definition *union-def*:
M + *N* = *Abs-multiset* (λ*a*. *count M a* + *count N a*)

instance ..

end

definition *single* :: *'a => 'a multiset* **where**
single a = Abs-multiset (λb. if b = a then 1 else 0)

syntax
-multiset :: *args => 'a multiset* ({#(-)#})
translations
{#*x*, *xs*#} == {#*x*#} + {#*xs*#}
{#*x*#} == *CONST single x*

lemma *count-empty* [*simp*]: *count* {#} *a = 0*
by (*simp add*: *Mempty-def in-multiset multiset-typedef*)

lemma *count-single* [*simp*]: *count* {#*b*#} *a* = (*if b = a then 1 else 0*)
by (*simp add*: *single-def in-multiset multiset-typedef*)

48.3 Basic operations

48.3.1 Union

lemma *count-union* [*simp*]: *count* (*M* + *N*) *a = count M a + count N a*
by (*simp add*: *union-def in-multiset multiset-typedef*)

instance *multiset* :: (*type*) *cancel-comm-monoid-add* **proof**
qed (*simp-all add*: *multiset-ext-iff*)

48.3.2 Difference

instantiation *multiset* :: (*type*) *minus*
begin

definition *diff-def*:
M − *N* = *Abs-multiset* (λ*a*. *count M a* − *count N a*)

instance ..

end

lemma *count-diff* [*simp*]: *count* (*M* − *N*) *a = count M a* − *count N a*
by (*simp add*: *diff-def in-multiset multiset-typedef*)

lemma *diff-empty* [*simp*]: *M* − {#} = *M* ∧ {#} − *M* = {#}
by(*simp add*: *multiset-ext-iff*)

lemma *diff-cancel*[*simp*]: *A* − *A* = {#}
by (*rule multiset-ext*) *simp*

lemma *diff-union-cancelR* [*simp*]: *M* + *N* − *N* = (*M*::*'a multiset*)
by(*simp add*: *multiset-ext-iff*)

lemma *diff-union-cancelL* [*simp*]: $N + M - N = (M::'a\ multiset)$
by(*simp add*: *multiset-ext-iff*)

lemma *insert-DiffM*:
$x \in\# M \Longrightarrow \{\#x\#\} + (M - \{\#x\#\}) = M$
by (*clarsimp simp*: *multiset-ext-iff*)

lemma *insert-DiffM2* [*simp*]:
$x \in\# M \Longrightarrow M - \{\#x\#\} + \{\#x\#\} = M$
by (*clarsimp simp*: *multiset-ext-iff*)

lemma *diff-right-commute*:
$(M::'a\ multiset) - N - Q = M - Q - N$
by (*auto simp add*: *multiset-ext-iff*)

lemma *diff-add*:
$(M::'a\ multiset) - (N + Q) = M - N - Q$
by (*simp add*: *multiset-ext-iff*)

lemma *diff-union-swap*:
$a \neq b \Longrightarrow M - \{\#a\#\} + \{\#b\#\} = M + \{\#b\#\} - \{\#a\#\}$
by (*auto simp add*: *multiset-ext-iff*)

lemma *diff-union-single-conv*:
$a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
by (*simp add*: *multiset-ext-iff*)

48.3.3 Equality of multisets

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
by (*simp add*: *multiset-ext-iff*)

lemma *single-eq-single* [*simp*]: $\{\#a\#\} = \{\#b\#\} \longleftrightarrow a = b$
by (*auto simp add*: *multiset-ext-iff*)

lemma *union-eq-empty* [*iff*]: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
by (*auto simp add*: *multiset-ext-iff*)

lemma *empty-eq-union* [*iff*]: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
by (*auto simp add*: *multiset-ext-iff*)

lemma *multi-self-add-other-not-self* [*simp*]: $M = M + \{\#x\#\} \longleftrightarrow False$
by (*auto simp add*: *multiset-ext-iff*)

lemma *diff-single-trivial*:
$\neg\ x \in\# M \Longrightarrow M - \{\#x\#\} = M$
by (*auto simp add*: *multiset-ext-iff*)

lemma *diff-single-eq-union*:

$x \in\# M \Longrightarrow M - \{\#x\#\} = N \longleftrightarrow M = N + \{\#x\#\}$
by *auto*

lemma *union-single-eq-diff*:
$M + \{\#x\#\} = N \Longrightarrow M = N - \{\#x\#\}$
by (*auto dest*: *sym*)

lemma *union-single-eq-member*:
$M + \{\#x\#\} = N \Longrightarrow x \in\# N$
by *auto*

lemma *union-is-single*:
$M + N = \{\#a\#\} \longleftrightarrow M = \{\#a\#\} \wedge N=\{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\}$ (**is** *?lhs = ?rhs*)**proof**
assume *?rhs* **then show** *?lhs* **by** *auto*
next
assume *?lhs* **thus** *?rhs*
by(*simp add*: *multiset-ext-iff split*:*if-splits*) (*metis add-is-1*)
qed

lemma *single-is-union*:
$\{\#a\#\} = M + N \longleftrightarrow \{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N$
by (*auto simp add*: *eq-commute* [*of* $\{\#a\#\}$ $M + N$] *union-is-single*)

lemma *add-eq-conv-diff*:
$M + \{\#a\#\} = N + \{\#b\#\} \longleftrightarrow M = N \wedge a = b \vee M = N - \{\#a\#\} + \{\#b\#\} \wedge N = M - \{\#b\#\} + \{\#a\#\}$ (**is** *?lhs = ?rhs*)

proof
assume *?rhs* **then show** *?lhs*
by (*auto simp add*: *add-assoc add-commute* [*of* $\{\#b\#\}$])
(*drule sym*, *simp add*: *add-assoc* [*symmetric*])
next
assume *?lhs*
show *?rhs*
proof (*cases* $a = b$)
case *True* **with** ‹*?lhs*› **show** *?thesis* **by** *simp*
next
case *False*
from ‹*?lhs*› **have** $a \in\# N + \{\#b\#\}$ **by** (*rule union-single-eq-member*)
with *False* **have** $a \in\# N$ **by** *auto*
moreover from ‹*?lhs*› **have** $M = N + \{\#b\#\} - \{\#a\#\}$ **by** (*rule union-single-eq-diff*)
moreover note *False*
ultimately show *?thesis* **by** (*auto simp add*: *diff-right-commute* [*of* - $\{\#a\#\}$] *diff-union-swap*)
qed
qed

lemma *insert-noteq-member*:

assumes *BC*: $B + \{\#b\#\} = C + \{\#c\#\}$
and *bnotc*: $b \neq c$
shows $c \in\# B$
proof −
have $c \in\# C + \{\#c\#\}$ **by** *simp*
have *nc*: $\neg\ c \in\# \{\#b\#\}$ **using** *bnotc* **by** *simp*
then have $c \in\# B + \{\#b\#\}$ **using** *BC* **by** *simp*
then show $c \in\# B$ **using** *nc* **by** *simp*
qed

lemma *add-eq-conv-ex*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee (\exists K.\ M = K + \{\#b\#\} \wedge N = K + \{\#a\#\}))$
by (*auto simp add*: *add-eq-conv-diff*)

48.3.4 Pointwise ordering induced by count

instantiation *multiset* :: (*type*) *ordered-ab-semigroup-add-imp-le*
begin

definition *less-eq-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
mset-le-def: $A \leq B \longleftrightarrow (\forall a.\ count\ A\ a \leq count\ B\ a)$

definition *less-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
mset-less-def: $(A::'a\ multiset) < B \longleftrightarrow A \leq B \wedge A \neq B$

instance proof
qed (*auto simp add*: *mset-le-def mset-less-def multiset-ext-iff intro*: *order-trans antisym*)

end

lemma *mset-less-eqI*:
$(\bigwedge x.\ count\ A\ x \leq count\ B\ x) \Longrightarrow A \leq B$
by (*simp add*: *mset-le-def*)

lemma *mset-le-exists-conv*:
$(A::'a\ multiset) \leq B \longleftrightarrow (\exists C.\ B = A + C)$
apply (*unfold mset-le-def*, *rule iffI*, *rule-tac* $x = B - A$ **in** *exI*)
apply (*auto intro*: *multiset-ext-iff* [*THEN iffD2*])
done

lemma *mset-le-mono-add-right-cancel* [*simp*]:
$(A::'a\ multiset) + C \leq B + C \longleftrightarrow A \leq B$
by (*fact add-le-cancel-right*)

lemma *mset-le-mono-add-left-cancel* [*simp*]:
$C + (A::'a\ multiset) \leq C + B \longleftrightarrow A \leq B$
by (*fact add-le-cancel-left*)

lemma *mset-le-mono-add*:
(*A*::'*a multiset*) ≤ *B* ⟹ *C* ≤ *D* ⟹ *A* + *C* ≤ *B* + *D*
by (*fact add-mono*)

lemma *mset-le-add-left* [*simp*]:
(*A*::'*a multiset*) ≤ *A* + *B*
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-add-right* [*simp*]:
B ≤ (*A*::'*a multiset*) + *B*
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-single*:
a :# *B* ⟹ {#*a*#} ≤ *B*
by (*simp add*: *mset-le-def*)

lemma *multiset-diff-union-assoc*:
C ≤ *B* ⟹ (*A*::'*a multiset*) + *B* − *C* = *A* + (*B* − *C*)
by (*simp add*: *multiset-ext-iff mset-le-def*)

lemma *mset-le-multiset-union-diff-commute*:
B ≤ *A* ⟹ (*A*::'*a multiset*) − *B* + *C* = *A* + *C* − *B*
by (*simp add*: *multiset-ext-iff mset-le-def*)

lemma *mset-lessD*: *A* < *B* ⟹ *x* ∈# *A* ⟹ *x* ∈# *B*
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply (*erule-tac x*=*x* **in** *allE*)
apply *auto*
done

lemma *mset-leD*: *A* ≤ *B* ⟹ *x* ∈# *A* ⟹ *x* ∈# *B*
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply (*erule-tac x* = *x* **in** *allE*)
apply *auto*
done

lemma *mset-less-insertD*: (*A* + {#*x*#} < *B*) ⟹ (*x* ∈# *B* ∧ *A* < *B*)
apply (*rule conjI*)
apply (*simp add*: *mset-lessD*)
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply *safe*
apply (*erule-tac x* = *a* **in** *allE*)
apply (*auto split*: *split-if-asm*)
done

lemma *mset-le-insertD*: (*A* + {#*x*#} ≤ *B*) ⟹ (*x* ∈# *B* ∧ *A* ≤ *B*)
apply (*rule conjI*)
apply (*simp add*: *mset-leD*)

apply (*force simp*: *mset-le-def mset-less-def split*: *split-if-asm*)
done

lemma *mset-less-of-empty*[*simp*]: $A < \{\#\} \longleftrightarrow \mathit{False}$
by (*auto simp add*: *mset-less-def mset-le-def multiset-ext-iff*)

lemma *multi-psub-of-add-self*[*simp*]: $A < A + \{\#x\#\}$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *multi-psub-self*[*simp*]: $(A::'a\ multiset) < A = \mathit{False}$
by *simp*

lemma *mset-less-add-bothsides*:
$T + \{\#x\#\} < S + \{\#x\#\} \Longrightarrow T < S$
by (*fact add-less-imp-less-right*)

lemma *mset-less-empty-nonempty*:
$\{\#\} < S \longleftrightarrow S \neq \{\#\}$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-diff-self*:
$c \in\# B \Longrightarrow B - \{\#c\#\} < B$
by (*auto simp*: *mset-le-def mset-less-def multiset-ext-iff*)

48.3.5 Intersection

instantiation *multiset* :: (*type*) *semilattice-inf*
begin

definition *inf-multiset* :: $'a\ multiset \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ **where**
multiset-inter-def: *inf-multiset A B* $= A - (A - B)$

instance proof −
have *aux*: $\bigwedge m\ n\ q :: nat.\ m \leq n \Longrightarrow m \leq q \Longrightarrow m \leq n - (n - q)$ **by** *arith*
show *OFCLASS*('*a multiset*, *semilattice-inf-class*) **proof**
qed (*auto simp add*: *multiset-inter-def mset-le-def aux*)
qed

end

abbreviation *multiset-inter* :: $'a\ multiset \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ (**infixl** $\#\cap$ *70*) **where**
multiset-inter $\equiv$ *inf*

lemma *multiset-inter-count*:
count $(A\ \#\cap\ B)\ x =$ *min* (*count A x*) (*count B x*)
by (*simp add*: *multiset-inter-def multiset-typedef*)

lemma *multiset-inter-single*: $a \neq b \Longrightarrow \{\#a\#\}\ \#\cap\ \{\#b\#\} = \{\#\}$

by (*rule multiset-ext*) (*auto simp add*: *multiset-inter-count*)

lemma *multiset-union-diff-commute*:
assumes $B \#\cap C = \{\#\}$
shows $A + B - C = A - C + B$
proof (*rule multiset-ext*)
fix x
from *assms* **have** *min* (*count B x*) (*count C x*) = *0*
by (*auto simp add*: *multiset-inter-count multiset-ext-iff*)
then have *count B x* = *0* ∨ *count C x* = *0*
by *auto*
then show *count* (*A* + *B* − *C*) *x* = *count* (*A* − *C* + *B*) *x*
by *auto*
qed

48.3.6 Comprehension (filter)

lemma *count-MCollect* [*simp*]:
count {# *x*:#*M*. *P x* #} *a* = (*if P a then count M a else 0*)
by (*simp add*: *MCollect-def in-multiset multiset-typedef*)

lemma *MCollect-empty* [*simp*]: *MCollect* {#} *P* = {#}
by (*rule multiset-ext*) *simp*

lemma *MCollect-single* [*simp*]:
MCollect {#*x*#} *P* = (*if P x then* {#*x*#} *else* {#})
by (*rule multiset-ext*) *simp*

lemma *MCollect-union* [*simp*]:
MCollect (*M* + *N*) *f* = *MCollect M f* + *MCollect N f*
by (*rule multiset-ext*) *simp*

48.3.7 Set of elements

definition *set-of* :: *'a multiset* => *'a set* **where**
set-of M = {*x*. *x* :# *M*}

lemma *set-of-empty* [*simp*]: *set-of* {#} = {}
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* ∪ *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 multiset-ext-iff*)

lemma *mem-set-of-iff* [*simp*]: (*x* ∈ *set-of M*) = (*x* :# *M*)

by (*auto simp add*: *set-of-def*)

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 *finite-set-of* [*iff*]: *finite* (*set-of M*)
using *count* [*of M*] **by** (*simp add*: *multiset-def set-of-def*)

48.3.8 Size

instantiation *multiset* :: (*type*) *size*
begin

definition *size-def*:
size M = *setsum* (*count M*) (*set-of M*)

instance ..

end

lemma *size-empty* [*simp*]: *size* {#} = *0*
by (*simp add*: *size-def*)

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
by (*simp add*: *size-def*)

lemma *setsum-count-Int*:
finite A ==> *setsum* (*count N*) (*A* ∩ *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*) = (λ*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* = {#})
by (*auto simp add*: *size-def multiset-ext-iff*)

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)
by (*metis gr0I gr-implies-not0 size-empty size-eq-0-iff-empty*)

lemma *size-eq-Suc-imp-elem*: *size M = Suc n ==> ∃ a. a :# M*
apply (*unfold size-def*)
apply (*drule setsum-SucD*)
apply *auto*
done

lemma *size-eq-Suc-imp-eq-union*:
assumes *size M = Suc n*
shows *∃ a N. M = N + {#a#}*
proof −
from *assms* **obtain** *a* **where** *a ∈# M*
by (*erule size-eq-Suc-imp-elem* [*THEN exE*])
then have *M = M − {#a#} + {#a#}* **by** *simp*
then show *?thesis* **by** *blast*
qed

48.4 Induction and case splits

lemma *setsum-decr*:
finite F ==> (0::nat) < f a ==>
setsum (f (a := f a − 1)) F = (if a∈F then setsum f F − 1 else setsum f F)
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* $\{x.\ x \neq a \longrightarrow f\ x \neq 0\} = \{x.\ f\ x \neq 0\}$)
prefer *2*
apply *blast*
apply (*subgoal-tac* $\{x.\ x \neq a \wedge f\ x \neq 0\} = \{x.\ f\ x \neq 0\} - \{a\}$)
prefer *2*
apply *blast*
apply (*simp add*: *le-imp-diff-is-add setsum-diff1-nat cong*: *conj-cong*)
done

theorem *rep-multiset-induct*:
$f \in$ *multiset* $\Longrightarrow$ $P\ (\lambda a.\ 0) \Longrightarrow$
$(!!f\ b.\ f \in multiset \Longrightarrow P\ f \Longrightarrow P\ (f\ (b := f\ b + 1))) \Longrightarrow P\ f$
using *rep-multiset-induct-aux* **by** *blast*

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
assumes *empty*: $P\ \{\#\}$
and *add*: $!!M\ x.\ P\ M \Longrightarrow P\ (M + \{\#x\#\})$
shows $P\ M$
proof −
note *defns* = *union-def single-def Mempty-def*
note *add′* = *add* [*unfolded defns*, *simplified*]
have *aux*: $\bigwedge a::'a.\ count\ (Abs\text{-}multiset\ (\lambda b.\ if\ b = a\ then\ 1\ else\ 0)) =$
$(\lambda b.\ if\ b = a\ then\ 1\ else\ 0)$ **by** (*simp add*: *Abs-multiset-inverse in-multiset*)
show *?thesis*
apply (*rule count-inverse* [*THEN subst*])
apply (*rule count* [*THEN rep-multiset-induct*])
apply (*rule empty* [*unfolded defns*])
apply (*subgoal-tac* $f(b := f\ b + 1) = (\lambda a.\ f\ a + (if\ a{=}b\ then\ 1\ else\ 0))$)
prefer *2*
apply (*simp add*: *expand-fun-eq*)
apply (*erule ssubst*)
apply (*erule Abs-multiset-inverse* [*THEN subst*])
apply (*drule add′*)
apply (*simp add*: *aux*)
done
qed

lemma *multi-nonempty-split*: $M \neq \{\#\} \Longrightarrow \exists A\ a.\ M = A + \{\#a\#\}$
by (*induct M*) *auto*

lemma *multiset-cases* [*cases type*, *case-names empty add*]:
assumes *em*: $M = \{\#\} \Longrightarrow P$
assumes *add*: $\bigwedge N\ x.\ M = N + \{\#x\#\} \Longrightarrow P$
shows P
proof (*cases* $M = \{\#\}$)
assume $M = \{\#\}$ **then show** *?thesis* **using** *em* **by** *simp*

next
assume $M \neq \{\#\}$
then obtain M' m **where** $M = M' + \{\#m\#\}$
by (*blast dest*: *multi-nonempty-split*)
then show *?thesis* **using** *add* **by** *simp*
qed

lemma *multi-member-split*: $x \in\# M \Longrightarrow \exists A.\ M = A + \{\#x\#\}$
apply (*cases M*)
apply *simp*
apply (*rule-tac x*=$M - \{\#x\#\}$ **in** *exI*, *simp*)
done

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
by (*cases* $B = \{\#\}$) (*auto dest*: *multi-member-split*)

lemma *multiset-partition*: $M = \{\#\ x{:}\#M.\ P\ x\ \#\} + \{\#\ x{:}\#M.\ \neg\ P\ x\ \#\}$
apply (*subst multiset-ext-iff*)
apply *auto*
done

lemma *mset-less-size*: $(A{::}'a\ multiset) < B \Longrightarrow size\ A < size\ B$
proof (*induct A arbitrary*: *B*)
case (*empty M*)
then have $M \neq \{\#\}$ **by** (*simp add*: *mset-less-empty-nonempty*)
then obtain M' x **where** $M = M' + \{\#x\#\}$
by (*blast dest*: *multi-nonempty-split*)
then show *?case* **by** *simp*
next
case (*add S x T*)
have *IH*: $\bigwedge B.\ S < B \Longrightarrow size\ S < size\ B$ **by** *fact*
have *SxsubT*: $S + \{\#x\#\} < T$ **by** *fact*
then have $x \in\# T$ **and** $S < T$ **by** (*auto dest*: *mset-less-insertD*)
then obtain T' **where** T: $T = T' + \{\#x\#\}$
by (*blast dest*: *multi-member-split*)
then have $S < T'$ **using** *SxsubT*
by (*blast intro*: *mset-less-add-bothsides*)
then have $size\ S < size\ T'$ **using** *IH* **by** *simp*
then show *?case* **using** *T* **by** *simp*
qed

48.4.1 Strong induction and subset induction for multisets

Well-foundedness of proper subset operator:

proper multiset subset

definition
mset-less-rel :: $('a\ multiset * 'a\ multiset)\ set$ **where**
mset-less-rel = $\{(A,B).\ A < B\}$

lemma *multiset-add-sub-el-shuffle*:
assumes $c \in\# B$ **and** $b \neq c$
shows $B - \{\#c\#\} + \{\#b\#\} = B + \{\#b\#\} - \{\#c\#\}$
proof −
from ‹$c \in\# B$› **obtain** A **where** B: $B = A + \{\#c\#\}$
by (*blast dest*: *multi-member-split*)
have $A + \{\#b\#\} = A + \{\#b\#\} + \{\#c\#\} - \{\#c\#\}$ **by** *simp*
then have $A + \{\#b\#\} = A + \{\#c\#\} + \{\#b\#\} - \{\#c\#\}$
by (*simp add*: *add-ac*)
then show *?thesis* **using** B **by** *simp*
qed

lemma *wf-mset-less-rel*: *wf mset-less-rel*
apply (*unfold mset-less-rel-def*)
apply (*rule wf-measure* [*THEN wf-subset*, **where** *f1*=*size*])
apply (*clarsimp simp*: *measure-def inv-image-def mset-less-size*)
done

The induction rules:

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall (A::'a\ multiset).\ A < B \longrightarrow P\ A \Longrightarrow P\ B$
shows $P\ B$
apply (*rule wf-mset-less-rel* [*THEN wf-induct*])
apply (*rule ih*, *auto simp*: *mset-less-rel-def*)
done

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \leq A$
and *empty*: $P\ \{\#\}$
and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (F + \{\#a\#\})$
shows $P\ F$
proof −
from ‹$F \leq A$›
show *?thesis*
proof (*induct F*)
show $P\ \{\#\}$ **by** *fact*
next
fix $x\ F$
assume P: $F \leq A \Longrightarrow P\ F$ **and** i: $F + \{\#x\#\} \leq A$
show $P\ (F + \{\#x\#\})$
proof (*rule insert*)
from i **show** $x \in\# A$ **by** (*auto dest*: *mset-le-insertD*)
from i **have** $F \leq A$ **by** (*auto dest*: *mset-le-insertD*)
with P **show** $P\ F$ **.**
qed
qed
qed

48.5 Alternative representations

48.5.1 Lists

primrec *multiset-of* :: *'a list* ⇒ *'a multiset* **where**
multiset-of [] = {#} |
multiset-of (*a* # *x*) = *multiset-of x* + {# *a* #}

lemma *in-multiset-in-set*:
x ∈# *multiset-of xs* ⟷ *x* ∈ *set xs*
by (*induct xs*) *simp-all*

lemma *count-multiset-of*:
count (*multiset-of xs*) *x* = *length* (*filter* (λ*y*. *x* = *y*) *xs*)
by (*induct xs*) *simp-all*

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* ∈ *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*: *add-ac*)

lemma *surj-multiset-of*: *surj multiset-of*
apply (*unfold surj-def*)
apply (*rule allI*)
apply (*rule-tac M* = *y* **in** *multiset-induct*)
apply *auto*
apply (*rule-tac x* = *x* # *xa* **in** *exI*)
apply *auto*
done

lemma *set-count-greater-0*: *set x* = {*a*. *count* (*multiset-of x*) *a* > *0*}
by (*induct x*) *auto*

lemma *distinct-count-atmost-1*:
distinct x = (! *a*. *count* (*multiset-of x*) *a* = (*if a* ∈ *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 ⟹ set xs = set ys
by (*rule*) (*auto simp add*:*multiset-ext-iff set-count-greater-0*)

lemma *set-eq-iff-multiset-of-eq-distinct*:
distinct x ⟹ distinct y ⟹
(set x = set y) = (multiset-of x = multiset-of y)
by (*auto simp*: *multiset-ext-iff 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←xs. P x*] + *multiset-of* [*x←xs. ¬P x*] = *multiset-of xs*
by (*induct xs*) (*auto simp*: *add-ac*)

lemma *count-filter*:
count (*multiset-of xs*) *x* = *length* [*y* ← *xs*. *y* = *x*]
by (*induct xs*) *auto*

lemma *nth-mem-multiset-of*: *i* < *length ls* ⟹ (*ls* ! *i*) :# *multiset-of ls*
apply (*induct ls arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)
apply *auto*
done

lemma *multiset-of-remove1*[*simp*]:
multiset-of (*remove1 a xs*) = *multiset-of xs* − {#*a*#}
by (*induct xs*) (*auto simp add*: *multiset-ext-iff*)

lemma *multiset-of-eq-length*:
assumes *multiset-of xs = multiset-of ys*
shows *length xs = length ys*
using *assms* **proof** (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
then have *x* ∈# *multiset-of ys* **by** (*simp add*: *union-single-eq-member*)
then have *x* ∈ *set ys* **by** (*simp add*: *in-multiset-in-set*)

```
  from Cons.prems [symmetric] have multiset-of xs = multiset-of (remove1 x ys)
    by simp
  with Cons.hyps have length xs = length (remove1 x ys) .
  with ‹x ∈ set ys› show ?case
    by (auto simp add: length-remove1 dest: length-pos-if-in-set)
qed

lemma (in linorder) multiset-of-insort [simp]:
  multiset-of (insort x xs) = {#x#} + multiset-of xs
  by (induct xs) (simp-all add: ac-simps)

lemma (in linorder) multiset-of-sort [simp]:
  multiset-of (sort xs) = multiset-of xs
  by (induct xs) (simp-all add: ac-simps)
```

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

```
lemma (in linorder) properties-for-sort:
  multiset-of ys = multiset-of xs ⟹ sorted ys ⟹ sort xs = ys
proof (induct xs arbitrary: ys)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  then have x ∈ set ys
    by (auto simp add:  mem-set-multiset-eq intro!: ccontr)
  with Cons.prems Cons.hyps [of remove1 x ys] show ?case
    by (simp add: sorted-remove1 multiset-of-remove1 insort-remove1)
qed

lemma multiset-of-remdups-le: multiset-of (remdups xs) ≤ multiset-of xs
  by (induct xs) (auto intro: order-trans)

lemma multiset-of-update:
  i < length ls ⟹ multiset-of (ls[i := v]) = multiset-of ls - {#ls ! i#} + {#v#}
proof (induct ls arbitrary: i)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases i)
    case 0 then show ?thesis by simp
  next
    case (Suc i')
    with Cons show ?thesis
      apply simp
      apply (subst add-assoc)
      apply (subst add-commute [of {#v#} {#x#}])
      apply (subst add-assoc [symmetric])
      apply simp
```

```
    apply (rule mset-le-multiset-union-diff-commute)
    apply (simp add: mset-le-single nth-mem-multiset-of)
    done
  qed
qed

lemma multiset-of-swap:
  i < length ls ⟹ j < length ls ⟹
    multiset-of (ls[j := ls ! i, i := ls ! j]) = multiset-of ls
  by (cases i = j) (simp-all add: multiset-of-update nth-mem-multiset-of)
```

48.5.2 Association lists – including rudimentary code generation

```
definition count-of :: ('a × nat) list ⇒ 'a ⇒ nat where
  count-of xs x = (case map-of xs x of None ⇒ 0 | Some n ⇒ n)

lemma count-of-multiset:
  count-of xs ∈ multiset
proof −
  let ?A = {x::'a. 0 < (case map-of xs x of None ⇒ 0::nat | Some (n::nat) ⇒ n)}
  have ?A ⊆ dom (map-of xs)
  proof
    fix x
    assume x ∈ ?A
    then have 0 < (case map-of xs x of None ⇒ 0::nat | Some (n::nat) ⇒ n) by
simp
    then have map-of xs x ≠ None by (cases map-of xs x) auto
    then show x ∈ dom (map-of xs) by auto
  qed
  with finite-dom-map-of [of xs] have finite ?A
    by (auto intro: finite-subset)
  then show ?thesis
    by (simp add: count-of-def expand-fun-eq multiset-def)
qed

lemma count-simps [simp]:
  count-of [] = (λ-. 0)
  count-of ((x, n) # xs) = (λy. if x = y then n else count-of xs y)
  by (simp-all add: count-of-def expand-fun-eq)

lemma count-of-empty:
  x ∉ fst ' set xs ⟹ count-of xs x = 0
  by (induct xs) (simp-all add: count-of-def)

lemma count-of-filter:
  count-of (filter (P ∘ fst) xs) x = (if P x then count-of xs x else 0)
  by (induct xs) auto

definition Bag :: ('a × nat) list ⇒ 'a multiset where
```

Bag xs = Abs-multiset (count-of xs)

code-datatype *Bag*

lemma *count-Bag* [*simp*, *code*]:
count (Bag xs) = count-of xs
by (*simp add*: *Bag-def count-of-multiset Abs-multiset-inverse*)

lemma *Mempty-Bag* [*code*]:
{#} = Bag []
by (*simp add*: *multiset-ext-iff*)

lemma *single-Bag* [*code*]:
{#x#} = Bag [(x, 1)]
by (*simp add*: *multiset-ext-iff*)

lemma *MCollect-Bag* [*code*]:
MCollect (Bag xs) P = Bag (filter (P ∘ fst) xs)
by (*simp add*: *multiset-ext-iff count-of-filter*)

lemma *mset-less-eq-Bag* [*code*]:
Bag xs ≤ A ⟷ (∀(x, n) ∈ set xs. count-of xs x ≤ count A x)
(**is** *?lhs ⟷ ?rhs*)
proof
assume *?lhs* **then show** *?rhs*
by (*auto simp add*: *mset-le-def count-Bag*)
next
assume *?rhs*
show *?lhs*
proof (*rule mset-less-eqI*)
fix *x*
from ‹*?rhs*› **have** *count-of xs x ≤ count A x*
by (*cases x ∈ fst ' set xs*) (*auto simp add*: *count-of-empty*)
then show *count (Bag xs) x ≤ count A x*
by (*simp add*: *mset-le-def count-Bag*)
qed
qed

instantiation *multiset* :: (*eq*) *eq*
begin

definition
HOL.eq A B ⟷ (A::'a multiset) ≤ B ∧ B ≤ A

instance proof
qed (*simp add*: *eq-multiset-def eq-iff*)

end

definition (**in** *term-syntax*)
bagify :: ('*a*::*typerep* × *nat*) *list* × (*unit* ⇒ *Code-Evaluation.term*)
⇒ '*a multiset* × (*unit* ⇒ *Code-Evaluation.term*) **where**
[*code-unfold*]: *bagify xs* = *Code-Evaluation.valtermify Bag* {·} *xs*

notation *fcomp* (**infixl** *o>* *60*)
notation *scomp* (**infixl** *o→* *60*)

instantiation *multiset* :: (*random*) *random*
begin

definition
Quickcheck.random i = *Quickcheck.random i o→* (λ*xs*. *Pair* (*bagify xs*))

instance ..

end

no-notation *fcomp* (**infixl** *o>* *60*)
no-notation *scomp* (**infixl** *o→* *60*)

hide-const (**open**) *bagify*

48.6 The multiset order

48.6.1 Well-foundedness

definition *mult1* :: ('*a* × '*a*) *set* => ('*a multiset* × '*a multiset*) *set* **where**
[*code del*]: *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**
[*code del*]: *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* = λ*K a*. ∀ *b*. *b* :# *K* −−> (*b*, *a*) ∈ *r*
let *?R* = λ*N 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: add-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: add-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*)

48.6.2 Closure-free presentation

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*: *add-assoc* [*symmetric*])
 apply (*drule-tac f* = λ*M*. *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-ext-iff split*: *nat-diff-split*)
 apply (*rule conjI*)
  apply (*drule-tac f* = λ*M*. *M* − {#*a*#} **in** *arg-cong*, *simp*)
  apply (*simp add*: *multiset-ext-iff 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 *one-step-implies-mult-aux*:
  trans r ==>
    ∀ *I J K*. (*size J* = *n* ∧ *J* ≠ {#} ∧ (∀ *k* ∈ *set-of K*. ∃ *j* ∈ *set-of J*. (*k*, *j*) ∈ *r*))
      −−> (*I* + *K*, *I* + *J*) ∈ *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* $P = \forall\, k \in$ *set-of* $K.$ *?P* k **in** *rev-mp*)
apply (*erule ssubst*)
apply (*simp add*: *Ball-def*, *auto*)
apply (*subgoal-tac*
 $((I + \{\#\ x :\#\ K.\ (x,\ a) \in r\ \#\}) + \{\#\ x :\#\ K.\ (x,\ a) \notin r\ \#\},$
 $(I + \{\#\ x :\#\ K.\ (x,\ a) \in r\ \#\}) + J') \in$ *mult* r)
 prefer *2*
 apply *force*
apply (*simp* (*no-asm-use*) *add*: *add-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*: *add-ac*)
done

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

48.6.3 Partial-order properties

definition *less-multiset* :: *'a*::*order multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** <# *50*)
where
 $M' <\# M \longleftrightarrow (M',\ M) \in$ *mult* $\{(x',\ x).\ x' < x\}$

definition *le-multiset* :: *'a*::*order multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** <=# *50*)
where
 $M' <=\# M \longleftrightarrow M' <\# M \vee M' = M$

notation (*xsymbols*) *less-multiset* (**infix** ⊂# *50*)
notation (*xsymbols*) *le-multiset* (**infix** ⊆# *50*)

interpretation *multiset-order*: *order le-multiset less-multiset*
proof −
 have *irrefl*: $\bigwedge M$ :: *'a multiset*. ¬ $M \subset\# M$
 proof
  fix M :: *'a multiset*
  assume $M \subset\# M$
 then have *MM*: $(M,\ M) \in$ *mult* $\{(x,\ y).\ x < y\}$ **by** (*simp add*: *less-multiset-def*)
  have *trans* $\{(x'::'a,\ x).\ x' < x\}$
   by (*rule transI*) *simp*
  moreover note *MM*
  ultimately have $\exists\, I\ J\ K.\ M = I + J \wedge M = I + K$
   $\wedge\ J \neq \{\#\} \wedge (\forall\, k{\in}$*set-of* $K.\ \exists\, j{\in}$*set-of* $J.\ (k,\ j) \in \{(x,\ y).\ x < y\})$

by (*rule mult-implies-one-step*)
then obtain $I\ J\ K$ **where** $M = I + J$ **and** $M = I + K$
and $J \neq \{\#\}$ **and** $(\forall k{\in}\textit{set-of}\ K.\ \exists j{\in}\textit{set-of}\ J.\ (k, j) \in \{(x, y).\ x < y\})$ **by** *blast*
then have *aux1*: $K \neq \{\#\}$ **and** *aux2*: $\forall k{\in}\textit{set-of}\ K.\ \exists j{\in}\textit{set-of}\ K.\ k < j$ **by** *auto*
have *finite* (*set-of* K) **by** *simp*
moreover note *aux2*
ultimately have *set-of* $K = \{\}$
by (*induct rule*: *finite-induct*) (*auto intro*: *order-less-trans*)
with *aux1* **show** *False* **by** *simp*
qed
have *trans*: $\bigwedge K\ M\ N :: 'a\ multiset.\ K \subset\# M \Longrightarrow M \subset\# N \Longrightarrow K \subset\# N$
unfolding *less-multiset-def mult-def* **by** (*blast intro*: *trancl-trans*)
show *class.order* (*le-multiset* :: *'a multiset* ⇒ -) *less-multiset* **proof**
qed (*auto simp add*: *le-multiset-def irrefl dest*: *trans*)
qed

lemma *mult-less-irrefl* [*elim*!]:
$M \subset\# (M::'a::order\ multiset)$ ==> *R*
by (*simp add*: *multiset-order.less-irrefl*)

48.6.4 Monotonicity of multiset union

lemma *mult1-union*:
$(B, D) \in \textit{mult1}\ r$ ==> *trans r* ==> $(C + B, C + D) \in \textit{mult1}\ r$
apply (*unfold mult1-def*)
apply *auto*
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = C + M0$ **in** *exI*)
apply (*simp add*: *add-assoc*)
done

lemma *union-less-mono2*: $B \subset\# D$ ==> $C + B \subset\# C + (D::'a::order\ multiset)$
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*: $B \subset\# D$ ==> $B + C \subset\# D + (C::'a::order\ multiset)$
apply (*subst add-commute* [*of B C*])
apply (*subst add-commute* [*of D C*])
apply (*erule union-less-mono2*)
done

lemma *union-less-mono*:
$A \subset\# C$ ==> $B \subset\# D$ ==> $A + B \subset\# C + (D::'a::order\ multiset)$
by (*blast intro*!: *union-less-mono1 union-less-mono2 multiset-order.less-trans*)

interpretation *multiset-order*: *ordered-ab-semigroup-add plus le-multiset less-multiset*
proof
qed (*auto simp add*: *le-multiset-def intro*: *union-less-mono2*)

48.7 The fold combinator

The intended behaviour is *fold-mset f z* $\{\#x_1, ..., x_n\#\} = f\ x_1\ (\ldots\ (f\ x_n\ z)\ldots)$ if f is associative-commutative.

The graph of *fold-mset*, z: the start element, f: folding function, A: the multiset, y: the result.

inductive
fold-msetG :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b \Rightarrow bool$
for $f :: 'a \Rightarrow 'b \Rightarrow 'b$
and $z :: 'b$
where
emptyI [*intro*]: *fold-msetG f z* $\{\#\}$ *z*
| *insertI* [*intro*]: *fold-msetG f z A y* $\Longrightarrow$ *fold-msetG f z* $(A + \{\#x\#\})$ $(f\ x\ y)$

inductive-cases *empty-fold-msetGE* [*elim*!]: *fold-msetG f z* $\{\#\}$ *x*
inductive-cases *insert-fold-msetGE*: *fold-msetG f z* $(A + \{\#\})$ *y*

definition
fold-mset :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b$ **where**
fold-mset f z A = $(THE\ x.$ *fold-msetG f z A x*$)$

lemma *Diff1-fold-msetG*:
fold-msetG f z $(A - \{\#x\#\})$ *y* $\Longrightarrow$ *x* $\in\#$ *A* $\Longrightarrow$ *fold-msetG f z A* $(f\ x\ y)$
apply (*frule-tac x* = *x* **in** *fold-msetG.insertI*)
apply *auto*
done

lemma *fold-msetG-nonempty*: $\exists x.$ *fold-msetG f z A x*
apply (*induct A*)
apply *blast*
apply *clarsimp*
apply (*drule-tac x* = *x* **in** *fold-msetG.insertI*)
apply *auto*
done

lemma *fold-mset-empty*[*simp*]: *fold-mset f z* $\{\#\}$ = *z*
unfolding *fold-mset-def* **by** *blast*

context *fun-left-comm*
begin

lemma *fold-msetG-determ*:
fold-msetG f z A x $\Longrightarrow$ *fold-msetG f z A y* $\Longrightarrow$ *y* = *x*

proof (*induct arbitrary*: *x y z rule*: *full-multiset-induct*)
case (*less M* x_1 x_2 *Z*)
have *IH*: $\forall A.\ A < M \longrightarrow$
$(\forall x\ x'\ x''.$ *fold-msetG f x'' A x* $\longrightarrow$ *fold-msetG f x'' A x'*
$\longrightarrow x' = x)$ **by** *fact*
have $Mfoldx_1$: *fold-msetG f Z M* x_1 **and** $Mfoldx_2$: *fold-msetG f Z M* x_2 **by** *fact+*
show *?case*
proof (*rule fold-msetG.cases* [*OF* $Mfoldx_1$])
assume $M = \{\#\}$ **and** $x_1 = Z$
then show *?case* **using** $Mfoldx_2$ **by** *auto*
next
fix *B b u*
assume $M = B + \{\#b\#\}$ **and** $x_1 = f\ b\ u$ **and** *Bu*: *fold-msetG f Z B u*
then have *MBb*: $M = B + \{\#b\#\}$ **and** x_1: $x_1 = f\ b\ u$ **by** *auto*
show *?case*
proof (*rule fold-msetG.cases* [*OF* $Mfoldx_2$])
assume $M = \{\#\}$ $x_2 = Z$
then show *?case* **using** $Mfoldx_1$ **by** *auto*
next
fix *C c v*
assume $M = C + \{\#c\#\}$ **and** $x_2 = f\ c\ v$ **and** *Cv*: *fold-msetG f Z C v*
then have *MCc*: $M = C + \{\#c\#\}$ **and** x_2: $x_2 = f\ c\ v$ **by** *auto*
then have *CsubM*: $C < M$ **by** *simp*
from *MBb* **have** *BsubM*: $B < M$ **by** *simp*
show *?case*
proof *cases*
assume *b=c*
then moreover have $B = C$ **using** *MBb MCc* **by** *auto*
ultimately show *?thesis* **using** *Bu Cv* x_1 x_2 *CsubM IH* **by** *auto*
next
assume *diff*: $b \neq c$
let *?D* $= B - \{\#c\#\}$
have *cinB*: $c \in\# B$ **and** *binC*: $b \in\# C$ **using** *MBb MCc diff*
by (*auto intro*: *insert-noteq-member dest*: *sym*)
have $B - \{\#c\#\} < B$ **using** *cinB* **by** (*rule mset-less-diff-self*)
then have *DsubM*: *?D* $< M$ **using** *BsubM* **by** (*blast intro*: *order-less-trans*)
from *MBb MCc* **have** $B + \{\#b\#\} = C + \{\#c\#\}$ **by** *blast*
then have [*simp*]: $B + \{\#b\#\} - \{\#c\#\} = C$
using *MBb MCc binC cinB* **by** *auto*
have *B*: $B =$ *?D* $+ \{\#c\#\}$ **and** *C*: $C =$ *?D* $+ \{\#b\#\}$
using *MBb MCc diff binC cinB*
by (*auto simp*: *multiset-add-sub-el-shuffle*)
then obtain *d* **where** *Dfoldd*: *fold-msetG f Z ?D d*
using *fold-msetG-nonempty* **by** *iprover*
then have *fold-msetG f Z B* (*f c d*) **using** *cinB*
by (*rule Diff1-fold-msetG*)
then have $f\ c\ d = u$ **using** *IH BsubM Bu* **by** *blast*
moreover
have *fold-msetG f Z C* (*f b d*) **using** *binC cinB diff Dfoldd*

```
        by (auto simp: multiset-add-sub-el-shuffle
          dest: fold-msetG.insertI [where x=b])
      then have f b d = v using IH CsubM Cv by blast
      ultimately show ?thesis using x1 x2
        by (auto simp: fun-left-comm)
    qed
  qed
 qed
qed

lemma fold-mset-insert-aux:
  (fold-msetG f z (A + {#x#}) v) =
    (∃ y. fold-msetG f z A y ∧ v = f x y)
apply (rule iffI)
 prefer 2
 apply blast
apply (rule-tac A=A and f=f in fold-msetG-nonempty [THEN exE, standard])
apply (blast intro: fold-msetG-determ)
done

lemma fold-mset-equality: fold-msetG f z A y ⟹ fold-mset f z A = y
unfolding fold-mset-def by (blast intro: fold-msetG-determ)

lemma fold-mset-insert:
  fold-mset f z (A + {#x#}) = f x (fold-mset f z A)
apply (simp add: fold-mset-def fold-mset-insert-aux add-commute)
apply (rule the-equality)
 apply (auto cong add: conj-cong
    simp add: fold-mset-def [symmetric] fold-mset-equality fold-msetG-nonempty)
done

lemma fold-mset-insert-idem:
  fold-mset f z (A + {#a#}) = f a (fold-mset f z A)
apply (simp add: fold-mset-def fold-mset-insert-aux)
apply (rule the-equality)
 apply (auto cong add: conj-cong
    simp add: fold-mset-def [symmetric] fold-mset-equality fold-msetG-nonempty)
done

lemma fold-mset-commute: f x (fold-mset f z A) = fold-mset f (f x z) A
by (induct A) (auto simp: fold-mset-insert fun-left-comm [of x])

lemma fold-mset-single [simp]: fold-mset f z {#x#} = f x z
using fold-mset-insert [of z {#}] by simp

lemma fold-mset-union [simp]:
  fold-mset f z (A+B) = fold-mset f (fold-mset f z A) B
proof (induct A)
  case empty then show ?case by simp
```

next
case (*add A x*)
have $A + \{\#x\#\} + B = (A+B) + \{\#x\#\}$ **by** (*simp add*: *add-ac*)
then have *fold-mset f z* $(A + \{\#x\#\} + B) =$ *f x* (*fold-mset f z* $(A + B)$)
by (*simp add*: *fold-mset-insert*)
also have $\ldots =$ *fold-mset f* (*fold-mset f z* $(A + \{\#x\#\}))$ *B*
by (*simp add*: *fold-mset-commute*[*of x*,*symmetric*] *add fold-mset-insert*)
finally show *?case* **.**
qed

lemma *fold-mset-fusion*:
assumes *fun-left-comm g*
shows $(\bigwedge x\ y.\ h\ (g\ x\ y) = f\ x\ (h\ y)) \Longrightarrow h$ (*fold-mset g w A*) = *fold-mset f* (*h w*) *A* (**is** *PROP ?P*)
proof −
interpret *fun-left-comm g* **by** (*fact assms*)
show *PROP ?P* **by** (*induct A*) *auto*
qed

lemma *fold-mset-rec*:
assumes $a \in\# A$
shows *fold-mset f z A* = *f a* (*fold-mset f z* $(A - \{\#a\#\})$)
proof −
from *assms* **obtain** A' **where** $A = A' + \{\#a\#\}$
by (*blast dest*: *multi-member-split*)
then show *?thesis* **by** *simp*
qed

end

A note on code generation: When defining some function containing a subterm *fold-mset F*, code generation is not automatic. When interpreting locale *left-commutative* with *F*, the would be code thms for *fold-mset* become thms like *fold-mset F z* $\{\#\} = z$ where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

48.8 Image

definition *image-mset* :: $('a \Rightarrow 'b) \Rightarrow$ *'a multiset* $\Rightarrow$ *'b multiset* **where**
image-mset f = *fold-mset* (*op* + *o single o f*) $\{\#\}$

interpretation *image-left-comm*: *fun-left-comm op* + *o single o f*
proof qed (*simp add*: *add-ac*)

lemma *image-mset-empty* [*simp*]: *image-mset f* $\{\#\} = \{\#\}$
by (*simp add*: *image-mset-def*)

lemma *image-mset-single* [*simp*]: *image-mset f* $\{\#x\#\} = \{\#f\ x\#\}$

by (*simp add*: *image-mset-def*)

lemma *image-mset-insert*:
image-mset f (*M* + {#*a*#}) = *image-mset f M* + {#*f a*#}
by (*simp add*: *image-mset-def add-ac*)

lemma *image-mset-union* [*simp*]:
image-mset f (*M*+*N*) = *image-mset f M* + *image-mset f N*
apply (*induct N*)
apply *simp*
apply (*simp add*: *add-assoc* [*symmetric*] *image-mset-insert*)
done

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*
by (*induct M*) *simp-all*

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} ⟷ *M* = {#}
by (*cases M*) *auto*

syntax
-comprehension1-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset*
(({#-/. - :# -#}))
translations
{#*e*. *x*:#*M*#} == *CONST image-mset* (%*x*. *e*) *M*

syntax
-comprehension2-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset*
(({#-/ | - :# -./ -#}))
translations
{#*e* | *x*:#*M*. *P*#} => {#*e*. *x* :# {# *x*:#*M*. *P*#}#}

This allows to write not just filters like {# *x* :# *M*. *x* < *c*#} but also images like {#*x* + *x*. *x* :# *M*#} and {#*x*+*x*|*x*:#*M*. *x*<*c*#}, where the latter is currently displayed as {#*x* + *x*. *x* :# {# *x* :# *M*. *x* < *c*#}#}.

48.9 Termination proofs with multiset orders

lemma *multi-member-skip*: *x* ∈# *XS* ⟹ *x* ∈# {# *y* #} + *XS*
and *multi-member-this*: *x* ∈# {# *x* #} + *XS*
and *multi-member-last*: *x* ∈# {# *x* #}
by *auto*

definition *ms-strict* = *mult pair-less*
definition [*code del*]: *ms-weak* = *ms-strict* ∪ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
unfolding *reduction-pair-def ms-strict-def ms-weak-def pair-less-def*
by (*auto intro*: *wf-mult1 wf-trancl simp*: *mult-def*)

lemma *smsI*:
(*set-of A*, *set-of B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-strict*
unfolding *ms-strict-def*
by (*rule one-step-implies-mult*) (*auto simp add*: *max-strict-def pair-less-def elim*!:*max-ext.cases*)

lemma *wmsI*:
(*set-of A*, *set-of B*) ∈ *max-strict* ∨ *A* = {#} ∧ *B* = {#}
⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-weak*
unfolding *ms-weak-def ms-strict-def*
by (*auto simp add*: *pair-less-def max-strict-def elim*!:*max-ext.cases intro*: *one-step-implies-mult*)

inductive *pw-leq*
where
pw-leq-empty: *pw-leq* {#} {#}
| *pw-leq-step*: ⟦(*x*,*y*) ∈ *pair-leq*; *pw-leq X Y* ⟧ ⟹ *pw-leq* ({#*x*#} + *X*) ({#*y*#}
+ *Y*)

lemma *pw-leq-lstep*:
(*x*, *y*) ∈ *pair-leq* ⟹ *pw-leq* {#*x*#} {#*y*#}
by (*drule pw-leq-step*) (*rule pw-leq-empty*, *simp*)

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows ∃ *A B Z*. *X* = *A* + *Z* ∧ *Y* = *B* + *Z* ∧ ((*set-of A*, *set-of B*) ∈ *max-strict*
∨ (*B* = {#} ∧ *A* = {#}))
using *assms*
proof (*induct*)
case *pw-leq-empty* **thus** *?case* **by** *auto*
next
case (*pw-leq-step x y X Y*)
then obtain *A B Z* **where**
[*simp*]: *X* = *A* + *Z Y* = *B* + *Z*
and *1*[*simp*]: (*set-of A*, *set-of B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#})
by *auto*
from *pw-leq-step* **have** *x* = *y* ∨ (*x*, *y*) ∈ *pair-less*
unfolding *pair-leq-def* **by** *auto*
thus *?case*
proof
assume [*simp*]: *x* = *y*
have
{#*x*#} + *X* = *A* + ({#*y*#}+*Z*)
∧ {#*y*#} + *Y* = *B* + ({#*y*#}+*Z*)
∧ ((*set-of A*, *set-of B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#}))
by (*auto simp*: *add-ac*)
thus *?case* **by** (*intro exI*)
next
assume *A*: (*x*, *y*) ∈ *pair-less*
let *?A′* = {#*x*#} + *A* **and** *?B′* = {#*y*#} + *B*
have {#*x*#} + *X* = *?A′* + *Z*

```
    {#y#} + Y = ?B' + Z
    by (auto simp add: add-ac)
  moreover have
    (set-of ?A', set-of ?B') ∈ max-strict
    using 1 A unfolding max-strict-def
    by (auto elim!: max-ext.cases)
  ultimately show ?thesis by blast
 qed
qed

lemma
 assumes pwleq: pw-leq Z Z'
  shows ms-strictI: (set-of A, set-of B) ∈ max-strict ⟹ (Z + A, Z' + B) ∈
ms-strict
  and   ms-weakI1: (set-of A, set-of B) ∈ max-strict ⟹ (Z + A, Z' + B) ∈
ms-weak
 and   ms-weakI2: (Z + {#}, Z' + {#}) ∈ ms-weak
proof −
 from pw-leq-split[OF pwleq]
 obtain A' B' Z''
  where [simp]: Z = A' + Z'' Z' = B' + Z''
  and mx-or-empty: (set-of A', set-of B') ∈ max-strict ∨ (A' = {#} ∧ B' = {#})
  by blast
 {
  assume max: (set-of A, set-of B) ∈ max-strict
  from mx-or-empty
  have (Z'' + (A + A'), Z'' + (B + B')) ∈ ms-strict
  proof
    assume max': (set-of A', set-of B') ∈ max-strict
    with max have (set-of (A + A'), set-of (B + B')) ∈ max-strict
      by (auto simp: max-strict-def intro: max-ext-additive)
    thus ?thesis by (rule smsI)
  next
    assume [simp]: A' = {#} ∧ B' = {#}
    show ?thesis by (rule smsI) (auto intro: max)
  qed
  thus (Z + A, Z' + B) ∈ ms-strict by (simp add:add-ac)
  thus (Z + A, Z' + B) ∈ ms-weak by (simp add: ms-weak-def)
 }
 from mx-or-empty
 have (Z'' + A', Z'' + B') ∈ ms-weak by (rule wmsI)
 thus (Z + {#}, Z' + {#}) ∈ ms-weak by (simp add:add-ac)
qed

lemma empty-idemp: {#} + x = x x + {#} = x
and nonempty-plus: {# x #} + rs ≠ {#}
and nonempty-single: {# x #} ≠ {#}
by auto
```

```
setup ⟪
let
  fun msetT T = Type (@{type-name multiset}, [T]);

  fun mk-mset T [] = Const (@{const-abbrev Mempty}, msetT T)
    | mk-mset T [x] = Const (@{const-name single}, T --> msetT T) $ x
    | mk-mset T (x :: xs) =
        Const (@{const-name plus}, msetT T --> msetT T --> msetT T) $
          mk-mset T [x] $ mk-mset T xs

  fun mset-member-tac m i =
      (if m <= 0 then
        rtac @{thm multi-member-this} i ORELSE rtac @{thm multi-member-last}
i
       else
         rtac @{thm multi-member-skip} i THEN mset-member-tac (m - 1) i)

  val mset-nonempty-tac =
      rtac @{thm nonempty-plus} ORELSE' rtac @{thm nonempty-single}

  val regroup-munion-conv =
      Function-Lib.regroup-conv @{const-abbrev Mempty} @{const-name plus}
      (map (fn t => t RS eq-reflection) (@{thms add-ac} @ @{thms empty-idemp}))

  fun unfold-pwleq-tac i =
    (rtac @{thm pw-leq-step} i THEN (fn st => unfold-pwleq-tac (i + 1) st))
      ORELSE (rtac @{thm pw-leq-lstep} i)
      ORELSE (rtac @{thm pw-leq-empty} i)

 val set-of-simps = [@{thm set-of-empty}, @{thm set-of-single}, @{thm set-of-union},
                     @{thm Un-insert-left}, @{thm Un-empty-left}]
in
  ScnpReconstruct.multiset-setup (ScnpReconstruct.Multiset
  {
    msetT=msetT, mk-mset=mk-mset, mset-regroup-conv=regroup-munion-conv,
    mset-member-tac=mset-member-tac, mset-nonempty-tac=mset-nonempty-tac,
    mset-pwleq-tac=unfold-pwleq-tac, set-of-simps=set-of-simps,
     smsI'= @{thm ms-strictI}, wmsI2''= @{thm ms-weakI2}, wmsI1= @{thm
ms-weakI1},
    reduction-pair= @{thm ms-reduction-pair}
  })
end
⟫
```

48.10 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-ext-iff* [*symmetric*]

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$

by (*fact add-commute*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
by (*fact add-assoc*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
by (*fact add-left-commute*)

lemmas *union-ac* = *union-assoc union-commute union-lcomm*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ multiset)$
by (*fact add-right-cancel*)

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ multiset)$
by (*fact add-left-cancel*)

lemma *multi-union-self-other-eq*: $(A::'a\ multiset) + X = A + Y \Longrightarrow X = Y$
by (*fact add-imp-eq*)

lemma *mset-less-trans*: $(M::'a\ multiset) < K \Longrightarrow K < N \Longrightarrow M < N$
by (*fact order-less-trans*)

lemma *multiset-inter-commute*: $A \#\cap B = B \#\cap A$
by (*fact inf.commute*)

lemma *multiset-inter-assoc*: $A \#\cap (B \#\cap C) = A \#\cap B \#\cap C$
by (*fact inf.assoc* [*symmetric*])

lemma *multiset-inter-left-commute*: $A \#\cap (B \#\cap C) = B \#\cap (A \#\cap C)$
by (*fact inf.left-commute*)

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *mult-less-not-refl*:
$\neg\ M \subset\# (M::'a::order\ multiset)$
by (*fact multiset-order.less-irrefl*)

lemma *mult-less-trans*:
$K \subset\# M \Longrightarrow M \subset\# N \Longrightarrow K \subset\# (N::'a::order\ multiset)$
by (*fact multiset-order.less-trans*)

lemma *mult-less-not-sym*:
$M \subset\# N \Longrightarrow \neg\ N \subset\# (M::'a::order\ multiset)$
by (*fact multiset-order.less-not-sym*)

lemma *mult-less-asym*:

```
  M ⊂# N ==> (¬ P ==> N ⊂# (M::'a::order multiset)) ==> P
  by (fact multiset-order.less-asym)

ML ⟪
fun multiset-postproc - maybe-name all-values (T as Type (-, [elem-T]))
                      (Const - $ t') =
    let
      val (maybe-opt, ps) =
        Nitpick-Model.dest-plain-fun t' ||> op ~~
        ||> map (apsnd (snd o HOLogic.dest-number))
      fun elems-for t =
        case AList.lookup (op =) ps t of
          SOME n => replicate n t
        | NONE => [Const (maybe-name, elem-T --> elem-T) $ t]
    in
      case maps elems-for (all-values elem-T) @
          (if maybe-opt then [Const (Nitpick-Model.unrep (), elem-T)]
           else []) of
        [] => Const (@{const-name zero-class.zero}, T)
      | ts => foldl1 (fn (t1, t2) =>
                         Const (@{const-name plus-class.plus}, T --> T --> T)
                         $ t1 $ t2)
                     (map (curry (op $) (Const (@{const-name single},
                                                elem-T --> T))) ts)
    end
  | multiset-postproc - - - - t = t
⟫

setup ⟪
Nitpick.register-term-postprocessor @{typ 'a multiset} multiset-postproc
⟫

end
```

49 Nat-Infinity: Natural numbers with infinity

theory *Nat-Infinity*
imports *Main*
begin

49.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

datatype *inat* = *Fin nat* | *Infty*

notation (*xsymbols*)

Infty (∞)

notation (*HTML* **output**)
Infty (∞)

lemma *not-Infty-eq*[*iff*]: (x ~= *Infty*) = (*EX* i. x = *Fin* i)
by (*cases* x) *auto*

lemma *not-Fin-eq* [*iff*]: (*ALL* y. x ~= *Fin* y) = (x = *Infty*)
by (*cases* x) *auto*

49.2 Constructors and numbers

instantiation *inat* :: {*zero*, *one*, *number*}
begin

definition
0 = *Fin 0*

definition
[*code-unfold*]: *1* = *Fin 1*

definition
[*code-unfold*, *code del*]: *number-of k* = *Fin* (*number-of k*)

instance ..

end

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

lemma *Fin-0*: *Fin 0* = *0*
by (*simp add*: *zero-inat-def*)

lemma *Fin-1*: *Fin 1* = *1*
by (*simp add*: *one-inat-def*)

lemma *Fin-number*: *Fin* (*number-of k*) = *number-of k*
by (*simp add*: *number-of-inat-def*)

lemma *one-iSuc*: *1* = *iSuc 0*
by (*simp add*: *zero-inat-def one-inat-def iSuc-def*)

lemma *Infty-ne-i0* [*simp*]: ∞ ≠ *0*
by (*simp add*: *zero-inat-def*)

lemma *i0-ne-Infty* [*simp*]: *0* ≠ ∞

by (*simp add*: *zero-inat-def*)

lemma *zero-inat-eq* [*simp*]:
number-of k = (*0*::*inat*) ⟷ *number-of k* = (*0*::*nat*)
(*0*::*inat*) = *number-of k* ⟷ *number-of k* = (*0*::*nat*)
unfolding *zero-inat-def number-of-inat-def* **by** *simp-all*

lemma *one-inat-eq* [*simp*]:
number-of k = (*1*::*inat*) ⟷ *number-of k* = (*1*::*nat*)
(*1*::*inat*) = *number-of k* ⟷ *number-of k* = (*1*::*nat*)
unfolding *one-inat-def number-of-inat-def* **by** *simp-all*

lemma *zero-one-inat-neq* [*simp*]:
¬ *0* = (*1*::*inat*)
¬ *1* = (*0*::*inat*)
unfolding *zero-inat-def one-inat-def* **by** *simp-all*

lemma *Infty-ne-i1* [*simp*]: ∞ ≠ *1*
by (*simp add*: *one-inat-def*)

lemma *i1-ne-Infty* [*simp*]: *1* ≠ ∞
by (*simp add*: *one-inat-def*)

lemma *Infty-ne-number* [*simp*]: ∞ ≠ *number-of k*
by (*simp add*: *number-of-inat-def*)

lemma *number-ne-Infty* [*simp*]: *number-of k* ≠ ∞
by (*simp add*: *number-of-inat-def*)

lemma *iSuc-Fin*: *iSuc* (*Fin n*) = *Fin* (*Suc n*)
by (*simp add*: *iSuc-def*)

lemma *iSuc-number-of*: *iSuc* (*number-of k*) = *Fin* (*Suc* (*number-of k*))
by (*simp add*: *iSuc-Fin number-of-inat-def*)

lemma *iSuc-Infty* [*simp*]: *iSuc* ∞ = ∞
by (*simp add*: *iSuc-def*)

lemma *iSuc-ne-0* [*simp*]: *iSuc n* ≠ *0*
by (*simp add*: *iSuc-def zero-inat-def split*: *inat.splits*)

lemma *zero-ne-iSuc* [*simp*]: *0* ≠ *iSuc n*
by (*rule iSuc-ne-0* [*symmetric*])

lemma *iSuc-inject* [*simp*]: *iSuc m* = *iSuc n* ⟷ *m* = *n*
by (*simp add*: *iSuc-def split*: *inat.splits*)

lemma *number-of-inat-inject* [*simp*]:
(*number-of k* :: *inat*) = *number-of l* ⟷ (*number-of k* :: *nat*) = *number-of l*

by (*simp add*: *number-of-inat-def*)

49.3 Addition

instantiation *inat* :: *comm-monoid-add*
begin

definition
[*code del*]: $m + n = (case\ m\ of\ \infty \Rightarrow \infty \mid Fin\ m \Rightarrow (case\ n\ of\ \infty \Rightarrow \infty \mid Fin\ n \Rightarrow Fin\ (m + n)))$

lemma *plus-inat-simps* [*simp*, *code*]:
$Fin\ m + Fin\ n = Fin\ (m + n)$
$\infty + q = \infty$
$q + \infty = \infty$
by (*simp-all add*: *plus-inat-def split*: *inat.splits*)

instance proof
fix *n m q* :: *inat*
show $n + m + q = n + (m + q)$
by (*cases n*, *auto*, *cases m*, *auto*, *cases q*, *auto*)
show $n + m = m + n$
by (*cases n*, *auto*, *cases m*, *auto*)
show $0 + n = n$
by (*cases n*) (*simp-all add*: *zero-inat-def*)
qed

end

lemma *plus-inat-0* [*simp*]:
$0 + (q::inat) = q$
$(q::inat) + 0 = q$
by (*simp-all add*: *plus-inat-def zero-inat-def split*: *inat.splits*)

lemma *plus-inat-number* [*simp*]:
(*number-of k* :: *inat*) + *number-of l* = (*if k* < *Int.Pls then number-of l*
else if l < *Int.Pls then number-of k else number-of* (*k* + *l*))
unfolding *number-of-inat-def plus-inat-simps nat-arith*(*1*) *if-distrib* [*symmetric*, *of - Fin*] **..**

lemma *iSuc-number* [*simp*]:
iSuc (*number-of k*) = (*if neg* (*number-of k* :: *int*) *then 1 else number-of* (*Int.succ k*))
unfolding *iSuc-number-of*
unfolding *one-inat-def number-of-inat-def Suc-nat-number-of if-distrib* [*symmetric*]
..

lemma *iSuc-plus-1*:
iSuc n = *n* + *1*

by (*cases n*) (*simp-all add*: *iSuc-Fin one-inat-def*)

lemma *plus-1-iSuc*:
1 + q = iSuc q
q + 1 = iSuc q
unfolding *iSuc-plus-1* **by** (*simp-all add*: *add-ac*)

49.4 Multiplication

instantiation *inat* :: *comm-semiring-1*
begin

definition
times-inat-def [*code del*]:
*m * n = (case m of ∞ ⇒ if n = 0 then 0 else ∞ | Fin m ⇒*
*(case n of ∞ ⇒ if m = 0 then 0 else ∞ | Fin n ⇒ Fin (m * n)))*

lemma *times-inat-simps* [*simp*, *code*]:
*Fin m * Fin n = Fin (m * n)*
*∞ * ∞ = ∞*
*∞ * Fin n = (if n = 0 then 0 else ∞)*
*Fin m * ∞ = (if m = 0 then 0 else ∞)*
unfolding *times-inat-def zero-inat-def*
by (*simp-all split*: *inat.split*)

instance proof
fix *a b c* :: *inat*
show *(a * b) * c = a * (b * c)*
unfolding *times-inat-def zero-inat-def*
by (*simp split*: *inat.split*)
show *a * b = b * a*
unfolding *times-inat-def zero-inat-def*
by (*simp split*: *inat.split*)
show *1 * a = a*
unfolding *times-inat-def zero-inat-def one-inat-def*
by (*simp split*: *inat.split*)
show *(a + b) * c = a * c + b * c*
unfolding *times-inat-def zero-inat-def*
by (*simp split*: *inat.split add*: *left-distrib*)
show *0 * a = 0*
unfolding *times-inat-def zero-inat-def*
by (*simp split*: *inat.split*)
show *a * 0 = 0*
unfolding *times-inat-def zero-inat-def*
by (*simp split*: *inat.split*)
show *(0*::*inat) ≠ 1*
unfolding *zero-inat-def one-inat-def*
by *simp*
qed

end

lemma *mult-iSuc*: *iSuc m * n = n + m * n*
unfolding *iSuc-plus-1* **by** (*simp add*: *algebra-simps*)

lemma *mult-iSuc-right*: *m * iSuc n = m + m * n*
unfolding *iSuc-plus-1* **by** (*simp add*: *algebra-simps*)

lemma *of-nat-eq-Fin*: *of-nat n = Fin n*
apply (*induct n*)
apply (*simp add*: *Fin-0*)
apply (*simp add*: *plus-1-iSuc iSuc-Fin*)
done

instance *inat* :: *semiring-char-0*
by *default* (*simp add*: *of-nat-eq-Fin*)

49.5 Ordering

instantiation *inat* :: *linordered-ab-semigroup-add*
begin

definition
[*code del*]: *m ≤ n = (case n of Fin n1 ⇒ (case m of Fin m1 ⇒ m1 ≤ n1 | ∞ ⇒ False)*
| ∞ ⇒ True)

definition
[*code del*]: *m < n = (case m of Fin m1 ⇒ (case n of Fin n1 ⇒ m1 < n1 | ∞ ⇒ True)*
| ∞ ⇒ False)

lemma *inat-ord-simps* [*simp*]:
Fin m ≤ Fin n ⟷ m ≤ n
Fin m < Fin n ⟷ m < n
q ≤ ∞
q < ∞ ⟷ q ≠ ∞
∞ ≤ q ⟷ q = ∞
∞ < q ⟷ False
by (*simp-all add*: *less-eq-inat-def less-inat-def split*: *inat.splits*)

lemma *inat-ord-code* [*code*]:
Fin m ≤ Fin n ⟷ m ≤ n
Fin m < Fin n ⟷ m < n
q ≤ ∞ ⟷ True
Fin m < ∞ ⟷ True
∞ ≤ Fin n ⟷ False
∞ < q ⟷ False

by *simp-all*

instance by *default*
(*auto simp add*: *less-eq-inat-def less-inat-def plus-inat-def split*: *inat.splits*)

end

instance *inat* :: *ordered-comm-semiring*
proof
fix *a b c* :: *inat*
assume $a \leq b$ **and** $0 \leq c$
thus $c * a \leq c * b$
unfolding *times-inat-def less-eq-inat-def zero-inat-def*
by (*simp split*: *inat.splits*)
qed

lemma *inat-ord-number* [*simp*]:
(*number-of m* :: *inat*) $\leq$ *number-of n* $\longleftrightarrow$ (*number-of m* :: *nat*) $\leq$ *number-of n*
(*number-of m* :: *inat*) $<$ *number-of n* $\longleftrightarrow$ (*number-of m* :: *nat*) $<$ *number-of n*
by (*simp-all add*: *number-of-inat-def*)

lemma *i0-lb* [*simp*]: (*0*::*inat*) $\leq n$
by (*simp add*: *zero-inat-def less-eq-inat-def split*: *inat.splits*)

lemma *i0-neq* [*simp*]: $n \leq$ (*0*::*inat*) $\longleftrightarrow n = 0$
by (*simp add*: *zero-inat-def less-eq-inat-def split*: *inat.splits*)

lemma *Infty-ileE* [*elim*!]: $\infty \leq$ *Fin m* $\Longrightarrow R$
by (*simp add*: *zero-inat-def less-eq-inat-def split*: *inat.splits*)

lemma *Infty-ilessE* [*elim*!]: $\infty <$ *Fin m* $\Longrightarrow R$
by *simp*

lemma *not-ilessi0* [*simp*]: $\neg\ n <$ (*0*::*inat*)
by (*simp add*: *zero-inat-def less-inat-def split*: *inat.splits*)

lemma *i0-eq* [*simp*]: (*0*::*inat*) $< n \longleftrightarrow n \neq 0$
by (*simp add*: *zero-inat-def less-inat-def split*: *inat.splits*)

lemma *iSuc-ile-mono* [*simp*]: *iSuc n* $\leq$ *iSuc m* $\longleftrightarrow n \leq m$
by (*simp add*: *iSuc-def less-eq-inat-def split*: *inat.splits*)

lemma *iSuc-mono* [*simp*]: *iSuc n* $<$ *iSuc m* $\longleftrightarrow n < m$
by (*simp add*: *iSuc-def less-inat-def split*: *inat.splits*)

lemma *ile-iSuc* [*simp*]: $n \leq$ *iSuc n*
by (*simp add*: *iSuc-def less-eq-inat-def split*: *inat.splits*)

lemma *not-iSuc-ilei0* [*simp*]: $\neg$ *iSuc n* $\leq$ *0*

by (*simp add*: *zero-inat-def iSuc-def less-eq-inat-def split*: *inat.splits*)

lemma *i0-iless-iSuc* [*simp*]: $0 < iSuc\ n$
by (*simp add*: *zero-inat-def iSuc-def less-inat-def split*: *inat.splits*)

lemma *ileI1*: $m < n \Longrightarrow iSuc\ m \leq n$
by (*simp add*: *iSuc-def less-eq-inat-def less-inat-def split*: *inat.splits*)

lemma *Suc-ile-eq*: $Fin\ (Suc\ m) \leq n \longleftrightarrow Fin\ m < n$
by (*cases n*) *auto*

lemma *iless-Suc-eq* [*simp*]: $Fin\ m < iSuc\ n \longleftrightarrow Fin\ m \leq n$
by (*auto simp add*: *iSuc-def less-inat-def split*: *inat.splits*)

lemma *min-inat-simps* [*simp*]:
$min\ (Fin\ m)\ (Fin\ n) = Fin\ (min\ m\ n)$
$min\ q\ 0 = 0$
$min\ 0\ q = 0$
$min\ q\ \infty = q$
$min\ \infty\ q = q$
by (*auto simp add*: *min-def*)

lemma *max-inat-simps* [*simp*]:
$max\ (Fin\ m)\ (Fin\ n) = Fin\ (max\ m\ n)$
$max\ q\ 0 = q$
$max\ 0\ q = q$
$max\ q\ \infty = \infty$
$max\ \infty\ q = \infty$
by (*simp-all add*: *max-def*)

lemma *Fin-ile*: $n \leq Fin\ m \Longrightarrow \exists k.\ n = Fin\ k$
by (*cases n*) *simp-all*

lemma *Fin-iless*: $n < Fin\ m \Longrightarrow \exists k.\ n = Fin\ k$
by (*cases n*) *simp-all*

lemma *chain-incr*: $\forall i.\ \exists j.\ Y\ i < Y\ j \Longrightarrow \exists j.\ Fin\ k < Y\ j$
apply (*induct-tac k*)
apply (*simp* (*no-asm*) *only*: *Fin-0*)
apply (*fast intro*: *le-less-trans* [*OF 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*) *le-less-trans*)
done

instantiation *inat* :: {*bot*, *top*}
begin

definition *bot-inat* :: *inat* **where**
bot-inat = *0*

definition *top-inat* :: *inat* **where**
top-inat = ∞

instance proof
qed (*simp-all add*: *bot-inat-def top-inat-def*)

end

49.6 Well-ordering

lemma *less-FinE*:
[| *n* < *Fin m*; !!*k*. *n* = *Fin k* ==> *k* < *m* ==> *P* |] ==> *P*
by (*induct n*) *auto*

lemma *less-InftyE*:
[| *n* < *Infty*; !!*k*. *n* = *Fin k* ==> *P* |] ==> *P*
by (*induct n*) *auto*

lemma *inat-less-induct*:
assumes *prem*: !!*n*. ∀ *m*::*inat*. *m* < *n* −−> *P m* ==> *P n* **shows** *P n*
proof −
have *P-Fin*: !!*k*. *P* (*Fin k*)
apply (*rule nat-less-induct*)
apply (*rule prem*, *clarify*)
apply (*erule less-FinE*, *simp*)
done
show *?thesis*
proof (*induct n*)
fix *nat*
show *P* (*Fin nat*) **by** (*rule P-Fin*)
next
show *P Infty*
apply (*rule prem*, *clarify*)
apply (*erule less-InftyE*)
apply (*simp add*: *P-Fin*)
done
qed
qed

instance *inat* :: *wellorder*
proof
fix *P* **and** *n*
assume *hyp*: (⋀*n*::*inat*. (⋀*m*::*inat*. *m* < *n* ⟹ *P m*) ⟹ *P n*)

show *P n* **by** (*blast intro*: *inat-less-induct hyp*)
qed

49.7 Traditional theorem names

lemmas *inat-defs* = *zero-inat-def one-inat-def number-of-inat-def iSuc-def plus-inat-def less-eq-inat-def less-inat-def*

lemmas *inat-splits* = *inat.splits*

end

50 Nested-Environment: Nested environments

theory *Nested-Environment*
imports *Main*
begin

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

datatype $('a, 'b, 'c)\ env =$
  Val $'a$
| *Env* $'b$ $'c \Rightarrow ('a, 'b, 'c)\ env\ option$

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

50.1 The lookup operation

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

primrec
  lookup :: $('a, 'b, 'c)\ env \Rightarrow 'c\ list \Rightarrow ('a, 'b, 'c)\ env\ option$
  and *lookup-option* :: $('a, 'b, 'c)\ env\ option \Rightarrow 'c\ list \Rightarrow ('a, 'b, 'c)\ env\ option$
where
  lookup (*Val a*) *xs* = (*if xs* = [] *then Some* (*Val a*) *else None*)

```
| lookup (Env b es) xs =
    (case xs of
      [] => Some (Env b es)
    | y # ys => lookup-option (es y) ys)
| lookup-option None xs = None
| lookup-option (Some e) xs = lookup e xs
```

hide-const *lookup-option*

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

```
theorem lookup-nil: lookup e [] = Some e
  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-lookup-option.simps [simp del]
  and lookup-simps [simp] = lookup-nil lookup-val-cons lookup-env-cons

theorem lookup-eq:
  lookup env xs =
    (case xs of
      [] => Some env
    | x # xs =>
      (case env of
        Val a => None
      | Env b es =>
          (case es x of
            None => None
          | Some e => lookup e xs)))
  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
```

50.2 The update operation

Update at a certain position in a nested environment may either delete an existing entry, or overwrite an existing one. Note that update at undefined positions is simple absorbed, i.e. the environment is left unchanged.

```
primrec
  update :: 'c list => ('a, 'b, 'c) env option
    => ('a, 'b, 'c) env => ('a, 'b, 'c) env
  and update-option :: 'c list => ('a, 'b, 'c) env option
    => ('a, 'b, 'c) env option => ('a, 'b, 'c) env option where
    update xs opt (Val a) =
      (if xs = [] then (case opt of None => Val a | Some e => e)
      else Val a)
  | update xs opt (Env b es) =
      (case xs of
        [] => (case opt of None => Env b es | Some e => e)
      | y # ys => Env b (es (y := update-option ys opt (es y))))
  | update-option xs opt None =
      (if xs = [] then opt else None)
  | update-option xs opt (Some e) =
```

(*if xs = [] then opt else Some (update xs opt e)*)

hide-const *update-option*

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

theorem *update-nil-none*: *update [] None env = env*
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-update-option.simps* [*simp del*]
and *update-simps* [*simp*] = *update-nil-none update-nil-some*
update-cons-val update-cons-nil-env update-cons-cons-env

lemma *update-eq*:
update xs opt env =
(case xs of
[] =>
(case opt of
None => env
| Some e => e)
| x # xs =>
(case env of
Val a => Val a
| Env b es =>
(case xs of
[] => Env b (es (x := opt))
| y # ys =>
Env b (es (x :=
(case es x of
None => None
| Some e => Some (update (y # ys) opt e)))))))

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
```

Environments and code generation

lemma [*code*, *code del*]:
fixes *e1 e2* :: ('*b*::*eq*, '*a*::*eq*, '*c*::*eq*) *env*
shows *eq-class.eq e1 e2* ⟷ *eq-class.eq e1 e2* **..**

lemma *eq-env-code* [*code*]:
fixes *x y* :: '*a*::*eq*
and *f g* :: '*c*::{*eq*, *finite*} ⇒ ('*b*::*eq*, '*a*, '*c*) *env option*
shows *eq-class.eq* (*Env x f*) (*Env y g*) ⟷
eq-class.eq x y ∧ (∀ *z*∈*UNIV*. *case f z*
of None ⇒ (*case g z*
of None ⇒ *True* | *Some* - ⇒ *False*)
| *Some a* ⇒ (*case g z*
of None ⇒ *False* | *Some b* ⇒ *eq-class.eq a b*)) (**is** *?env*)
and *eq-class.eq* (*Val a*) (*Val b*) ⟷ *eq-class.eq a b*
and *eq-class.eq* (*Val a*) (*Env y g*) ⟷ *False*
and *eq-class.eq* (*Env x f*) (*Val b*) ⟷ *False*
proof (*unfold eq*)
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 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*)) **by** *simp*
qed *simp-all*

lemma [*code*, *code del*]:
(*Code-Evaluation.term-of* :: ('*a*::{*term-of*, *type*}, '*b*::{*term-of*, *type*}, '*c*::{*term-of*, *type*}) *env* ⇒ *term*) = *Code-Evaluation.term-of* **..**

end

51 Numeral-Type: Numeral Syntax for Types

theory *Numeral-Type*
imports *Main*
begin

51.1 Preliminary lemmas

lemma (**in** *type-definition*) *univ*:
  $UNIV = Abs\ `\ A$
proof
  show $Abs\ `\ A \subseteq UNIV$ **by** (*rule subset-UNIV*)
  show $UNIV \subseteq Abs\ `\ A$
  proof
    fix $x :: 'b$
    have $x = Abs\ (Rep\ x)$ **by** (*rule Rep-inverse* [*symmetric*])
    moreover have $Rep\ x \in A$ **by** (*rule Rep*)
    ultimately show $x \in Abs\ `\ A$ **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*)

51.2 Cardinalities of types

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

translations $CARD('t) => CONST\ card\ (CONST\ UNIV :: 't\ set)$

typed-print-translation ⟪
let
  fun card-univ-tr' show-sorts - [*Const* (@{*const-syntax UNIV*}, *Type*(-, [*T*, -]))] =
    Syntax.const @{*syntax-const -type-card*} $ *Syntax.term-of-typ show-sorts T*;
in [(@{*const-syntax card*}, *card-univ-tr'*)]
end
⟫

lemma *card-unit* [*simp*]: $CARD(unit) = 1$
  unfolding *UNIV-unit* **by** *simp*

lemma *card-bool* [*simp*]: $CARD(bool) = 2$
  unfolding *UNIV-bool* **by** *simp*

lemma *card-prod* [*simp*]: $CARD('a \times 'b) = CARD('a::finite) * CARD('b::finite)$
  unfolding *UNIV-Times-UNIV* [*symmetric*] **by** (*simp only*: *card-cartesian-product*)

lemma *card-sum* [*simp*]: $CARD('a + 'b) = CARD('a::finite) + CARD('b::finite)$
  unfolding *UNIV-Plus-UNIV* [*symmetric*] **by** (*simp only*: *finite card-Plus*)

lemma *card-option* [*simp*]: *CARD*('*a option*) = *Suc CARD*('*a*::*finite*)
unfolding *UNIV-option-conv*
apply (*subgoal-tac* (*None*::'*a option*) ∉ *range Some*)
apply (*simp add*: *card-image*)
apply *fast*
done

lemma *card-set* [*simp*]: *CARD*('*a set*) = *2* ^ *CARD*('*a*::*finite*)
unfolding *Pow-UNIV* [*symmetric*]
by (*simp only*: *card-Pow finite numeral-2-eq-2*)

lemma *card-nat* [*simp*]: *CARD*(*nat*) = *0*
by (*simp add*: *infinite-UNIV-nat card-eq-0-iff*)

51.3 Classes with at least 1 and 2

Class finite already captures "at least 1"

lemma *zero-less-card-finite* [*simp*]: *0* < *CARD*('*a*::*finite*)
unfolding *neq0-conv* [*symmetric*] **by** *simp*

lemma *one-le-card-finite* [*simp*]: *Suc 0* ≤ *CARD*('*a*::*finite*)
by (*simp add*: *less-Suc-eq-le* [*symmetric*])

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*

lemma *one-less-int-card*: *1* < *int CARD*('*a*::*card2*)
using *one-less-card* [**where** '*a*='*a*] **by** *simp*

51.4 Numeral Types

typedef (**open**) *num0* = *UNIV* :: *nat set* **..**
typedef (**open**) *num1* = *UNIV* :: *unit set* **..**

typedef (**open**) '*a bit0* = {*0* ..< *2* ∗ *int CARD*('*a*::*finite*)}
proof
show *0* ∈ {*0* ..< *2* ∗ *int CARD*('*a*)}
by *simp*
qed

typedef (**open**) '*a bit1* = {*0* ..< *1* + *2* ∗ *int CARD*('*a*::*finite*)}
proof
show *0* ∈ {*0* ..< *1* + *2* ∗ *int CARD*('*a*)}
by *simp*

qed

lemma *card-num0* [*simp*]: *CARD* (*num0*) = *0*
unfolding *type-definition.card* [*OF type-definition-num0*]
by *simp*

lemma *card-num1* [*simp*]: *CARD*(*num1*) = *1*
unfolding *type-definition.card* [*OF type-definition-num1*]
by (*simp only*: *card-unit*)

lemma *card-bit0* [*simp*]: *CARD*('*a bit0*) = *2* ∗ *CARD*('*a*::*finite*)
unfolding *type-definition.card* [*OF type-definition-bit0*]
by *simp*

lemma *card-bit1* [*simp*]: *CARD*('*a bit1*) = *Suc* (*2* ∗ *CARD*('*a*::*finite*))
unfolding *type-definition.card* [*OF type-definition-bit1*]
by *simp*

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*) *card2*
proof
show *finite* (*UNIV*::'*a bit0 set*)
unfolding *type-definition.univ* [*OF type-definition-bit0*]
by *simp*
show *2* ≤ *CARD*('*a bit0*)
by *simp*
qed

instance *bit1* :: (*finite*) *card2*
proof
show *finite* (*UNIV*::'*a bit1 set*)
unfolding *type-definition.univ* [*OF type-definition-bit1*]
by *simp*
show *2* ≤ *CARD*('*a bit1*)
by *simp*
qed

51.5 Locale for modular arithmetic subtypes

locale *mod-type* =
fixes *n* :: *int*
and *Rep* :: '*a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*} ⇒ *int*
and *Abs* :: *int* ⇒ '*a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*}

assumes *type*: *type-definition Rep Abs* $\{0..<n\}$
and *size1*: $1 < n$
and *zero-def*: $0 = Abs\ 0$
and *one-def*: $1 = Abs\ 1$
and *add-def*: $x + y = Abs\ ((Rep\ x + Rep\ y)\ mod\ n)$
and *mult-def*: $x * y = Abs\ ((Rep\ x * Rep\ y)\ mod\ n)$
and *diff-def*: $x - y = Abs\ ((Rep\ x - Rep\ y)\ mod\ n)$
and *minus-def*: $- x = Abs\ ((-\ Rep\ x)\ mod\ n)$
begin

lemma *size0*: $0 < n$
using *size1* **by** *simp*

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

lemma *Rep-less-n*: $Rep\ x < n$
by (*rule type-definition.Rep* [*OF type*, *simplified*, *THEN conjunct2*])

lemma *Rep-le-n*: $Rep\ x \leq n$
by (*rule Rep-less-n* [*THEN order-less-imp-le*])

lemma *Rep-inject-sym*: $x = y \longleftrightarrow Rep\ x = Rep\ y$
by (*rule type-definition.Rep-inject* [*OF type*, *symmetric*])

lemma *Rep-inverse*: $Abs\ (Rep\ x) = x$
by (*rule type-definition.Rep-inverse* [*OF type*])

lemma *Abs-inverse*: $m \in \{0..<n\} \Longrightarrow Rep\ (Abs\ m) = m$
by (*rule type-definition.Abs-inverse* [*OF type*])

lemma *Rep-Abs-mod*: $Rep\ (Abs\ (m\ mod\ n)) = m\ mod\ n$
by (*simp add*: *Abs-inverse pos-mod-conj* [*OF size0*])

lemma *Rep-Abs-0*: $Rep\ (Abs\ 0) = 0$
by (*simp add*: *Abs-inverse size0*)

lemma *Rep-0*: $Rep\ 0 = 0$
by (*simp add*: *zero-def Rep-Abs-0*)

lemma *Rep-Abs-1*: $Rep\ (Abs\ 1) = 1$
by (*simp add*: *Abs-inverse size1*)

lemma *Rep-1*: $Rep\ 1 = 1$
by (*simp add*: *one-def Rep-Abs-1*)

lemma *Rep-mod*: $Rep\ x\ mod\ n = Rep\ x$
apply (*rule-tac x*=*x* **in** *type-definition.Abs-cases* [*OF type*])
apply (*simp add*: *type-definition.Abs-inverse* [*OF type*])

apply (*simp add*: *mod-pos-pos-trivial*)
done

lemmas *Rep-simps* =
Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1

lemma *comm-ring-1*: *OFCLASS*('*a*, *comm-ring-1-class*)
apply (*intro-classes*, *unfold definitions*)
apply (*simp-all add*: *Rep-simps zmod-simps field-simps*)
done

end

locale *mod-ring* = *mod-type* +
constrains *n* :: *int*
and *Rep* :: '*a*::{*number-ring*} ⇒ *int*
and *Abs* :: *int* ⇒ '*a*::{*number-ring*}
begin

lemma *of-nat-eq*: *of-nat k* = *Abs* (*int k mod n*)
apply (*induct k*)
apply (*simp add*: *zero-def*)
apply (*simp add*: *Rep-simps add-def one-def zmod-simps add-ac*)
done

lemma *of-int-eq*: *of-int z* = *Abs* (*z mod n*)
apply (*cases z rule*: *int-diff-cases*)
apply (*simp add*: *Rep-simps of-nat-eq diff-def zmod-simps*)
done

lemma *Rep-number-of*:
Rep (*number-of w*) = *number-of w mod n*
by (*simp add*: *number-of-eq of-int-eq Rep-Abs-mod*)

lemma *iszero-number-of*:
iszero (*number-of w*::'*a*) ⟷ *number-of w mod n* = *0*
by (*simp add*: *Rep-simps number-of-eq of-int-eq iszero-def zero-def*)

lemma *cases*:
assumes *1*: ⋀*z*. ⟦(*x*::'*a*) = *of-int z*; *0* ≤ *z*; *z* < *n*⟧ ⟹ *P*
shows *P*
apply (*cases x rule*: *type-definition.Abs-cases* [*OF type*])
apply (*rule-tac z*=*y* **in** *1*)
apply (*simp-all add*: *of-int-eq mod-pos-pos-trivial*)
done

lemma *induct*:
(⋀*z*. ⟦*0* ≤ *z*; *z* < *n*⟧ ⟹ *P* (*of-int z*)) ⟹ *P* (*x*::'*a*)
by (*cases x rule*: *cases*) *simp*

end

51.6 Number ring instances

Unfortunately a number ring instance is not possible for *num1*, since 0 and 1 are not distinct.

instantiation *num1* :: {*comm-ring*,*comm-monoid-mult*,*number*}
begin

lemma *num1-eq-iff*: (*x*::*num1*) = (*y*::*num1*) ⟷ *True*
by (*induct x*, *induct y*) *simp*

instance proof
qed (*simp-all add*: *num1-eq-iff*)

end

instantiation
bit0 **and** *bit1* :: (*finite*) {*zero*,*one*,*plus*,*times*,*uminus*,*minus*}
begin

definition *Abs-bit0'* :: *int* ⇒ *'a bit0* **where**
Abs-bit0' x = *Abs-bit0* (*x mod int CARD*(*'a bit0*))

definition *Abs-bit1'* :: *int* ⇒ *'a bit1* **where**
Abs-bit1' x = *Abs-bit1* (*x mod int CARD*(*'a bit1*))

definition *0* = *Abs-bit0 0*
definition *1* = *Abs-bit0 1*
definition *x* + *y* = *Abs-bit0'* (*Rep-bit0 x* + *Rep-bit0 y*)
definition *x* ∗ *y* = *Abs-bit0'* (*Rep-bit0 x* ∗ *Rep-bit0 y*)
definition *x* − *y* = *Abs-bit0'* (*Rep-bit0 x* − *Rep-bit0 y*)
definition − *x* = *Abs-bit0'* (− *Rep-bit0 x*)

definition *0* = *Abs-bit1 0*
definition *1* = *Abs-bit1 1*
definition *x* + *y* = *Abs-bit1'* (*Rep-bit1 x* + *Rep-bit1 y*)
definition *x* ∗ *y* = *Abs-bit1'* (*Rep-bit1 x* ∗ *Rep-bit1 y*)
definition *x* − *y* = *Abs-bit1'* (*Rep-bit1 x* − *Rep-bit1 y*)
definition − *x* = *Abs-bit1'* (− *Rep-bit1 x*)

instance ..

end

interpretation *bit0*:
mod-type int CARD(*'a*::*finite bit0*)
Rep-bit0 :: *'a*::*finite bit0* ⇒ *int*

Abs-bit0 :: *int* ⇒ *'a*::*finite bit0*
apply (*rule mod-type.intro*)
apply (*simp add*: *int-mult type-definition-bit0*)
apply (*rule one-less-int-card*)
apply (*rule zero-bit0-def*)
apply (*rule one-bit0-def*)
apply (*rule plus-bit0-def* [*unfolded Abs-bit0'-def*])
apply (*rule times-bit0-def* [*unfolded Abs-bit0'-def*])
apply (*rule minus-bit0-def* [*unfolded Abs-bit0'-def*])
apply (*rule uminus-bit0-def* [*unfolded Abs-bit0'-def*])
done

interpretation *bit1*:
mod-type int CARD('*a*::*finite bit1*)
Rep-bit1 :: *'a*::*finite bit1* ⇒ *int*
Abs-bit1 :: *int* ⇒ *'a*::*finite bit1*
apply (*rule mod-type.intro*)
apply (*simp add*: *int-mult type-definition-bit1*)
apply (*rule one-less-int-card*)
apply (*rule zero-bit1-def*)
apply (*rule one-bit1-def*)
apply (*rule plus-bit1-def* [*unfolded Abs-bit1'-def*])
apply (*rule times-bit1-def* [*unfolded Abs-bit1'-def*])
apply (*rule minus-bit1-def* [*unfolded Abs-bit1'-def*])
apply (*rule uminus-bit1-def* [*unfolded Abs-bit1'-def*])
done

instance *bit0* :: (*finite*) *comm-ring-1*
by (*rule bit0.comm-ring-1*)+

instance *bit1* :: (*finite*) *comm-ring-1*
by (*rule bit1.comm-ring-1*)+

instantiation *bit0* **and** *bit1* :: (*finite*) *number-ring*
begin

definition (*number-of w* :: - *bit0*) = *of-int w*

definition (*number-of w* :: - *bit1*) = *of-int w*

instance proof
qed (*rule number-of-bit0-def number-of-bit1-def*)+

end

interpretation *bit0*:
mod-ring int CARD('*a*::*finite bit0*)
Rep-bit0 :: *'a*::*finite bit0* ⇒ *int*
Abs-bit0 :: *int* ⇒ *'a*::*finite bit0*

..

interpretation *bit1*:
mod-ring int CARD('a::finite bit1)
Rep-bit1 :: 'a::finite bit1 ⇒ int
Abs-bit1 :: int ⇒ 'a::finite bit1
..

Set up cases, induction, and arithmetic

lemmas *bit0-cases* [*case-names of-int*, *cases type*: *bit0*] = *bit0.cases*
lemmas *bit1-cases* [*case-names of-int*, *cases type*: *bit1*] = *bit1.cases*

lemmas *bit0-induct* [*case-names of-int*, *induct type*: *bit0*] = *bit0.induct*
lemmas *bit1-induct* [*case-names of-int*, *induct type*: *bit1*] = *bit1.induct*

lemmas *bit0-iszero-number-of* [*simp*] = *bit0.iszero-number-of*
lemmas *bit1-iszero-number-of* [*simp*] = *bit1.iszero-number-of*

51.7 Syntax

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

translations
(*type*) *1* == (*type*) *num1*
(*type*) *0* == (*type*) *num0*

parse-translation ⟪
let
fun mk-bintype n =
let
fun mk-bit 0 = Syntax.const @{*type-syntax bit0*}
| *mk-bit 1 = Syntax.const* @{*type-syntax bit1*};
fun bin-of n =
if n = 1 then Syntax.const @{*type-syntax num1*}
else if n = 0 then Syntax.const @{*type-syntax num0*}
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 (*the* (*Int.fromString str*))
| *numeral-tr* (∗*-NumeralType*∗) *ts = raise TERM* (*numeral-tr*, *ts*);

in [(@{*syntax-const -NumeralType*}, *numeral-tr*)] *end*;
⟫

print-translation ⟪
let
fun int-of [] = *0*
  | *int-of* (*b* :: *bs*) = *b* + *2* ∗ *int-of bs*;

fun bin-of (*Const* (@{*type-syntax num0*}, -)) = []
  | *bin-of* (*Const* (@{*type-syntax num1*}, -)) = [*1*]
  | *bin-of* (*Const* (@{*type-syntax bit0*}, -) $ *bs*) = *0* :: *bin-of bs*
  | *bin-of* (*Const* (@{*type-syntax 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 @{*syntax-const -NumeralType*} $ *Syntax.free num*
    end
  | *bit-tr'* *b* - = *raise Match*;
in [(@{*type-syntax bit0*}, *bit-tr' 0*), (@{*type-syntax bit1*}, *bit-tr' 1*)] *end*;
⟫

51.8 Examples

lemma *CARD*(*0*) = *0* **by** *simp*
lemma *CARD*(*17*) = *17* **by** *simp*
lemma *8* ∗ *11* ^ *3* − *6* = (*2*::*5*) **by** *simp*

end

52 Option-ord: Canonical order on option type

theory *Option-ord*
imports *Option Main*
begin

instantiation *option* :: (*preorder*) *preorder*
begin

definition *less-eq-option* **where**
  [*code del*]: *x* ≤ *y* ⟷ (*case x of None* ⇒ *True* | *Some x* ⇒ (*case y of None* ⇒ *False* | *Some y* ⇒ *x* ≤ *y*))

definition *less-option* **where**

[*code del*]: $x < y \longleftrightarrow$ (*case y of None* $\Rightarrow$ *False* | *Some y* $\Rightarrow$ (*case x of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ $x < y$))

lemma *less-eq-option-None* [*simp*]: *None* $\leq x$
by (*simp add*: *less-eq-option-def*)

lemma *less-eq-option-None-code* [*code*]: *None* $\leq x \longleftrightarrow$ *True*
by *simp*

lemma *less-eq-option-None-is-None*: $x \leq$ *None* $\Longrightarrow x =$ *None*
by (*cases x*) (*simp-all add*: *less-eq-option-def*)

lemma *less-eq-option-Some-None* [*simp*, *code*]: *Some x* $\leq$ *None* $\longleftrightarrow$ *False*
by (*simp add*: *less-eq-option-def*)

lemma *less-eq-option-Some* [*simp*, *code*]: *Some x* $\leq$ *Some y* $\longleftrightarrow x \leq y$
by (*simp add*: *less-eq-option-def*)

lemma *less-option-None* [*simp*, *code*]: $x <$ *None* $\longleftrightarrow$ *False*
by (*simp add*: *less-option-def*)

lemma *less-option-None-is-Some*: *None* $< x \Longrightarrow \exists z.\ x =$ *Some z*
by (*cases x*) (*simp-all add*: *less-option-def*)

lemma *less-option-None-Some* [*simp*]: *None* $<$ *Some x*
by (*simp add*: *less-option-def*)

lemma *less-option-None-Some-code* [*code*]: *None* $<$ *Some x* $\longleftrightarrow$ *True*
by *simp*

lemma *less-option-Some* [*simp*, *code*]: *Some x* $<$ *Some y* $\longleftrightarrow x < y$
by (*simp add*: *less-option-def*)

instance proof
qed (*auto simp add*: *less-eq-option-def less-option-def less-le-not-le elim*: *order-trans split*: *option.splits*)

end

instance *option* :: (*order*) *order* **proof**
qed (*auto simp add*: *less-eq-option-def less-option-def split*: *option.splits*)

instance *option* :: (*linorder*) *linorder* **proof**
qed (*auto simp add*: *less-eq-option-def less-option-def split*: *option.splits*)

instantiation *option* :: (*preorder*) *bot*
begin

definition *bot* = *None*

```
instance proof
qed (simp add: bot-option-def)

end

instantiation option :: (top) top
begin

definition top = Some top

instance proof
qed (simp add: top-option-def less-eq-option-def split: option.split)

end

instance option :: (wellorder) wellorder proof
  fix P :: 'a option ⇒ bool and z :: 'a option
  assume H: ⋀x. (⋀y. y < x ⟹ P y) ⟹ P x
  have P None by (rule H) simp
  then have P-Some [case-names Some]:
    ⋀z. (⋀x. z = Some x ⟹ (P o Some) x) ⟹ P z
  proof −
    fix z
    assume ⋀x. z = Some x ⟹ (P o Some) x
    with ‹P None› show P z by (cases z) simp-all
  qed
  show P z proof (cases z rule: P-Some)
    case (Some w)
    show (P o Some) w proof (induct rule: less-induct)
      case (less x)
      have P (Some x) proof (rule H)
        fix y :: 'a option
        assume y < Some x
        show P y proof (cases y rule: P-Some)
          case (Some v) with ‹y < Some x› have v < x by simp
          with less show (P o Some) v .
        qed
      qed
      then show ?case by simp
    qed
  qed
qed

end
```

53 Permutation: Permutations

theory *Permutation*
imports *Main Multiset*
begin

inductive
perm :: *'a list => 'a list => bool* (- <~~> - [*50*, *50*] *50*)
where
Nil [*intro*!]: [] <~~> []
| *swap* [*intro*!]: *y # x # l* <~~> *x # y # l*
| *Cons* [*intro*!]: *xs* <~~> *ys* ==> *z # xs* <~~> *z # ys*
| *trans* [*intro*]: *xs* <~~> *ys* ==> *ys* <~~> *zs* ==> *xs* <~~> *zs*

lemma *perm-refl* [*iff*]: *l* <~~> *l*
by (*induct l*) *auto*

53.1 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*

53.2 Ways of making new permutations

We can insert the head anywhere in the list.

lemma *perm-append-Cons*: *a # xs @ ys* <~~> *xs @ a # ys*
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*)

53.3 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*)

53.4 Removing elements

lemma *perm-remove*: *x* ∈ *set ys* ==> *ys* <~~> *x* # *remove1 x ys*
by (*induct ys*) *auto*

Congruence rule

lemma *perm-remove-perm*: *xs* <~~> *ys* ==> *remove1 z xs* <~~> *remove1 z ys*
by (*induct pred*: *perm*) *auto*

lemma *remove-hd* [*simp*]: *remove1 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* = *remove1 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 (*subgoal-tac length* (*remdups xs*) < *Suc* (*length xs*))
apply *simp*
apply (*subgoal-tac length* (*remdups xs*) ≤ *length xs*)
apply *simp*
apply (*rule length-remdups-leq*)
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

54 Poly-Deriv: Polynomials and Differentiation

theory *Poly-Deriv*
imports *Deriv Polynomial*
begin

54.1 Derivatives of univariate polynomials

definition
pderiv :: *'a*::*real-normed-field poly* ⇒ *'a poly* **where**
pderiv = *poly-rec 0* (λ*a p p'*. *p* + *pCons 0 p'*)

lemma *pderiv-0* [*simp*]: *pderiv 0* = *0*
unfolding *pderiv-def* **by** (*simp add*: *poly-rec-0*)

lemma *pderiv-pCons*: *pderiv* (*pCons a p*) = *p* + *pCons 0* (*pderiv p*)
unfolding *pderiv-def* **by** (*simp add*: *poly-rec-pCons*)

lemma *coeff-pderiv*: *coeff* (*pderiv p*) *n* = *of-nat* (*Suc n*) ∗ *coeff p* (*Suc n*)
apply (*induct p arbitrary*: *n*, *simp*)
apply (*simp add*: *pderiv-pCons coeff-pCons algebra-simps split*: *nat.split*)
done

lemma *pderiv-eq-0-iff*: *pderiv p* = *0* ⟷ *degree p* = *0*

apply (*rule iffI*)
apply (*cases p*, *simp*)
apply (*simp add*: *expand-poly-eq coeff-pderiv del*: *of-nat-Suc*)
apply (*simp add*: *expand-poly-eq coeff-pderiv coeff-eq-0*)
done

lemma *degree-pderiv*: *degree* (*pderiv p*) = *degree p* − *1*
apply (*rule order-antisym* [*OF degree-le*])
apply (*simp add*: *coeff-pderiv coeff-eq-0*)
apply (*cases degree p*, *simp*)
apply (*rule le-degree*)
apply (*simp add*: *coeff-pderiv del*: *of-nat-Suc*)
apply (*rule subst*, *assumption*)
apply (*rule leading-coeff-neq-0*, *clarsimp*)
done

lemma *pderiv-singleton* [*simp*]: *pderiv* [:*a*:] = *0*
by (*simp add*: *pderiv-pCons*)

lemma *pderiv-add*: *pderiv* (*p* + *q*) = *pderiv p* + *pderiv q*
by (*rule poly-ext*, *simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-minus*: *pderiv* (− *p*) = − *pderiv p*
by (*rule poly-ext*, *simp add*: *coeff-pderiv*)

lemma *pderiv-diff*: *pderiv* (*p* − *q*) = *pderiv p* − *pderiv q*
by (*rule poly-ext*, *simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-smult*: *pderiv* (*smult a p*) = *smult a* (*pderiv p*)
by (*rule poly-ext*, *simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-mult*: *pderiv* (*p* ∗ *q*) = *p* ∗ *pderiv q* + *q* ∗ *pderiv p*
apply (*induct p*)
apply *simp*
apply (*simp add*: *pderiv-add pderiv-smult pderiv-pCons algebra-simps*)
done

lemma *pderiv-power-Suc*:
pderiv (*p* ^ *Suc n*) = *smult* (*of-nat* (*Suc n*)) (*p* ^ *n*) ∗ *pderiv p*
apply (*induct n*)
apply *simp*
apply (*subst power-Suc*)
apply (*subst pderiv-mult*)
apply (*erule ssubst*)
apply (*simp add*: *smult-add-left algebra-simps*)
done

lemma *DERIV-cmult2*: *DERIV f x* :> *D* ==> *DERIV* (%*x*. (*f x*) ∗ *c* :: *real*) *x*
:> *D* ∗ *c*

by (*simp add*: *DERIV-cmult mult-commute* [*of* - *c*])

lemma *DERIV-pow2*: *DERIV* (%*x*. *x* ^ *Suc n*) *x* :> *real* (*Suc n*) * (*x* ^ *n*)
by (*rule lemma-DERIV-subst*, *rule DERIV-pow*, *simp*)
declare *DERIV-pow2* [*simp*] *DERIV-pow* [*simp*]

lemma *DERIV-add-const*: *DERIV f x* :> *D* ==> *DERIV* (%*x*. *a* + *f x* :: 'a::*real-normed-field*) *x* :> *D*
by (*rule lemma-DERIV-subst*, *rule DERIV-add*, *auto*)

lemma *poly-DERIV*[*simp*]: *DERIV* (%*x*. *poly p x*) *x* :> *poly* (*pderiv p*) *x*
by (*induct p*, *auto intro*!: *DERIV-intros simp add*: *pderiv-pCons*)

Consequences of the derivative theorem above

lemma *poly-differentiable*[*simp*]: (%*x*. *poly p x*) *differentiable* (*x*::*real*)
apply (*simp add*: *differentiable-def*)
apply (*blast intro*: *poly-DERIV*)
done

lemma *poly-isCont*[*simp*]: *isCont* (%*x*. *poly p x*) (*x*::*real*)
by (*rule poly-DERIV* [*THEN DERIV-isCont*])

lemma *poly-IVT-pos*: [| *a* < *b*; *poly p* (*a*::*real*) < *0*; *0* < *poly p b* |]
==> ∃ *x*. *a* < *x* & *x* < *b* & (*poly p x* = *0*)
apply (*cut-tac f* = %*x*. *poly p x* **and** *a* = *a* **and** *b* = *b* **and** *y* = *0* **in** *IVT-objl*)
apply (*auto simp add*: *order-le-less*)
done

lemma *poly-IVT-neg*: [| (*a*::*real*) < *b*; *0* < *poly p a*; *poly p b* < *0* |]
==> ∃ *x*. *a* < *x* & *x* < *b* & (*poly p x* = *0*)
by (*insert poly-IVT-pos* [**where** *p* = - *p*]) *simp*

lemma *poly-MVT*: (*a*::*real*) < *b* ==>
∃ *x*. *a* < *x* & *x* < *b* & (*poly p b* - *poly p a* = (*b* - *a*) * *poly* (*pderiv p*) *x*)
apply (*drule-tac f* = *poly p* **in** *MVT*, *auto*)
apply (*rule-tac x* = *z* **in** *exI*)
apply (*auto simp add*: *real-mult-left-cancel poly-DERIV* [*THEN DERIV-unique*])
done

Lemmas for Derivatives

lemma *order-unique-lemma*:
fixes *p* :: 'a::*idom poly*
assumes [:−*a*, *1*:] ^ *n dvd p* ∧ ¬ [:−*a*, *1*:] ^ *Suc n dvd p*
shows *n* = *order a p*
unfolding *Polynomial.order-def*
apply (*rule Least-equality* [*symmetric*])
apply (*rule assms* [*THEN conjunct2*])
apply (*erule contrapos-np*)
apply (*rule power-le-dvd*)

apply (*rule assms* [*THEN conjunct1*])
apply *simp*
done

lemma *lemma-order-pderiv1*:
  pderiv ([:− *a*, *1*:] ^ *Suc n* ∗ *q*) = [:− *a*, *1*:] ^ *Suc n* ∗ *pderiv q* +
    smult (*of-nat* (*Suc n*)) (*q* ∗ [:− *a*, *1*:] ^ *n*)
apply (*simp only*: *pderiv-mult pderiv-power-Suc*)
apply (*simp del*: *power-Suc of-nat-Suc add*: *pderiv-pCons*)
done

lemma *dvd-add-cancel1*:
  fixes *a b c* :: *'a*::*comm-ring-1*
  shows *a dvd b* + *c* ⟹ *a dvd b* ⟹ *a dvd c*
  by (*drule* (*1*) *Rings.dvd-diff*, *simp*)

lemma *lemma-order-pderiv* [*rule-format*]:
    ∀ *p q a*. *0* < *n* &
      pderiv p ≠ *0* &
      p = [:− *a*, *1*:] ^ *n* ∗ *q* & ~ [:− *a*, *1*:] *dvd q*
      −−> *n* = *Suc* (*order a* (*pderiv p*))
 apply (*cases n*, *safe*, *rename-tac n p q a*)
 apply (*rule order-unique-lemma*)
 apply (*rule conjI*)
  apply (*subst lemma-order-pderiv1*)
  apply (*rule dvd-add*)
   apply (*rule dvd-mult2*)
   apply (*rule le-imp-power-dvd*, *simp*)
  apply (*rule dvd-smult*)
  apply (*rule dvd-mult*)
  apply (*rule dvd-refl*)
 apply (*subst lemma-order-pderiv1*)
 apply (*erule contrapos-nn*) **back**
 apply (*subgoal-tac* [:− *a*, *1*:] ^ *Suc n dvd q* ∗ [:− *a*, *1*:] ^ *n*)
  apply (*simp del*: *mult-pCons-left*)
 apply (*drule dvd-add-cancel1*)
  apply (*simp del*: *mult-pCons-left*)
 apply (*drule dvd-smult-cancel*, *simp del*: *of-nat-Suc*)
 apply *assumption*
done

lemma *order-decomp*:
    p ≠ *0*
     ==> ∃ *q*. *p* = [:−*a*, *1*:] ^ (*order a p*) ∗ *q* &
           ~([:−*a*, *1*:] *dvd q*)
apply (*drule order* [**where** *a*=*a*])
apply (*erule conjE*)
apply (*erule dvdE*)
apply (*rule exI*)

apply (*rule conjI*, *assumption*)
apply (*erule contrapos-nn*)
apply (*erule ssubst*) **back**
apply (*subst power-Suc2*)
apply (*erule mult-dvd-mono* [*OF dvd-refl*])
done

lemma *order-pderiv*: [| *pderiv p* ≠ *0*; *order a p* ≠ *0* |]
    ==> (*order a p* = *Suc* (*order a* (*pderiv p*)))
apply (*case-tac p* = *0*, *simp*)
apply (*drule-tac a* = *a* **and** *p* = *p* **in** *order-decomp*)
using *neq0-conv*
apply (*blast intro*: *lemma-order-pderiv*)
done

lemma *order-mult*: *p* ∗ *q* ≠ *0* ⟹ *order a* (*p* ∗ *q*) = *order a p* + *order a q*
proof −
  def *i* ≡ *order a p*
  def *j* ≡ *order a q*
  def *t* ≡ [:−*a*, *1*:]
  have *t-dvd-iff*: ⋀*u*. *t dvd u* ⟷ *poly u a* = *0*
    unfolding *t-def* **by** (*simp add*: *dvd-iff-poly-eq-0*)
  assume *p* ∗ *q* ≠ *0*
  then show *order a* (*p* ∗ *q*) = *i* + *j*
    apply *clarsimp*
    apply (*drule order* [**where** *a*=*a* **and** *p*=*p*, *folded i-def t-def*])
    apply (*drule order* [**where** *a*=*a* **and** *p*=*q*, *folded j-def t-def*])
    apply *clarify*
    apply (*rule order-unique-lemma* [*symmetric*], *fold t-def*)
    apply (*erule dvdE*)+
    apply (*simp add*: *power-add t-dvd-iff*)
    done
qed

Now justify the standard squarefree decomposition, i.e. f / gcd(f,f').

lemma *order-divides*: [:−*a*, *1*:] ^ *n dvd p* ⟷ *p* = *0* ∨ *n* ≤ *order a p*
apply (*cases p* = *0*, *auto*)
apply (*drule order-2* [**where** *a*=*a* **and** *p*=*p*])
apply (*erule contrapos-np*)
apply (*erule power-le-dvd*)
apply *simp*
apply (*erule power-le-dvd* [*OF order-1*])
done

lemma *poly-squarefree-decomp-order*:
  assumes *pderiv p* ≠ *0*
  and *p*: *p* = *q* ∗ *d*
  and *p'*: *pderiv p* = *e* ∗ *d*
  and *d*: *d* = *r* ∗ *p* + *s* ∗ *pderiv p*

```
  shows order a q = (if order a p = 0 then 0 else 1)
proof (rule classical)
  assume 1: order a q ≠ (if order a p = 0 then 0 else 1)
  from ‹pderiv p ≠ 0› have p ≠ 0 by auto
  with p have order a p = order a q + order a d
    by (simp add: order-mult)
  with 1 have order a p ≠ 0 by (auto split: if-splits)
  have order a (pderiv p) = order a e + order a d
    using ‹pderiv p ≠ 0› ‹pderiv p = e * d› by (simp add: order-mult)
  have order a p = Suc (order a (pderiv p))
    using ‹pderiv p ≠ 0› ‹order a p ≠ 0› by (rule order-pderiv)
  have d ≠ 0 using ‹p ≠ 0› ‹p = q * d› by simp
  have ([:-a, 1:] ^ (order a (pderiv p))) dvd d
    apply (simp add: d)
    apply (rule dvd-add)
    apply (rule dvd-mult)
    apply (simp add: order-divides ‹p ≠ 0›
          ‹order a p = Suc (order a (pderiv p))›)
    apply (rule dvd-mult)
    apply (simp add: order-divides)
    done
  then have order a (pderiv p) ≤ order a d
    using ‹d ≠ 0› by (simp add: order-divides)
  show ?thesis
    using ‹order a p = order a q + order a d›
    using ‹order a (pderiv p) = order a e + order a d›
    using ‹order a p = Suc (order a (pderiv p))›
    using ‹order a (pderiv p) ≤ order a d›
    by auto
qed

lemma poly-squarefree-decomp-order2: [| pderiv p ≠ 0;
        p = q * d;
        pderiv p = e * d;
        d = r * p + s * pderiv p
     |] ==> ∀ a. order a q = (if order a p = 0 then 0 else 1)
apply (blast intro: poly-squarefree-decomp-order)
done

lemma order-pderiv2: [| pderiv p ≠ 0; order a p ≠ 0 |]
     ==> (order a (pderiv p) = n) = (order a p = Suc n)
apply (auto dest: order-pderiv)
done

definition
  rsquarefree :: 'a::idom poly => bool where
  rsquarefree p = (p ≠ 0 & (∀ a. (order a p = 0) | (order a p = 1)))

lemma pderiv-iszero: pderiv p = 0 ⟹ ∃ h. p = [:h:]
```

apply (*simp add*: *pderiv-eq-0-iff*)
apply (*case-tac p*, *auto split*: *if-splits*)
done

lemma *rsquarefree-roots*:
  rsquarefree p = (∀ *a*. ~(*poly p a* = *0* & *poly* (*pderiv p*) *a* = *0*))
apply (*simp add*: *rsquarefree-def*)
apply (*case-tac p* = *0*, *simp*, *simp*)
apply (*case-tac pderiv p* = *0*)
apply *simp*
apply (*drule pderiv-iszero*, *clarify*)
apply *simp*
apply (*rule allI*)
apply (*cut-tac p* = [:*h*:] **and** *a* = *a* **in** *order-root*)
apply *simp*
apply (*auto simp add*: *order-root order-pderiv2*)
apply (*erule-tac x*=*a* **in** *allE*, *simp*)
done

lemma *poly-squarefree-decomp*:
  assumes *pderiv p* ≠ *0*
    and *p* = *q* ∗ *d*
    and *pderiv p* = *e* ∗ *d*
    and *d* = *r* ∗ *p* + *s* ∗ *pderiv p*
  shows *rsquarefree q* & (∀ *a*. (*poly q a* = *0*) = (*poly p a* = *0*))
proof −
  from ‹*pderiv p* ≠ *0*› **have** *p* ≠ *0* **by** *auto*
  with ‹*p* = *q* ∗ *d*› **have** *q* ≠ *0* **by** *simp*
  have ∀ *a*. *order a q* = (*if order a p* = *0 then 0 else 1*)
    using *assms* **by** (*rule poly-squarefree-decomp-order2*)
  with ‹*p* ≠ *0*› ‹*q* ≠ *0*› **show** *?thesis*
    by (*simp add*: *rsquarefree-def order-root*)
qed

end

55 Preorder: Preorders with explicit equivalence relation

theory *Preorder*
imports *Orderings*
begin

class *preorder-equiv* = *preorder*
begin

definition *equiv* :: *'a* ⇒ *'a* ⇒ *bool* **where**

equiv x y ⟷ $x \leq y \wedge y \leq x$

notation
equiv (*op* ~~) **and**
equiv ((-/ ~~ -) [*51*, *51*] *50*)

notation (*xsymbols*)
equiv (*op* ≈) **and**
equiv ((-/ ≈ -) [*51*, *51*] *50*)

notation (*HTML* **output**)
equiv (*op* ≈) **and**
equiv ((-/ ≈ -) [*51*, *51*] *50*)

lemma *refl* [*iff*]:
$x \approx x$
unfolding *equiv-def* **by** *simp*

lemma *trans*:
$x \approx y \Longrightarrow y \approx z \Longrightarrow x \approx z$
unfolding *equiv-def* **by** (*auto intro*: *order-trans*)

lemma *antisym*:
$x \leq y \Longrightarrow y \leq x \Longrightarrow x \approx y$
unfolding *equiv-def* **..**

lemma *less-le*: $x < y$ ⟷ $x \leq y \wedge \neg\, x \approx y$
by (*auto simp add*: *equiv-def less-le-not-le*)

lemma *le-less*: $x \leq y$ ⟷ $x < y \vee x \approx y$
by (*auto simp add*: *equiv-def less-le*)

lemma *le-imp-less-or-eq*: $x \leq y \Longrightarrow x < y \vee x \approx y$
by (*simp add*: *less-le*)

lemma *less-imp-not-eq*: $x < y \Longrightarrow x \approx y$ ⟷ *False*
by (*simp add*: *less-le*)

lemma *less-imp-not-eq2*: $x < y \Longrightarrow y \approx x$ ⟷ *False*
by (*simp add*: *equiv-def less-le*)

lemma *neq-le-trans*: $\neg\, a \approx b \Longrightarrow a \leq b \Longrightarrow a < b$
by (*simp add*: *less-le*)

lemma *le-neq-trans*: $a \leq b \Longrightarrow \neg\, a \approx b \Longrightarrow a < b$
by (*simp add*: *less-le*)

lemma *antisym-conv*: $y \leq x \Longrightarrow x \leq y$ ⟷ $x \approx y$
by (*simp add*: *equiv-def*)

end

end

56 Quicksort: Quicksort

theory *Quicksort*
imports *Main Multiset*
begin

context *linorder*
begin

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

lemma [*code*]:
  quicksort [] = []
  quicksort (*x*#*xs*) = *quicksort* [*y*←*xs*. *y*<*x*] @ [*x*] @ *quicksort* [*y*←*xs*. *x*≤*y*]
  by (*simp-all add*: *not-le*)

lemma *quicksort-permutes* [*simp*]:
  multiset-of (*quicksort xs*) = *multiset-of xs*
  by (*induct xs rule*: *quicksort.induct*) (*simp-all add*: *ac-simps*)

lemma *set-quicksort* [*simp*]: *set* (*quicksort xs*) = *set xs*
  by (*simp add*: *set-count-greater-0*)

lemma *sorted-quicksort*: *sorted* (*quicksort xs*)
  by (*induct xs rule*: *quicksort.induct*) (*auto simp add*: *sorted-Cons sorted-append not-le less-imp-le*)

theorem *quicksort-sort* [*code-unfold*]:
  sort = *quicksort*
  by (*rule ext*, *rule properties-for-sort*) (*fact quicksort-permutes sorted-quicksort*)+

end

end

57 Quotient-Syntax: Pretty syntax for Quotient operations

theory *Quotient-Syntax*

imports *Main*
begin

notation
rel-conj (**infixr** *OOO* *75*) **and**
fun-map (**infixr** *−−−>* *55*) **and**
fun-rel (**infixr** *===>* *55*)

end

58 Quotient-List: Quotient infrastructure for the list type

theory *Quotient-List*
imports *Main Quotient-Syntax*
begin

fun
list-rel
where
list-rel R [] [] = *True*
| *list-rel R* (*x*#*xs*) [] = *False*
| *list-rel R* [] (*x*#*xs*) = *False*
| *list-rel R* (*x*#*xs*) (*y*#*ys*) = (*R x y* ∧ *list-rel R xs ys*)

declare [[*map list* = (*map*, *list-rel*)]]

lemma *split-list-all*:
shows (∀*x*. *P x*) ⟷ *P* [] ∧ (∀*x xs*. *P* (*x*#*xs*))
apply(*auto*)
apply(*case-tac x*)
apply(*simp-all*)
done

lemma *map-id*[*id-simps*]:
shows *map id* = *id*
apply(*simp add*: *expand-fun-eq*)
apply(*rule allI*)
apply(*induct-tac x*)
apply(*simp-all*)
done

lemma *list-rel-reflp*:
shows *equivp R* ⟹ *list-rel R xs xs*
apply(*induct xs*)
apply(*simp-all add*: *equivp-reflp*)

done

lemma *list-rel-symp*:
assumes *a*: *equivp R*
shows *list-rel R xs ys* $\Longrightarrow$ *list-rel R ys xs*
apply(*induct xs ys rule*: *list-induct2'*)
apply(*simp-all*)
apply(*rule equivp-symp*[*OF a*])
apply(*simp*)
done

lemma *list-rel-transp*:
assumes *a*: *equivp R*
shows *list-rel R xs1 xs2* $\Longrightarrow$ *list-rel R xs2 xs3* $\Longrightarrow$ *list-rel R xs1 xs3*
using *a*
apply(*induct R xs1 xs2 arbitrary*: *xs3 rule*: *list-rel.induct*)
apply(*simp*)
apply(*simp*)
apply(*simp*)
apply(*case-tac xs3*)
apply(*clarify*)
apply(*simp* (*no-asm-use*))
apply(*clarify*)
apply(*simp* (*no-asm-use*))
apply(*auto intro*: *equivp-transp*)
done

lemma *list-equivp*[*quot-equiv*]:
assumes *a*: *equivp R*
shows *equivp* (*list-rel R*)
apply(*rule equivpI*)
unfolding *reflp-def symp-def transp-def*
apply(*subst split-list-all*)
apply(*simp add*: *equivp-reflp*[*OF a*] *list-rel-reflp*[*OF a*])
apply(*blast intro*: *list-rel-symp*[*OF a*])
apply(*blast intro*: *list-rel-transp*[*OF a*])
done

lemma *list-rel-rel*:
assumes *q*: *Quotient R Abs Rep*
shows *list-rel R r s* = (*list-rel R r r* $\wedge$ *list-rel R s s* $\wedge$ (*map Abs r* = *map Abs s*))
apply(*induct r s rule*: *list-induct2'*)
apply(*simp-all*)
using *Quotient-rel*[*OF q*]
apply(*metis*)
done

lemma *list-quotient*[*quot-thm*]:

assumes *q*: *Quotient R Abs Rep*
shows *Quotient* (*list-rel R*) (*map Abs*) (*map Rep*)
unfolding *Quotient-def*
apply(*subst split-list-all*)
apply(*simp add*: *Quotient-abs-rep*[*OF q*] *abs-o-rep*[*OF q*] *map-id*)
apply(*rule conjI*)
apply(*rule allI*)
apply(*induct-tac a*)
apply(*simp*)
apply(*simp*)
apply(*simp add*: *Quotient-rep-reflp*[*OF q*])
apply(*rule allI*)+
apply(*rule list-rel-rel*[*OF q*])
done

lemma *cons-prs-aux*:
assumes *q*: *Quotient R Abs Rep*
shows (*map Abs*) ((*Rep h*) # (*map Rep t*)) = *h* # *t*
by (*induct t*) (*simp-all add*: *Quotient-abs-rep*[*OF q*])

lemma *cons-prs*[*quot-preserve*]:
assumes *q*: *Quotient R Abs Rep*
shows (*Rep* −−−> (*map Rep*) −−−> (*map Abs*)) (*op* #) = (*op* #)
by (*simp only*: *expand-fun-eq fun-map-def cons-prs-aux*[*OF q*])
(*simp*)

lemma *cons-rsp*[*quot-respect*]:
assumes *q*: *Quotient R Abs Rep*
shows (*R* ===> *list-rel R* ===> *list-rel R*) (*op* #) (*op* #)
by (*auto*)

lemma *nil-prs*[*quot-preserve*]:
assumes *q*: *Quotient R Abs Rep*
shows *map Abs* [] = []
by *simp*

lemma *nil-rsp*[*quot-respect*]:
assumes *q*: *Quotient R Abs Rep*
shows *list-rel R* [] []
by *simp*

lemma *map-prs-aux*:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows (*map abs2*) (*map* ((*abs1* −−−> *rep2*) *f*) (*map rep1 l*)) = *map f l*
by (*induct l*)
(*simp-all add*: *Quotient-abs-rep*[*OF a*] *Quotient-abs-rep*[*OF b*])

lemma *map-prs*[*quot-preserve*]:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows ((*abs1* −−−> *rep2*) −−−> (*map rep1*) −−−> (*map abs2*)) *map* = *map*
and ((*abs1* −−−> *id*) −−−> *map rep1* −−−> *id*) *map* = *map*
by (*simp-all only*: *expand-fun-eq fun-map-def map-prs-aux*[*OF a b*])
(*simp-all add*: *Quotient-abs-rep*[*OF a*])

lemma *map-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
shows ((*R1* ===> *R2*) ===> (*list-rel R1*) ===> *list-rel R2*) *map map*
and ((*R1* ===> *op* =) ===> (*list-rel R1*) ===> *op* =) *map map*
apply *simp-all*
apply(*rule-tac* [!] *allI*)+
apply(*rule-tac* [!] *impI*)
apply(*rule-tac* [!] *allI*)+
apply (*induct-tac* [!] *xa ya rule*: *list-induct2'*)
apply *simp-all*
done

lemma *foldr-prs-aux*:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows *abs2* (*foldr* ((*abs1* −−−> *abs2* −−−> *rep2*) *f*) (*map rep1 l*) (*rep2 e*))
= *foldr f l e*
by (*induct l*) (*simp-all add*: *Quotient-abs-rep*[*OF a*] *Quotient-abs-rep*[*OF b*])

lemma *foldr-prs*[*quot-preserve*]:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows ((*abs1* −−−> *abs2* −−−> *rep2*) −−−> (*map rep1*) −−−> *rep2* −−−>
abs2) *foldr* = *foldr*
by (*simp only*: *expand-fun-eq fun-map-def foldr-prs-aux*[*OF a b*])
(*simp*)

lemma *foldl-prs-aux*:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows *abs1* (*foldl* ((*abs1* −−−> *abs2* −−−> *rep1*) *f*) (*rep1 e*) (*map rep2 l*))
= *foldl f e l*
by (*induct l arbitrary*:*e*) (*simp-all add*: *Quotient-abs-rep*[*OF a*] *Quotient-abs-rep*[*OF b*])

lemma *foldl-prs*[*quot-preserve*]:
assumes *a*: *Quotient R1 abs1 rep1*
and *b*: *Quotient R2 abs2 rep2*
shows ((*abs1* −−−> *abs2* −−−> *rep1*) −−−> *rep1* −−−> (*map rep2*) −−−>

abs1) *foldl* = *foldl*
by (*simp only*: *expand-fun-eq fun-map-def foldl-prs-aux*[*OF a b*])
(*simp*)

lemma *list-rel-empty*:
shows *list-rel R* [] *b* ⟹ *length b* = *0*
by (*induct b*) (*simp-all*)

lemma *list-rel-len*:
shows *list-rel R a b* ⟹ *length a* = *length b*
apply (*induct a arbitrary*: *b*)
apply (*simp add*: *list-rel-empty*)
apply (*case-tac b*)
apply *simp-all*
done

lemma *foldl-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
shows ((*R1* ===> *R2* ===> *R1*) ===> *R1* ===> *list-rel R2* ===> *R1*) *foldl foldl*
apply(*auto*)
apply (*subgoal-tac R1 xa ya* ⟶ *list-rel R2 xb yb* ⟶ *R1* (*foldl x xa xb*) (*foldl y ya yb*))
apply *simp*
apply (*rule-tac x*=*xa* **in** *spec*)
apply (*rule-tac x*=*ya* **in** *spec*)
apply (*rule-tac xs*=*xb* **and** *ys*=*yb* **in** *list-induct2*)
apply (*rule list-rel-len*)
apply (*simp-all*)
done

lemma *foldr-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
shows ((*R1* ===> *R2* ===> *R2*) ===> *list-rel R1* ===> *R2* ===> *R2*) *foldr foldr*
apply *auto*
apply(*subgoal-tac R2 xb yb* ⟶ *list-rel R1 xa ya* ⟶ *R2* (*foldr x xa xb*) (*foldr y ya yb*))
apply *simp*
apply (*rule-tac xs*=*xa* **and** *ys*=*ya* **in** *list-induct2*)
apply (*rule list-rel-len*)
apply (*simp-all*)
done

lemma *list-rel-rsp*:
assumes *r*: ∀ *x y*. *R x y* ⟶ (∀ *a b*. *R a b* ⟶ *S x a* = *T y b*)

```
  and l1: list-rel R x y
  and l2: list-rel R a b
  shows list-rel S x a = list-rel T y b
  proof −
    have a: length y = length x by (rule list-rel-len[OF l1, symmetric])
    have c: length a = length b by (rule list-rel-len[OF l2])
    show ?thesis proof (cases length x = length a)
      case True
      have b: length x = length a by fact
      show ?thesis using a b c r l1 l2 proof (induct rule: list-induct4)
        case Nil
        show ?case using assms by simp
      next
        case (Cons h t)
        then show ?case by auto
      qed
    next
      case False
      have d: length x ≠ length a by fact
      then have e: ¬list-rel S x a using list-rel-len by auto
      have length y ≠ length b using d a c by simp
      then have ¬list-rel T y b using list-rel-len by auto
      then show ?thesis using e by simp
    qed
  qed

lemma[quot-respect]:
  ((R ===> R ===> op =) ===> list-rel R ===> list-rel R ===> op =)
list-rel list-rel
  by (simp add: list-rel-rsp)

lemma[quot-preserve]:
  assumes a: Quotient R abs1 rep1
 shows ((abs1 −−−> abs1 −−−> id) −−−> map rep1 −−−> map rep1 −−−>
id) list-rel = list-rel
  apply (simp add: expand-fun-eq)
  apply clarify
  apply (induct-tac xa xb rule: list-induct2')
  apply (simp-all add: Quotient-abs-rep[OF a])
  done

lemma[quot-preserve]:
  assumes a: Quotient R abs1 rep1
  shows (list-rel ((rep1 −−−> rep1 −−−> id) R) l m) = (l = m)
  by (induct l m rule: list-induct2') (simp-all add: Quotient-rel-rep[OF a])

lemma list-rel-eq[id-simps]:
  shows (list-rel (op =)) = (op =)
  unfolding expand-fun-eq
```

apply(*rule allI*)+
apply(*induct-tac x xa rule*: *list-induct2'*)
apply(*simp-all*)
done

lemma *list-rel-find-element*:
assumes *a*: $x \in set\ a$
and *b*: *list-rel R a b*
shows $\exists y.\ (y \in set\ b \wedge R\ x\ y)$
proof −
have *length a* = *length b* **using** *b* **by** (*rule list-rel-len*)
then show *?thesis* **using** *a b* **by** (*induct a b rule*: *list-induct2*) *auto*
qed

lemma *list-rel-refl*:
assumes *a*: $\bigwedge x\ y.\ R\ x\ y = (R\ x = R\ y)$
shows *list-rel R x x*
by (*induct x*) (*auto simp add*: *a*)

end

59 Quotient-Option: Quotient infrastructure for the option type

theory *Quotient-Option*
imports *Main Quotient-Syntax*
begin

fun
option-rel
where
option-rel R None None = *True*
| *option-rel R* (*Some x*) *None* = *False*
| *option-rel R None* (*Some x*) = *False*
| *option-rel R* (*Some x*) (*Some y*) = *R x y*

declare [[*map option* = (*Option.map*, *option-rel*)]]

should probably be in Option.thy

lemma *split-option-all*:
shows $(\forall x.\ P\ x) \longleftrightarrow P\ None \wedge (\forall a.\ P\ (Some\ a))$
apply(*auto*)
apply(*case-tac x*)
apply(*simp-all*)
done

lemma *option-quotient*[*quot-thm*]:

assumes *q*: *Quotient R Abs Rep*
shows *Quotient* (*option-rel R*) (*Option.map Abs*) (*Option.map Rep*)
unfolding *Quotient-def*
apply(*simp add*: *split-option-all*)
apply(*simp add*: *Quotient-abs-rep*[*OF q*] *Quotient-rel-rep*[*OF q*])
using *q*
unfolding *Quotient-def*
apply(*blast*)
done

lemma *option-equivp*[*quot-equiv*]:
assumes *a*: *equivp R*
shows *equivp* (*option-rel R*)
apply(*rule equivpI*)
unfolding *reflp-def symp-def transp-def*
apply(*simp-all add*: *split-option-all*)
apply(*blast intro*: *equivp-reflp*[*OF a*])
apply(*blast intro*: *equivp-symp*[*OF a*])
apply(*blast intro*: *equivp-transp*[*OF a*])
done

lemma *option-None-rsp*[*quot-respect*]:
assumes *q*: *Quotient R Abs Rep*
shows *option-rel R None None*
by *simp*

lemma *option-Some-rsp*[*quot-respect*]:
assumes *q*: *Quotient R Abs Rep*
shows (*R* ===> *option-rel R*) *Some Some*
by *simp*

lemma *option-None-prs*[*quot-preserve*]:
assumes *q*: *Quotient R Abs Rep*
shows *Option.map Abs None* = *None*
by *simp*

lemma *option-Some-prs*[*quot-preserve*]:
assumes *q*: *Quotient R Abs Rep*
shows (*Rep* ---> *Option.map Abs*) *Some* = *Some*
apply(*simp add*: *expand-fun-eq*)
apply(*simp add*: *Quotient-abs-rep*[*OF q*])
done

lemma *option-map-id*[*id-simps*]:
shows *Option.map id* = *id*
by (*simp add*: *expand-fun-eq split-option-all*)

lemma *option-rel-eq*[*id-simps*]:
shows *option-rel* (*op* =) = (*op* =)

by (*simp add*: *expand-fun-eq split-option-all*)

end

60 Quotient-Product: Quotient infrastructure for the product type

theory *Quotient-Product*
imports *Main Quotient-Syntax*
begin

fun
prod-rel
where
prod-rel R1 R2 = (λ(*a*, *b*) (*c*, *d*). *R1 a c* ∧ *R2 b d*)

declare [[*map* ∗ = (*prod-fun*, *prod-rel*)]]

lemma *prod-equivp*[*quot-equiv*]:
assumes *a*: *equivp R1*
assumes *b*: *equivp R2*
shows *equivp* (*prod-rel R1 R2*)
apply(*rule equivpI*)
unfolding *reflp-def symp-def transp-def*
apply(*simp-all add*: *split-paired-all*)
apply(*blast intro*: *equivp-reflp*[*OF a*] *equivp-reflp*[*OF b*])
apply(*blast intro*: *equivp-symp*[*OF a*] *equivp-symp*[*OF b*])
apply(*blast intro*: *equivp-transp*[*OF a*] *equivp-transp*[*OF b*])
done

lemma *prod-quotient*[*quot-thm*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows *Quotient* (*prod-rel R1 R2*) (*prod-fun Abs1 Abs2*) (*prod-fun Rep1 Rep2*)
unfolding *Quotient-def*
apply(*simp add*: *split-paired-all*)
apply(*simp add*: *Quotient-abs-rep*[*OF q1*] *Quotient-rel-rep*[*OF q1*])
apply(*simp add*: *Quotient-abs-rep*[*OF q2*] *Quotient-rel-rep*[*OF q2*])
using *q1 q2*
unfolding *Quotient-def*
apply(*blast*)
done

lemma *Pair-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*

shows (*R1 ===> R2 ===> prod-rel R1 R2*) *Pair Pair*
by *simp*

lemma *Pair-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*Rep1 ---> Rep2 ---> (prod-fun Abs1 Abs2)*) *Pair = Pair*
apply(*simp add*: *expand-fun-eq*)
apply(*simp add*: *Quotient-abs-rep*[*OF q1*] *Quotient-abs-rep*[*OF q2*])
done

lemma *fst-rsp*[*quot-respect*]:
assumes *Quotient R1 Abs1 Rep1*
assumes *Quotient R2 Abs2 Rep2*
shows (*prod-rel R1 R2 ===> R1*) *fst fst*
by *simp*

lemma *fst-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*prod-fun Rep1 Rep2 ---> Abs1*) *fst = fst*
apply(*simp add*: *expand-fun-eq*)
apply(*simp add*: *Quotient-abs-rep*[*OF q1*])
done

lemma *snd-rsp*[*quot-respect*]:
assumes *Quotient R1 Abs1 Rep1*
assumes *Quotient R2 Abs2 Rep2*
shows (*prod-rel R1 R2 ===> R2*) *snd snd*
by *simp*

lemma *snd-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*prod-fun Rep1 Rep2 ---> Abs2*) *snd = snd*
apply(*simp add*: *expand-fun-eq*)
apply(*simp add*: *Quotient-abs-rep*[*OF q2*])
done

lemma *split-rsp*[*quot-respect*]:
shows ((*R1 ===> R2 ===> (op =)*) *===> (prod-rel R1 R2) ===> (op =)*) *split split*
by *auto*

lemma *split-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
and *q2*: *Quotient R2 Abs2 Rep2*
shows (((*Abs1 ---> Abs2 ---> id*) *---> prod-fun Rep1 Rep2 ---> id*) *split*) = *split*

by (*simp add*: *expand-fun-eq Quotient-abs-rep*[*OF q1*] *Quotient-abs-rep*[*OF q2*])

lemma [*quot-respect*]:
shows ((*R2* ===> *R2* ===> *op* =) ===> (*R1* ===> *R1* ===> *op* =) ===>
prod-rel R2 R1 ===> *prod-rel R2 R1* ===> *op* =) *prod-rel prod-rel*
by *auto*

lemma [*quot-preserve*]:
assumes *q1*: *Quotient R1 abs1 rep1*
and *q2*: *Quotient R2 abs2 rep2*
shows ((*abs1* ---> *abs1* ---> *id*) ---> (*abs2* ---> *abs2* ---> *id*) --->
prod-fun rep1 rep2 ---> *prod-fun rep1 rep2* ---> *id*) *prod-rel* = *prod-rel*
by (*simp add*: *expand-fun-eq Quotient-abs-rep*[*OF q1*] *Quotient-abs-rep*[*OF q2*])

lemma [*quot-preserve*]:
shows(*prod-rel* ((*rep1* ---> *rep1* ---> *id*) *R1*) ((*rep2* ---> *rep2* ---> *id*) *R2*)
(*l1*, *l2*) (*r1*, *r2*)) = (*R1* (*rep1 l1*) (*rep1 r1*) ∧ *R2* (*rep2 l2*) (*rep2 r2*))
by *simp*

declare *Pair-eq*[*quot-preserve*]

lemma *prod-fun-id*[*id-simps*]:
shows *prod-fun id id* = *id*
by (*simp add*: *prod-fun-def*)

lemma *prod-rel-eq*[*id-simps*]:
shows *prod-rel* (*op* =) (*op* =) = (*op* =)
by (*simp add*: *expand-fun-eq*)

end

61 Quotient-Sum: Quotient infrastructure for the sum type

theory *Quotient-Sum*
imports *Main Quotient-Syntax*
begin

fun
sum-rel
where
sum-rel R1 R2 (*Inl a1*) (*Inl b1*) = *R1 a1 b1*
| *sum-rel R1 R2* (*Inl a1*) (*Inr b2*) = *False*
| *sum-rel R1 R2* (*Inr a2*) (*Inl b1*) = *False*

| *sum-rel R1 R2 (Inr a2) (Inr b2) = R2 a2 b2*

fun
sum-map
where
sum-map f1 f2 (Inl a) = Inl (f1 a)
| *sum-map f1 f2 (Inr a) = Inr (f2 a)*

declare [[*map + = (sum-map, sum-rel)*]]

should probably be in *Sum-Type*

lemma *split-sum-all*:
shows $(\forall x.\ P\ x) \longleftrightarrow (\forall x.\ P\ (Inl\ x)) \wedge (\forall x.\ P\ (Inr\ x))$
apply(*auto*)
apply(*case-tac x*)
apply(*simp-all*)
done

lemma *sum-equivp*[*quot-equiv*]:
assumes *a*: *equivp R1*
assumes *b*: *equivp R2*
shows *equivp* (*sum-rel R1 R2*)
apply(*rule equivpI*)
unfolding *reflp-def symp-def transp-def*
apply(*simp-all add*: *split-sum-all*)
apply(*blast intro*: *equivp-reflp*[*OF a*] *equivp-reflp*[*OF b*])
apply(*blast intro*: *equivp-symp*[*OF a*] *equivp-symp*[*OF b*])
apply(*blast intro*: *equivp-transp*[*OF a*] *equivp-transp*[*OF b*])
done

lemma *sum-quotient*[*quot-thm*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows *Quotient* (*sum-rel R1 R2*) (*sum-map Abs1 Abs2*) (*sum-map Rep1 Rep2*)
unfolding *Quotient-def*
apply(*simp add*: *split-sum-all*)
apply(*simp-all add*: *Quotient-abs-rep*[*OF q1*] *Quotient-rel-rep*[*OF q1*])
apply(*simp-all add*: *Quotient-abs-rep*[*OF q2*] *Quotient-rel-rep*[*OF q2*])
using *q1 q2*
unfolding *Quotient-def*
apply(*blast*)+
done

lemma *sum-Inl-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*R1 ===> sum-rel R1 R2*) *Inl Inl*
by *simp*

lemma *sum-Inr-rsp*[*quot-respect*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*R2 ===> sum-rel R1 R2*) *Inr Inr*
by *simp*

lemma *sum-Inl-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*Rep1 ---> sum-map Abs1 Abs2*) *Inl* = *Inl*
apply(*simp add*: *expand-fun-eq*)
apply(*simp add*: *Quotient-abs-rep*[*OF q1*])
done

lemma *sum-Inr-prs*[*quot-preserve*]:
assumes *q1*: *Quotient R1 Abs1 Rep1*
assumes *q2*: *Quotient R2 Abs2 Rep2*
shows (*Rep2 ---> sum-map Abs1 Abs2*) *Inr* = *Inr*
apply(*simp add*: *expand-fun-eq*)
apply(*simp add*: *Quotient-abs-rep*[*OF q2*])
done

lemma *sum-map-id*[*id-simps*]:
shows *sum-map id id* = *id*
by (*simp add*: *expand-fun-eq split-sum-all*)

lemma *sum-rel-eq*[*id-simps*]:
shows *sum-rel* (*op* =) (*op* =) = (*op* =)
by (*simp add*: *expand-fun-eq split-sum-all*)

end

62 Quotient-Type: Quotient types

theory *Quotient-Type*
imports *Main*
begin

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

62.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim :: 'a => 'a => bool$.

class *eqv* =
fixes *eqv* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\sim$ *50*)

class *equiv* = *eqv* +
  assumes *equiv-refl* [*intro*]: $x \sim x$
  assumes *equiv-trans* [*trans*]: $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$
  assumes *equiv-sym* [*sym*]: $x \sim y \Longrightarrow y \sim x$

lemma *equiv-not-sym* [*sym*]: $\neg\ (x \sim y) \Longrightarrow \neg\ (y \sim (x::'a::equiv))$
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*]: $\neg\ (x \sim y) \Longrightarrow y \sim z \Longrightarrow \neg\ (x \sim (z::'a::equiv))$
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*]: $x \sim y \Longrightarrow \neg\ (y \sim z) \Longrightarrow \neg\ (x \sim (z::'a::equiv))$
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*]: $R \in quot \Longrightarrow (!!a.\ R = \{x.\ a \sim x\} \Longrightarrow C) \Longrightarrow C$
  unfolding *quot-def* **by** *blast*

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

definition
  class :: $'a::equiv \Rightarrow 'a\ quot$ ($\lfloor$-$\rfloor$) **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*]: $(!!a.\ A = \lfloor a \rfloor \Longrightarrow C) \Longrightarrow C$
using *quot-exhaust* **by** *blast*

62.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $(\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim b)$
proof
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 $a \in \{x.\ b \sim x\}$ **by** *blast*
then have $b \sim a$ **by** *blast*
then show *?thesis* **..**
qed
next
assume *ab*: $a \sim b$
show $\lfloor a \rfloor = \lfloor b \rfloor$
proof −
have $\{x.\ a \sim x\} = \{x.\ b \sim x\}$
proof (*rule Collect-cong*)
fix *x* **show** $(a \sim x) = (b \sim x)$
proof
from *ab* **have** $b \sim a$ **..**
also assume $a \sim x$
finally show $b \sim x$ **.**
next
note *ab*
also assume $b \sim x$
finally show $a \sim x$ **.**
qed
qed
then show *?thesis* **by** (*simp only*: *class-def*)
qed
qed

62.3 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'. ⌊x⌋ = ⌊x'⌋ ==> ⌊y⌋ = ⌊y'⌋
      ==> P ⌊x⌋ ⌊y⌋ ==> P ⌊x'⌋ ⌊y'⌋ ==> g x y = g x' y'
    and P: P ⌊a⌋ ⌊b⌋
  shows f ⌊a⌋ ⌊b⌋ = g a b
proof −
  from eq and P have f ⌊a⌋ ⌊b⌋ = g (pick ⌊a⌋) (pick ⌊b⌋) by (simp only:)
  also have ... = g a b
  proof (rule cong)
    show ⌊pick ⌊a⌋⌋ = ⌊a⌋ ..
    moreover
    show ⌊pick ⌊b⌋⌋ = ⌊b⌋ ..
    moreover
    show P ⌊a⌋ ⌊b⌋ by (rule P)
    ultimately show P ⌊pick ⌊a⌋⌋ ⌊pick ⌊b⌋⌋ 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'. ⌊x⌋ = ⌊x'⌋ ==> ⌊y⌋ = ⌊y'⌋ ==> g x y = g x' y'
  shows f ⌊a⌋ ⌊b⌋ = 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

63 Ramsey: Ramsey's Theorem

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

63.1 Preliminaries

63.1.1 "Axiom" of Dependent Choice

primrec *choice* :: ('*a* => *bool*) => ('*a* * '*a*) *set* => *nat* => '*a* **where**
— An integer-indexed chain of choices
choice-0: *choice P r 0* = (*SOME x. P x*)
| *choice-Suc*: *choice P r* (*Suc n*) = (*SOME y. P y* & (*choice P r n*, *y*) ∈ *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*) $\in$ *r* **using** *PSuc*
by (*auto intro*: *less-Suc-induct* [*OF less*] *transD* [*OF trans*])
qed

63.1.2 Partitions of a Set

definition
part :: *nat* => *nat* => *'a set* => (*'a set* => *nat*) => *bool*
— the function *f* partitions the *r*-subsets of the typically infinite set *Y* into *s* distinct categories.
where
part r s Y f = (∀ *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* ⊆ *YY* ==> *part r s Y f*
unfolding *part-def* **by** *blast*

63.2 Ramsey's Theorem: Infinitary Version

lemma *Ramsey-induction*:
fixes *s* **and** *r*::*nat*
shows
!!(*YY*::*'a set*) (*f*::*'a set* => *nat*).
[|*infinite YY*; *part r s YY f*|]
==> ∃ *Y' t'*. *Y'* ⊆ *YY* & *infinite Y'* & *t'* < *s* &
(∀ *X*. *X* ⊆ *Y'* & *finite X* & *card X* = *r* --> *f X* = *t'*)
proof (*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* ∈ *YY* **by** *blast*
let *?ramr* = {((*y*,*Y*,*t*),(*y'*,*Y'*,*t'*)). *y'* ∈ *Y* & *Y'* ⊆ *Y*}
let *?propr* = %(*y*,*Y*,*t*).
y ∈ *YY* & *y* ∉ *Y* & *Y* ⊆ *YY* & *infinite Y* & *t* < *s*
& (∀ *X*. *X*⊆*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 ∘ 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 ⊆ YY - {yy}  infinite Y0  t0 < s
        ∀X. X⊆Y0 ∧ finite X ∧ card X = r ⟶ (f ∘ insert yy) X = t0
      by blast
  with yy have propr0: ?propr(yy,Y0,t0) by blast
  have proprstep: ⋀x. ?propr x ⟹ ∃y. ?propr y ∧ (x, y) ∈ ?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' ∈ 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 ∘ insert yx')
        by (simp add: o-def part-Suc-imp-part part-subset [where YY=YY and
Y=Yx])
      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 = fst o g
  let ?gt = snd o snd o g
  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: ∀x∈Z. ∀y∈Z. x≠y --> f{x,y} < s
 shows
  ∃Y t. Y ⊆ Z & infinite Y & t < s & (∀x∈Y. ∀y∈Y. x≠y --> f{x,y} = t)
proof -
 have part2: ∀X. X ⊆ 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 ⊆ Z infinite Y t < s
       (∀X. X ⊆ Y & finite X & card X = 2 --> f X = t)
  by (insert Ramsey [OF infZ part2]) auto
 moreover from this have ∀x∈Y. ∀y∈Y. x ≠ y ⟶ f {x, y} = t by auto
 ultimately show ?thesis by iprover
qed
```

63.3 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [3].

```
definition
 disj-wf        :: ('a * 'a)set => bool
where
 disj-wf r = (∃T. ∃n::nat. (∀i<n. wf(T i)) & r = (⋃i<n. T i))

definition
```

transition-idx :: [*nat* => *'a*, *nat* => (*'a*∗*'a*)*set*, *nat set*] => *nat*
where
transition-idx s T A =
(*LEAST k*. ∃ *i j*. *A* = {*i*,*j*} & *i*<*j* & (*s j*, *s i*) ∈ *T k*)

lemma *transition-idx-less*:
[|*i*<*j*; (*s j*, *s i*) ∈ *T k*; *k*<*n*|] ==> *transition-idx s T* {*i*,*j*} < *n*
apply (*subgoal-tac transition-idx s T* {*i*, *j*} ≤ *k*, *simp*)
apply (*simp add*: *transition-idx-def*, *blast intro*: *Least-le*)
done

lemma *transition-idx-in*:
[|*i*<*j*; (*s j*, *s i*) ∈ *T k*|] ==> (*s j*, *s i*) ∈ *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
```

64 Reflection: Generic reflection and reification

```
theory Reflection
imports Main
uses reify-data.ML (reflection.ML)
```

begin

setup ⟪ *Reify-Data.setup* ⟫

lemma *ext2*: (∀ *x*. *f x* = *g x*) ⟹ *f* = *g*
by (*blast intro*: *ext*)

use *reflection.ML*

method-setup *reify* = ⟪
Attrib.thms −−
Scan.option (*Scan.lift* (*Args.*$$$ () |−− *Args.term* −−| *Scan.lift* (*Args.*$$$)))
>>
(*fn* (*eqs*, *to*) => *fn ctxt* => *SIMPLE-METHOD'* (*Reflection.genreify-tac ctxt* (*eqs* @ (*fst* (*Reify-Data.get ctxt*))) *to*))
⟫ *partial automatic reification*

method-setup *reflection* = ⟪
let
fun keyword k = *Scan.lift* (*Args.*$$$ *k* −− *Args.colon*) >> *K* ();
val onlyN = *only*;
val rulesN = *rules*;
val any-keyword = *keyword onlyN* || *keyword rulesN*;
val thms = *Scan.repeat* (*Scan.unless any-keyword Attrib.multi-thm*) >> *flat*;
val terms = *thms* >> *map* (*term-of o Drule.dest-term*);
in
thms −−
Scan.optional (*keyword rulesN* |−− *thms*) [] −−
Scan.option (*keyword onlyN* |−− *Args.term*) >>
(*fn* ((*eqs*,*ths*),*to*) => *fn ctxt* =>
let
val (*ceqs*,*cths*) = *Reify-Data.get ctxt*
val corr-thms = *ths*@*cths*
val raw-eqs = *eqs*@*ceqs*
in SIMPLE-METHOD' (*Reflection.reflection-tac ctxt corr-thms raw-eqs to*) *end*)
end
⟫ *reflection*

end

65 RBT-Impl: Implementation of Red-Black Trees

theory *RBT-Impl*
imports *Main*
begin

For applications, you should use theory *RBT* which defines an abstract type of red-black tree obeying the invariant.

65.1 Datatype of RB trees

datatype *color* = *R* | *B*
datatype ('*a*, '*b*) *rbt* = *Empty* | *Branch color* ('*a*, '*b*) *rbt* '*a* '*b* ('*a*, '*b*) *rbt*

lemma *rbt-cases*:
  obtains (*Empty*) *t* = *Empty*
  | (*Red*) *l k v r* **where** *t* = *Branch R l k v r*
  | (*Black*) *l k v r* **where** *t* = *Branch B l k v r*
proof (*cases t*)
  case *Empty* **with** *that* **show** *thesis* **by** *blast*
next
  case (*Branch c*) **with** *that* **show** *thesis* **by** (*cases c*) *blast+*
qed

65.2 Tree properties

65.2.1 Content of a tree

primrec *entries* :: ('*a*, '*b*) *rbt* ⇒ ('*a* × '*b*) *list*
where
  entries Empty = []
| *entries* (*Branch - l k v r*) = *entries l* @ (*k*,*v*) # *entries r*

abbreviation (*input*) *entry-in-tree* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *rbt* ⇒ *bool*
where
  entry-in-tree k v t ≡ (*k*, *v*) ∈ *set* (*entries t*)

definition *keys* :: ('*a*, '*b*) *rbt* ⇒ '*a list* **where**
  keys t = *map fst* (*entries t*)

lemma *keys-simps* [*simp*, *code*]:
  keys Empty = []
  keys (*Branch c l k v r*) = *keys l* @ *k* # *keys r*
  by (*simp-all add*: *keys-def*)

lemma *entry-in-tree-keys*:
  assumes (*k*, *v*) ∈ *set* (*entries t*)
  shows *k* ∈ *set* (*keys t*)
proof −
  from *assms* **have** *fst* (*k*, *v*) ∈ *fst* ' *set* (*entries t*) **by** (*rule imageI*)
  then show *?thesis* **by** (*simp add*: *keys-def*)
qed

lemma *keys-entries*:
  k ∈ *set* (*keys t*) ⟷ (∃ *v*. (*k*, *v*) ∈ *set* (*entries t*))
  by (*auto intro*: *entry-in-tree-keys*) (*auto simp add*: *keys-def*)

65.2.2 Search tree properties

definition *tree-less* :: *'a*::*order* ⇒ (*'a*, *'b*) *rbt* ⇒ *bool*
where
tree-less-prop: *tree-less k t* ⟷ (∀ *x*∈*set* (*keys t*). *x* < *k*)

abbreviation *tree-less-symbol* (**infix** |≪ *50*)
where *t* |≪ *x* ≡ *tree-less x t*

definition *tree-greater* :: *'a*::*order* ⇒ (*'a*, *'b*) *rbt* ⇒ *bool* (**infix** ≪| *50*)
where
tree-greater-prop: *tree-greater k t* = (∀ *x*∈*set* (*keys t*). *k* < *x*)

lemma *tree-less-simps* [*simp*]:
tree-less k Empty = *True*
tree-less k (*Branch c lt kt v rt*) ⟷ *kt* < *k* ∧ *tree-less k lt* ∧ *tree-less k rt*
by (*auto simp add*: *tree-less-prop*)

lemma *tree-greater-simps* [*simp*]:
tree-greater k Empty = *True*
tree-greater k (*Branch c lt kt v rt*) ⟷ *k* < *kt* ∧ *tree-greater k lt* ∧ *tree-greater k rt*
by (*auto simp add*: *tree-greater-prop*)

lemmas *tree-ord-props* = *tree-less-prop tree-greater-prop*

lemmas *tree-greater-nit* = *tree-greater-prop entry-in-tree-keys*
lemmas *tree-less-nit* = *tree-less-prop entry-in-tree-keys*

lemma *tree-less-eq-trans*: *l* |≪ *u* ⟹ *u* ≤ *v* ⟹ *l* |≪ *v*
and *tree-less-trans*: *t* |≪ *x* ⟹ *x* < *y* ⟹ *t* |≪ *y*
and *tree-greater-eq-trans*: *u* ≤ *v* ⟹ *v* ≪| *r* ⟹ *u* ≪| *r*
and *tree-greater-trans*: *x* < *y* ⟹ *y* ≪| *t* ⟹ *x* ≪| *t*
by (*auto simp*: *tree-ord-props*)

primrec *sorted* :: (*'a*::*linorder*, *'b*) *rbt* ⇒ *bool*
where
sorted Empty = *True*
| *sorted* (*Branch c l k v r*) = (*l* |≪ *k* ∧ *k* ≪| *r* ∧ *sorted l* ∧ *sorted r*)

lemma *sorted-entries*:
sorted t ⟹ *List.sorted* (*List.map fst* (*entries t*))
by (*induct t*)
(*force simp*: *sorted-append sorted-Cons tree-ord-props*
dest!: *entry-in-tree-keys*)+

lemma *distinct-entries*:
sorted t ⟹ *distinct* (*List.map fst* (*entries t*))
by (*induct t*)
(*force simp*: *sorted-append sorted-Cons tree-ord-props*

dest!: *entry-in-tree-keys*)+

65.2.3 Tree lookup

primrec *lookup* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ '*a* ⇀ '*b*
where
lookup Empty k = *None*
| *lookup* (*Branch* - *l x y r*) *k* = (*if k* < *x then lookup l k else if x* < *k then lookup r k else Some y*)

lemma *lookup-keys*: *sorted t* ⟹ *dom* (*lookup t*) = *set* (*keys t*)
by (*induct t*) (*auto simp*: *dom-def tree-greater-prop tree-less-prop*)

lemma *dom-lookup-Branch*:
sorted (*Branch c t1 k v t2*) ⟹
dom (*lookup* (*Branch c t1 k v t2*))
= *Set.insert k* (*dom* (*lookup t1*) ∪ *dom* (*lookup t2*))
proof −
assume *sorted* (*Branch c t1 k v t2*)
moreover from *this* **have** *sorted t1 sorted t2* **by** *simp-all*
ultimately show *?thesis* **by** (*simp add*: *lookup-keys*)
qed

lemma *finite-dom-lookup* [*simp*, *intro*!]: *finite* (*dom* (*lookup t*))
proof (*induct t*)
case *Empty* **then show** *?case* **by** *simp*
next
case (*Branch color t1 a b t2*)
let *?A* = *Set.insert a* (*dom* (*lookup t1*) ∪ *dom* (*lookup t2*))
have *dom* (*lookup* (*Branch color t1 a b t2*)) ⊆ *?A* **by** (*auto split*: *split-if-asm*)
moreover from *Branch* **have** *finite* (*insert a* (*dom* (*lookup t1*) ∪ *dom* (*lookup t2*))) **by** *simp*
ultimately show *?case* **by** (*rule finite-subset*)
qed

lemma *lookup-tree-less*[*simp*]: *t* |≪ *k* ⟹ *lookup t k* = *None*
by (*induct t*) *auto*

lemma *lookup-tree-greater*[*simp*]: *k* ≪| *t* ⟹ *lookup t k* = *None*
by (*induct t*) *auto*

lemma *lookup-Empty*: *lookup Empty* = *empty*
by (*rule ext*) *simp*

lemma *map-of-entries*:
sorted t ⟹ *map-of* (*entries t*) = *lookup t*
proof (*induct t*)
case *Empty* **thus** *?case* **by** (*simp add*: *lookup-Empty*)
next

```
  case (Branch c t1 k v t2)
  have lookup (Branch c t1 k v t2) = lookup t2 ++ [k↦v] ++ lookup t1
  proof (rule ext)
    fix x
    from Branch have SORTED: sorted (Branch c t1 k v t2) by simp
   let ?thesis = lookup (Branch c t1 k v t2) x = (lookup t2 ++ [k ↦ v] ++ lookup
t1) x

    have DOM-T1: !!k'. k'∈dom (lookup t1) ⟹ k>k'
    proof −
      fix k'
      from SORTED have t1 |≪ k by simp
      with tree-less-prop have ∀k'∈set (keys t1). k>k' by auto
      moreover assume k'∈dom (lookup t1)
      ultimately show k>k' using lookup-keys SORTED by auto
    qed

    have DOM-T2: !!k'. k'∈dom (lookup t2) ⟹ k<k'
    proof −
      fix k'
      from SORTED have k ≪| t2 by simp
      with tree-greater-prop have ∀k'∈set (keys t2). k<k' by auto
      moreover assume k'∈dom (lookup t2)
      ultimately show k<k' using lookup-keys SORTED by auto
    qed

    {
      assume C: x<k
      hence lookup (Branch c t1 k v t2) x = lookup t1 x by simp
      moreover from C have x∉dom [k↦v] by simp
      moreover have x∉dom (lookup t2) proof
        assume x∈dom (lookup t2)
        with DOM-T2 have k<x by blast
        with C show False by simp
      qed
    ultimately have ?thesis by (simp add: map-add-upd-left map-add-dom-app-simps)
    } moreover {
      assume [simp]: x=k
      hence lookup (Branch c t1 k v t2) x = [k ↦ v] x by simp
      moreover have x∉dom (lookup t1) proof
        assume x∈dom (lookup t1)
        with DOM-T1 have k>x by blast
        thus False by simp
      qed
    ultimately have ?thesis by (simp add: map-add-upd-left map-add-dom-app-simps)
    } moreover {
      assume C: x>k
    hence lookup (Branch c t1 k v t2) x = lookup t2 x by (simp add: less-not-sym[of
k x])
```

```
    moreover from C have x∉dom [k↦v] by simp
    moreover have x∉dom (lookup t1) proof
      assume x∈dom (lookup t1)
      with DOM-T1 have k>x by simp
      with C show False by simp
    qed
   ultimately have ?thesis by (simp add: map-add-upd-left map-add-dom-app-simps)
   } ultimately show ?thesis using less-linear by blast
 qed
 also from Branch have lookup t2 ++ [k ↦ v] ++ lookup t1 = map-of (entries
(Branch c t1 k v t2)) by simp
 finally show ?case by simp
qed

lemma lookup-in-tree: sorted t ⟹ lookup t k = Some v ⟷ (k, v) ∈ set (entries
t)
 by (simp add: map-of-entries [symmetric] distinct-entries)

lemma set-entries-inject:
 assumes sorted: sorted t1 sorted t2
 shows set (entries t1) = set (entries t2) ⟷ entries t1 = entries t2
proof −
 from sorted have distinct (map fst (entries t1))
   distinct (map fst (entries t2))
   by (auto intro: distinct-entries)
 with sorted show ?thesis
  by (auto intro: map-sorted-distinct-set-unique sorted-entries simp add: distinct-map)
qed

lemma entries-eqI:
 assumes sorted: sorted t1 sorted t2
 assumes lookup: lookup t1 = lookup t2
 shows entries t1 = entries t2
proof −
 from sorted lookup have map-of (entries t1) = map-of (entries t2)
   by (simp add: map-of-entries)
 with sorted have set (entries t1) = set (entries t2)
   by (simp add: map-of-inject-set distinct-entries)
 with sorted show ?thesis by (simp add: set-entries-inject)
qed

lemma entries-lookup:
 assumes sorted t1 sorted t2
 shows entries t1 = entries t2 ⟷ lookup t1 = lookup t2
 using assms by (auto intro: entries-eqI simp add: map-of-entries [symmetric])

lemma lookup-from-in-tree:
 assumes sorted t1 sorted t2
 and ⋀v. (k::'a::linorder, v) ∈ set (entries t1) ⟷ (k, v) ∈ set (entries t2)
```

shows *lookup t1 k = lookup t2 k*
proof −
from *assms* **have** *k ∈ dom (lookup t1) ⟷ k ∈ dom (lookup t2)*
by (*simp add*: *keys-entries lookup-keys*)
with *assms* **show** *?thesis* **by** (*auto simp add*: *lookup-in-tree* [*symmetric*])
qed

65.2.4 Red-black properties

primrec *color-of* :: (*'a*, *'b*) *rbt* ⇒ *color*
where
color-of Empty = B
| *color-of* (*Branch c - - - -*) = *c*

primrec *bheight* :: (*'a*,*'b*) *rbt* ⇒ *nat*
where
bheight Empty = 0
| *bheight* (*Branch c lt k v rt*) = (*if c = B then Suc* (*bheight lt*) *else bheight lt*)

primrec *inv1* :: (*'a*, *'b*) *rbt* ⇒ *bool*
where
inv1 Empty = True
| *inv1* (*Branch c lt k v rt*) ⟷ *inv1 lt* ∧ *inv1 rt* ∧ (*c = B* ∨ *color-of lt = B* ∧ *color-of rt = B*)

primrec *inv1l* :: (*'a*, *'b*) *rbt* ⇒ *bool* — Weaker version
where
inv1l Empty = True
| *inv1l* (*Branch c l k v r*) = (*inv1 l* ∧ *inv1 r*)
lemma [*simp*]: *inv1 t* ⟹ *inv1l t* **by** (*cases t*) *simp+*

primrec *inv2* :: (*'a*, *'b*) *rbt* ⇒ *bool*
where
inv2 Empty = True
| *inv2* (*Branch c lt k v rt*) = (*inv2 lt* ∧ *inv2 rt* ∧ *bheight lt = bheight rt*)

definition *is-rbt* :: (*'a*::*linorder*, *'b*) *rbt* ⇒ *bool* **where**
is-rbt t ⟷ inv1 t ∧ *inv2 t* ∧ *color-of t = B* ∧ *sorted t*

lemma *is-rbt-sorted* [*simp*]:
is-rbt t ⟹ *sorted t* **by** (*simp add*: *is-rbt-def*)

theorem *Empty-is-rbt* [*simp*]:
is-rbt Empty **by** (*simp add*: *is-rbt-def*)

65.3 Insertion

fun
balance :: (*'a*,*'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where

balance (*Branch R a w x b*) *s t* (*Branch R c y z d*) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance (*Branch R* (*Branch R a w x b*) *s t c*) *y z d* = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance (*Branch R a w x* (*Branch R b s t c*)) *y z d* = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a w x (*Branch R b s t* (*Branch R c y z d*)) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a w x (*Branch R* (*Branch R b s t c*) *y z d*) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a s t b = *Branch B a s t b*

lemma *balance-inv1*: ⟦*inv1l l*; *inv1l r*⟧ ⟹ *inv1* (*balance l k v r*)
by (*induct l k v r rule*: *balance.induct*) *auto*

lemma *balance-bheight*: *bheight l* = *bheight r* ⟹ *bheight* (*balance l k v r*) = *Suc* (*bheight l*)
by (*induct l k v r rule*: *balance.induct*) *auto*

lemma *balance-inv2*:
assumes *inv2 l inv2 r bheight l* = *bheight r*
shows *inv2* (*balance l k v r*)
using *assms*
by (*induct l k v r rule*: *balance.induct*) *auto*

lemma *balance-tree-greater*[*simp*]: (*v* ≪| *balance a k x b*) = (*v* ≪| *a* ∧ *v* ≪| *b* ∧ *v* < *k*)
by (*induct a k x b rule*: *balance.induct*) *auto*

lemma *balance-tree-less*[*simp*]: (*balance a k x b* |≪ *v*) = (*a* |≪ *v* ∧ *b* |≪ *v* ∧ *k* < *v*)
by (*induct a k x b rule*: *balance.induct*) *auto*

lemma *balance-sorted*:
fixes *k* :: *'a*::*linorder*
assumes *sorted l sorted r l* |≪ *k k* ≪| *r*
shows *sorted* (*balance l k v r*)
using *assms* **proof** (*induct l k v r rule*: *balance.induct*)
case (*2-2 a x w b y t c z s va vb vd vc*)
hence *y* < *z* ∧ *z* ≪| *Branch B va vb vd vc*
by (*auto simp add*: *tree-ord-props*)
hence *tree-greater y* (*Branch B va vb vd vc*) **by** (*blast dest*: *tree-greater-trans*)
with *2-2* **show** *?case* **by** *simp*
next
case (*3-2 va vb vd vc x w b y s c z*)
from *3-2* **have** *x* < *y* ∧ *tree-less x* (*Branch B va vb vd vc*)
by *simp*
hence *tree-less y* (*Branch B va vb vd vc*) **by** (*blast dest*: *tree-less-trans*)
with *3-2* **show** *?case* **by** *simp*

next
case (*3-3 x w b y s c z t va vb vd vc*)
from *3-3* **have** *y < z ∧ tree-greater z (Branch B va vb vd vc)* **by** *simp*
hence *tree-greater y (Branch B va vb vd vc)* **by** (*blast dest*: *tree-greater-trans*)
with *3-3* **show** *?case* **by** *simp*
next
case (*3-4 vd ve vg vf x w b y s c z t va vb vii vc*)
hence *x < y ∧ tree-less x (Branch B vd ve vg vf)* **by** *simp*
hence *1*: *tree-less y (Branch B vd ve vg vf)* **by** (*blast dest*: *tree-less-trans*)
from *3-4* **have** *y < z ∧ tree-greater z (Branch B va vb vii vc)* **by** *simp*
hence *tree-greater y (Branch B va vb vii vc)* **by** (*blast dest*: *tree-greater-trans*)
with *1 3-4* **show** *?case* **by** *simp*
next
case (*4-2 va vb vd vc x w b y s c z t dd*)
hence *x < y ∧ tree-less x (Branch B va vb vd vc)* **by** *simp*
hence *tree-less y (Branch B va vb vd vc)* **by** (*blast dest*: *tree-less-trans*)
with *4-2* **show** *?case* **by** *simp*
next
case (*5-2 x w b y s c z t va vb vd vc*)
hence *y < z ∧ tree-greater z (Branch B va vb vd vc)* **by** *simp*
hence *tree-greater y (Branch B va vb vd vc)* **by** (*blast dest*: *tree-greater-trans*)
with *5-2* **show** *?case* **by** *simp*
next
case (*5-3 va vb vd vc x w b y s c z t*)
hence *x < y ∧ tree-less x (Branch B va vb vd vc)* **by** *simp*
hence *tree-less y (Branch B va vb vd vc)* **by** (*blast dest*: *tree-less-trans*)
with *5-3* **show** *?case* **by** *simp*
next
case (*5-4 va vb vg vc x w b y s c z t vd ve vii vf*)
hence *x < y ∧ tree-less x (Branch B va vb vg vc)* **by** *simp*
hence *1*: *tree-less y (Branch B va vb vg vc)* **by** (*blast dest*: *tree-less-trans*)
from *5-4* **have** *y < z ∧ tree-greater z (Branch B vd ve vii vf)* **by** *simp*
hence *tree-greater y (Branch B vd ve vii vf)* **by** (*blast dest*: *tree-greater-trans*)
with *1 5-4* **show** *?case* **by** *simp*
qed *simp+*

lemma *entries-balance* [*simp*]:
entries (balance l k v r) = entries l @ (k, v) # entries r
by (*induct l k v r rule*: *balance.induct*) *auto*

lemma *keys-balance* [*simp*]:
keys (balance l k v r) = keys l @ k # keys r
by (*simp add*: *keys-def*)

lemma *balance-in-tree*:
entry-in-tree k x (balance l v y r) ⟷ entry-in-tree k x l ∨ k = v ∧ x = y ∨ entry-in-tree k x r
by (*auto simp add*: *keys-def*)

lemma *lookup-balance*[*simp*]:
fixes *k* :: *'a*::*linorder*
assumes *sorted l sorted r l* |≪ *k k* ≪| *r*
shows *lookup* (*balance l k v r*) *x* = *lookup* (*Branch B l k v r*) *x*
by (*rule lookup-from-in-tree*) (*auto simp*:*assms balance-in-tree balance-sorted*)

primrec *paint* :: *color* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
paint c Empty = *Empty*
| *paint c* (*Branch - l k v r*) = *Branch c l k v r*

lemma *paint-inv1l*[*simp*]: *inv1l t* ⟹ *inv1l* (*paint c t*) **by** (*cases t*) *auto*
lemma *paint-inv1*[*simp*]: *inv1l t* ⟹ *inv1* (*paint B t*) **by** (*cases t*) *auto*
lemma *paint-inv2*[*simp*]: *inv2 t* ⟹ *inv2* (*paint c t*) **by** (*cases t*) *auto*
lemma *paint-color-of*[*simp*]: *color-of* (*paint B t*) = *B* **by** (*cases t*) *auto*
lemma *paint-sorted*[*simp*]: *sorted t* ⟹ *sorted* (*paint c t*) **by** (*cases t*) *auto*
lemma *paint-in-tree*[*simp*]: *entry-in-tree k x* (*paint c t*) = *entry-in-tree k x t* **by** (*cases t*) *auto*
lemma *paint-lookup*[*simp*]: *lookup* (*paint c t*) = *lookup t* **by** (*rule ext*) (*cases t*, *auto*)
lemma *paint-tree-greater*[*simp*]: (*v* ≪| *paint c t*) = (*v* ≪| *t*) **by** (*cases t*) *auto*
lemma *paint-tree-less*[*simp*]: (*paint c t* |≪ *v*) = (*t* |≪ *v*) **by** (*cases t*) *auto*

fun
ins :: (*'a*::*linorder* ⇒ *'b* ⇒ *'b* ⇒ *'b*) ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
ins f k v Empty = *Branch R Empty k v Empty* |
ins f k v (*Branch B l x y r*) = (*if k* < *x then balance* (*ins f k v l*) *x y r*
else if k > *x then balance l x y* (*ins f k v r*)
else Branch B l x (*f k y v*) *r*) |
ins f k v (*Branch R l x y r*) = (*if k* < *x then Branch R* (*ins f k v l*) *x y r*
else if k > *x then Branch R l x y* (*ins f k v r*)
else Branch R l x (*f k y v*) *r*)

lemma *ins-inv1-inv2*:
assumes *inv1 t inv2 t*
shows *inv2* (*ins f k x t*) *bheight* (*ins f k x t*) = *bheight t*
color-of t = *B* ⟹ *inv1* (*ins f k x t*) *inv1l* (*ins f k x t*)
using *assms*
by (*induct f k x t rule*: *ins.induct*) (*auto simp*: *balance-inv1 balance-inv2 balance-bheight*)

lemma *ins-tree-greater*[*simp*]: (*v* ≪| *ins f k x t*) = (*v* ≪| *t* ∧ *k* > *v*)
by (*induct f k x t rule*: *ins.induct*) *auto*
lemma *ins-tree-less*[*simp*]: (*ins f k x t* |≪ *v*) = (*t* |≪ *v* ∧ *k* < *v*)
by (*induct f k x t rule*: *ins.induct*) *auto*
lemma *ins-sorted*[*simp*]: *sorted t* ⟹ *sorted* (*ins f k x t*)
by (*induct f k x t rule*: *ins.induct*) (*auto simp*: *balance-sorted*)

lemma *keys-ins*: *set* (*keys* (*ins f k v t*)) = { *k* } ∪ *set* (*keys t*)

by (*induct f k v t rule*: *ins.induct*) *auto*

lemma *lookup-ins*:
fixes *k* :: *'a*::*linorder*
assumes *sorted t*
shows *lookup* (*ins f k v t*) *x* = ((*lookup t*)(*k* |−> *case lookup t k of None* ⇒ *v*
| *Some w* ⇒ *f k w v*)) *x*
using *assms* **by** (*induct f k v t rule*: *ins.induct*) *auto*

definition
insert-with-key :: (*'a*::*linorder* ⇒ *'b* ⇒ *'b* ⇒ *'b*) ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
insert-with-key f k v t = *paint B* (*ins f k v t*)

lemma *insertwk-sorted*: *sorted t* ⟹ *sorted* (*insert-with-key f k x t*)
by (*auto simp*: *insert-with-key-def*)

theorem *insertwk-is-rbt*:
assumes *inv*: *is-rbt t*
shows *is-rbt* (*insert-with-key f k x t*)
using *assms*
unfolding *insert-with-key-def is-rbt-def*
by (*auto simp*: *ins-inv1-inv2*)

lemma *lookup-insertwk*:
assumes *sorted t*
shows *lookup* (*insert-with-key f k v t*) *x* = ((*lookup t*)(*k* |−> *case lookup t k of None* ⇒ *v*
| *Some w* ⇒ *f k w v*)) *x*
unfolding *insert-with-key-def* **using** *assms*
by (*simp add*:*lookup-ins*)

definition
insertw-def: *insert-with f* = *insert-with-key* (λ-. *f*)

lemma *insertw-sorted*: *sorted t* ⟹ *sorted* (*insert-with f k v t*) **by** (*simp add*: *insertwk-sorted insertw-def*)
theorem *insertw-is-rbt*: *is-rbt t* ⟹ *is-rbt* (*insert-with f k v t*) **by** (*simp add*: *insertwk-is-rbt insertw-def*)

lemma *lookup-insertw*:
assumes *is-rbt t*
shows *lookup* (*insert-with f k v t*) = (*lookup t*)(*k* ↦ (*if k*:*dom* (*lookup t*) *then f* (*the* (*lookup t k*)) *v else v*))
using *assms*
unfolding *insertw-def*
by (*rule-tac ext*) (*cases lookup t k*, *auto simp*:*lookup-insertwk dom-def*)

definition *insert* :: *'a::linorder* ⇒ *'b* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where**
insert = *insert-with-key* (λ- - *nv*. *nv*)

lemma *insert-sorted*: *sorted t* ⟹ *sorted* (*insert k v t*) **by** (*simp add*: *insertwk-sorted insert-def*)
theorem *insert-is-rbt* [*simp*]: *is-rbt t* ⟹ *is-rbt* (*insert k v t*) **by** (*simp add*: *insertwk-is-rbt insert-def*)

lemma *lookup-insert*:
assumes *is-rbt t*
shows *lookup* (*insert k v t*) = (*lookup t*)(*k*↦*v*)
unfolding *insert-def*
using *assms*
by (*rule-tac ext*) (*simp add*: *lookup-insertwk split*:*option.split*)

65.4 Deletion

lemma *bheight-paintR'*[*simp*]: *color-of t* = *B* ⟹ *bheight* (*paint R t*) = *bheight t* − *1*
by (*cases t rule*: *rbt-cases*) *auto*

fun
balance-left :: (*'a*,*'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
balance-left (*Branch R a k x b*) *s y c* = *Branch R* (*Branch B a k x b*) *s y c* |
balance-left bl k x (*Branch B a s y b*) = *balance bl k x* (*Branch R a s y b*) |
balance-left bl k x (*Branch R* (*Branch B a s y b*) *t z c*) = *Branch R* (*Branch B bl k x a*) *s y* (*balance b t z* (*paint R c*)) |
balance-left t k x s = *Empty*

lemma *balance-left-inv2-with-inv1*:
assumes *inv2 lt inv2 rt bheight lt* + *1* = *bheight rt inv1 rt*
shows *bheight* (*balance-left lt k v rt*) = *bheight lt* + *1*
and *inv2* (*balance-left lt k v rt*)
using *assms*
by (*induct lt k v rt rule*: *balance-left.induct*) (*auto simp*: *balance-inv2 balance-bheight*)

lemma *balance-left-inv2-app*:
assumes *inv2 lt inv2 rt bheight lt* + *1* = *bheight rt color-of rt* = *B*
shows *inv2* (*balance-left lt k v rt*)
bheight (*balance-left lt k v rt*) = *bheight rt*
using *assms*
by (*induct lt k v rt rule*: *balance-left.induct*) (*auto simp add*: *balance-inv2 balance-bheight*)+

lemma *balance-left-inv1*: ⟦*inv1l a*; *inv1 b*; *color-of b* = *B*⟧ ⟹ *inv1* (*balance-left a k x b*)
by (*induct a k x b rule*: *balance-left.induct*) (*simp add*: *balance-inv1*)+

lemma *balance-left-inv1l*: ⟦ *inv1l lt*; *inv1 rt* ⟧ ⟹ *inv1l* (*balance-left lt k x rt*)
by (*induct lt k x rt rule*: *balance-left.induct*) (*auto simp*: *balance-inv1*)

lemma *balance-left-sorted*: ⟦ *sorted l*; *sorted r*; *tree-less k l*; *tree-greater k r* ⟧ ⟹ *sorted* (*balance-left l k v r*)
apply (*induct l k v r rule*: *balance-left.induct*)
apply (*auto simp*: *balance-sorted*)
apply (*unfold tree-greater-prop tree-less-prop*)
by *force+*

lemma *balance-left-tree-greater*:
fixes *k* :: *'a*::*order*
assumes *k* ≪| *a k* ≪| *b k* < *x*
shows *k* ≪| *balance-left a x t b*
using *assms*
by (*induct a x t b rule*: *balance-left.induct*) *auto*

lemma *balance-left-tree-less*:
fixes *k* :: *'a*::*order*
assumes *a* |≪ *k b* |≪ *k x* < *k*
shows *balance-left a x t b* |≪ *k*
using *assms*
by (*induct a x t b rule*: *balance-left.induct*) *auto*

lemma *balance-left-in-tree*:
assumes *inv1l l inv1 r bheight l* + *1* = *bheight r*
shows *entry-in-tree k v* (*balance-left l a b r*) = (*entry-in-tree k v l* ∨ *k* = *a* ∧ *v* = *b* ∨ *entry-in-tree k v r*)
using *assms*
by (*induct l k v r rule*: *balance-left.induct*) (*auto simp*: *balance-in-tree*)

fun
balance-right :: (*'a*,*'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
balance-right a k x (*Branch R b s y c*) = *Branch R a k x* (*Branch B b s y c*) |
balance-right (*Branch B a k x b*) *s y bl* = *balance* (*Branch R a k x b*) *s y bl* |
balance-right (*Branch R a k x* (*Branch B b s y c*)) *t z bl* = *Branch R* (*balance* (*paint R a*) *k x b*) *s y* (*Branch B c t z bl*) |
balance-right t k x s = *Empty*

lemma *balance-right-inv2-with-inv1*:
assumes *inv2 lt inv2 rt bheight lt* = *bheight rt* + *1 inv1 lt*
shows *inv2* (*balance-right lt k v rt*) ∧ *bheight* (*balance-right lt k v rt*) = *bheight lt*
using *assms*
by (*induct lt k v rt rule*: *balance-right.induct*) (*auto simp*: *balance-inv2 balance-bheight*)

lemma *balance-right-inv1*: ⟦*inv1 a*; *inv1l b*; *color-of a* = *B*⟧ ⟹ *inv1* (*balance-right a k x b*)

by (*induct a k x b rule*: *balance-right.induct*) (*simp add*: *balance-inv1*)+

lemma *balance-right-inv1l*: ⟦ *inv1 lt*; *inv1l rt* ⟧ ⟹*inv1l* (*balance-right lt k x rt*)
by (*induct lt k x rt rule*: *balance-right.induct*) (*auto simp*: *balance-inv1*)

lemma *balance-right-sorted*: ⟦ *sorted l*; *sorted r*; *tree-less k l*; *tree-greater k r* ⟧ ⟹
sorted (*balance-right l k v r*)
apply (*induct l k v r rule*: *balance-right.induct*)
apply (*auto simp*:*balance-sorted*)
apply (*unfold tree-less-prop tree-greater-prop*)
by *force*+

lemma *balance-right-tree-greater*:
fixes *k* :: *'a*::*order*
assumes *k* ≪| *a k* ≪| *b k* < *x*
shows *k* ≪| *balance-right a x t b*
using *assms* **by** (*induct a x t b rule*: *balance-right.induct*) *auto*

lemma *balance-right-tree-less*:
fixes *k* :: *'a*::*order*
assumes *a* |≪ *k b* |≪ *k x* < *k*
shows *balance-right a x t b* |≪ *k*
using *assms* **by** (*induct a x t b rule*: *balance-right.induct*) *auto*

lemma *balance-right-in-tree*:
assumes *inv1 l inv1l r bheight l* = *bheight r* + *1 inv2 l inv2 r*
shows *entry-in-tree x y* (*balance-right l k v r*) = (*entry-in-tree x y l* ∨ *x* = *k* ∧
y = *v* ∨ *entry-in-tree x y r*)
using *assms* **by** (*induct l k v r rule*: *balance-right.induct*) (*auto simp*: *balance-in-tree*)

fun
combine :: (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
combine Empty x = *x*
| *combine x Empty* = *x*
| *combine* (*Branch R a k x b*) (*Branch R c s y d*) = (*case* (*combine b c*) *of*
Branch R b2 t z c2 ⇒ (*Branch R* (*Branch R a k x*
b2) *t z* (*Branch R c2 s y d*)) |
bc ⇒ *Branch R a k x* (*Branch R bc s y d*))
| *combine* (*Branch B a k x b*) (*Branch B c s y d*) = (*case* (*combine b c*) *of*
Branch R b2 t z c2 ⇒ *Branch R* (*Branch B a k x*
b2) *t z* (*Branch B c2 s y d*) |
bc ⇒ *balance-left a k x* (*Branch B bc s y d*))
| *combine a* (*Branch R b k x c*) = *Branch R* (*combine a b*) *k x c*
| *combine* (*Branch R a k x b*) *c* = *Branch R a k x* (*combine b c*)

lemma *combine-inv2*:
assumes *inv2 lt inv2 rt bheight lt* = *bheight rt*
shows *bheight* (*combine lt rt*) = *bheight lt inv2* (*combine lt rt*)

using *assms*
by (*induct lt rt rule*: *combine.induct*)
(*auto simp*: *balance-left-inv2-app split*: *rbt.splits color.splits*)

lemma *combine-inv1*:
assumes *inv1 lt inv1 rt*
shows *color-of lt* = *B* ⟹ *color-of rt* = *B* ⟹ *inv1* (*combine lt rt*)
inv1l (*combine lt rt*)
using *assms*
by (*induct lt rt rule*: *combine.induct*)
(*auto simp*: *balance-left-inv1 split*: *rbt.splits color.splits*)

lemma *combine-tree-greater*[*simp*]:
fixes *k* :: *'a*::*linorder*
assumes *k* ≪| *l k* ≪| *r*
shows *k* ≪| *combine l r*
using *assms*
by (*induct l r rule*: *combine.induct*)
(*auto simp*: *balance-left-tree-greater split*:*rbt.splits color.splits*)

lemma *combine-tree-less*[*simp*]:
fixes *k* :: *'a*::*linorder*
assumes *l* |≪ *k r* |≪ *k*
shows *combine l r* |≪ *k*
using *assms*
by (*induct l r rule*: *combine.induct*)
(*auto simp*: *balance-left-tree-less split*:*rbt.splits color.splits*)

lemma *combine-sorted*:
fixes *k* :: *'a*::*linorder*
assumes *sorted l sorted r l* |≪ *k k* ≪| *r*
shows *sorted* (*combine l r*)
using *assms* **proof** (*induct l r rule*: *combine.induct*)
case (*3 a x v b c y w d*)
hence *ineqs*: *a* |≪ *x x* ≪| *b b* |≪ *k k* ≪| *c c* |≪ *y y* ≪| *d*
by *auto*
with *3*
show *?case*
by (*cases combine b c rule*: *rbt-cases*)
(*auto*, (*metis combine-tree-greater combine-tree-less ineqs ineqs tree-less-simps*(*2*)
tree-greater-simps(*2*) *tree-greater-trans tree-less-trans*)+)
next
case (*4 a x v b c y w d*)
hence *x* < *k* ∧ *tree-greater k c* **by** *simp*
hence *tree-greater x c* **by** (*blast dest*: *tree-greater-trans*)
with *4* **have** *2*: *tree-greater x* (*combine b c*) **by** (*simp add*: *combine-tree-greater*)
from *4* **have** *k* < *y* ∧ *tree-less k b* **by** *simp*
hence *tree-less y b* **by** (*blast dest*: *tree-less-trans*)
with *4* **have** *3*: *tree-less y* (*combine b c*) **by** (*simp add*: *combine-tree-less*)

```
  show ?case
  proof (cases combine b c rule: rbt-cases)
    case Empty
    from 4 have x < y ∧ tree-greater y d by auto
    hence tree-greater x d by (blast dest: tree-greater-trans)
   with 4 Empty have sorted a and sorted (Branch B Empty y w d) and tree-less
x a and tree-greater x (Branch B Empty y w d) by auto
    with Empty show ?thesis by (simp add: balance-left-sorted)
  next
    case (Red lta va ka rta)
    with 2 4 have x < va ∧ tree-less x a by simp
    hence 5: tree-less va a by (blast dest: tree-less-trans)
    from Red 3 4 have va < y ∧ tree-greater y d by simp
    hence tree-greater va d by (blast dest: tree-greater-trans)
    with Red 2 3 4 5 show ?thesis by simp
  next
    case (Black lta va ka rta)
    from 4 have x < y ∧ tree-greater y d by auto
    hence tree-greater x d by (blast dest: tree-greater-trans)
    with Black 2 3 4 have sorted a and sorted (Branch B (combine b c) y w d)
and tree-less x a and tree-greater x (Branch B (combine b c) y w d) by auto
    with Black show ?thesis by (simp add: balance-left-sorted)
  qed
next
  case (5 va vb vd vc b x w c)
  hence k < x ∧ tree-less k (Branch B va vb vd vc) by simp
  hence tree-less x (Branch B va vb vd vc) by (blast dest: tree-less-trans)
  with 5 show ?case by (simp add: combine-tree-less)
next
  case (6 a x v b va vb vd vc)
  hence x < k ∧ tree-greater k (Branch B va vb vd vc) by simp
  hence tree-greater x (Branch B va vb vd vc) by (blast dest: tree-greater-trans)
  with 6 show ?case by (simp add: combine-tree-greater)
qed simp+

lemma combine-in-tree:
  assumes inv2 l inv2 r bheight l = bheight r inv1 l inv1 r
  shows entry-in-tree k v (combine l r) = (entry-in-tree k v l ∨ entry-in-tree k v
r)
using assms
proof (induct l r rule: combine.induct)
  case (4 - - - b c)
  hence a: bheight (combine b c) = bheight b by (simp add: combine-inv2)
  from 4 have b: inv1l (combine b c) by (simp add: combine-inv1)

  show ?case
  proof (cases combine b c rule: rbt-cases)
    case Empty
    with 4 a show ?thesis by (auto simp: balance-left-in-tree)
```

next
case (*Red lta ka va rta*)
with *4* **show** *?thesis* **by** *auto*
next
case (*Black lta ka va rta*)
with *a b 4* **show** *?thesis* **by** (*auto simp*: *balance-left-in-tree*)
qed
qed (*auto split*: *rbt.splits color.splits*)

fun
del-from-left :: ('*a*::*linorder*) ⇒ ('*a*,'*b*) *rbt* ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
and
del-from-right :: ('*a*::*linorder*) ⇒ ('*a*,'*b*) *rbt* ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt* **and**
del :: ('*a*::*linorder*) ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
del x Empty = *Empty* |
del x (*Branch c a y s b*) = (*if x* < *y then del-from-left x a y s b else* (*if x* > *y then del-from-right x a y s b else combine a b*)) |
del-from-left x (*Branch B lt z v rt*) *y s b* = *balance-left* (*del x* (*Branch B lt z v rt*)) *y s b* |
del-from-left x a y s b = *Branch R* (*del x a*) *y s b* |
del-from-right x a y s (*Branch B lt z v rt*) = *balance-right a y s* (*del x* (*Branch B lt z v rt*)) |
del-from-right x a y s b = *Branch R a y s* (*del x b*)

lemma
assumes *inv2 lt inv1 lt*
shows
⟦*inv2 rt*; *bheight lt* = *bheight rt*; *inv1 rt*⟧ ⟹
inv2 (*del-from-left x lt k v rt*) ∧ *bheight* (*del-from-left x lt k v rt*) = *bheight lt* ∧ (*color-of lt* = *B* ∧ *color-of rt* = *B* ∧ *inv1* (*del-from-left x lt k v rt*) ∨ (*color-of lt* ≠ *B* ∨ *color-of rt* ≠ *B*) ∧ *inv1l* (*del-from-left x lt k v rt*))
and ⟦*inv2 rt*; *bheight lt* = *bheight rt*; *inv1 rt*⟧ ⟹
inv2 (*del-from-right x lt k v rt*) ∧ *bheight* (*del-from-right x lt k v rt*) = *bheight lt* ∧ (*color-of lt* = *B* ∧ *color-of rt* = *B* ∧ *inv1* (*del-from-right x lt k v rt*) ∨ (*color-of lt* ≠ *B* ∨ *color-of rt* ≠ *B*) ∧ *inv1l* (*del-from-right x lt k v rt*))
and *del-inv1-inv2*: *inv2* (*del x lt*) ∧ (*color-of lt* = *R* ∧ *bheight* (*del x lt*) = *bheight lt* ∧ *inv1* (*del x lt*)
∨ *color-of lt* = *B* ∧ *bheight* (*del x lt*) = *bheight lt* − *1* ∧ *inv1l* (*del x lt*))
using *assms*
proof (*induct x lt k v rt* **and** *x lt k v rt* **and** *x lt rule*: *del-from-left-del-from-right-del.induct*)
case (*2 y c - y'*)
have *y* = *y'* ∨ *y* < *y'* ∨ *y* > *y'* **by** *auto*
thus *?case* **proof** (*elim disjE*)
assume *y* = *y'*
with *2* **show** *?thesis* **by** (*cases c*) (*simp add*: *combine-inv2 combine-inv1*)+
next
assume *y* < *y'*

with *2* **show** *?thesis* **by** (*cases c*) *auto*
next
assume $y' < y$
with *2* **show** *?thesis* **by** (*cases c*) *auto*
qed
next
case (*3 y lt z v rta y' ss bb*)
thus *?case* **by** (*cases color-of* (*Branch B lt z v rta*) = *B* ∧ *color-of bb* = *B*) (*simp add*: *balance-left-inv2-with-inv1 balance-left-inv1 balance-left-inv1l*)+
next
case (*5 y a y' ss lt z v rta*)
thus *?case* **by** (*cases color-of a* = *B* ∧ *color-of* (*Branch B lt z v rta*) = *B*) (*simp add*: *balance-right-inv2-with-inv1 balance-right-inv1 balance-right-inv1l*)+
next
case (*6-1 y a y' ss*) **thus** *?case* **by** (*cases color-of a* = *B* ∧ *color-of Empty* = *B*) *simp*+
qed *auto*

lemma
del-from-left-tree-less: ⟦*tree-less v lt*; *tree-less v rt*; *k* < *v*⟧ ⟹ *tree-less v* (*del-from-left x lt k y rt*)
and *del-from-right-tree-less*: ⟦*tree-less v lt*; *tree-less v rt*; *k* < *v*⟧ ⟹ *tree-less v* (*del-from-right x lt k y rt*)
and *del-tree-less*: *tree-less v lt* ⟹ *tree-less v* (*del x lt*)
by (*induct x lt k y rt* **and** *x lt k y rt* **and** *x lt rule*: *del-from-left-del-from-right-del.induct*)

(*auto simp*: *balance-left-tree-less balance-right-tree-less*)

lemma *del-from-left-tree-greater*: ⟦*tree-greater v lt*; *tree-greater v rt*; *k* > *v*⟧ ⟹ *tree-greater v* (*del-from-left x lt k y rt*)
and *del-from-right-tree-greater*: ⟦*tree-greater v lt*; *tree-greater v rt*; *k* > *v*⟧ ⟹ *tree-greater v* (*del-from-right x lt k y rt*)
and *del-tree-greater*: *tree-greater v lt* ⟹ *tree-greater v* (*del x lt*)
by (*induct x lt k y rt* **and** *x lt k y rt* **and** *x lt rule*: *del-from-left-del-from-right-del.induct*)
(*auto simp*: *balance-left-tree-greater balance-right-tree-greater*)

lemma ⟦*sorted lt*; *sorted rt*; *tree-less k lt*; *tree-greater k rt*⟧ ⟹ *sorted* (*del-from-left x lt k y rt*)
and ⟦*sorted lt*; *sorted rt*; *tree-less k lt*; *tree-greater k rt*⟧ ⟹ *sorted* (*del-from-right x lt k y rt*)
and *del-sorted*: *sorted lt* ⟹ *sorted* (*del x lt*)
proof (*induct x lt k y rt* **and** *x lt k y rt* **and** *x lt rule*: *del-from-left-del-from-right-del.induct*)
case (*3 x lta zz v rta yy ss bb*)
from *3* **have** *tree-less yy* (*Branch B lta zz v rta*) **by** *simp*
hence *tree-less yy* (*del x* (*Branch B lta zz v rta*)) **by** (*rule del-tree-less*)
with *3* **show** *?case* **by** (*simp add*: *balance-left-sorted*)
next
case (*4-2 x vaa vbb vdd vc yy ss bb*)
hence *tree-less yy* (*Branch R vaa vbb vdd vc*) **by** *simp*

hence *tree-less yy* (*del x* (*Branch R vaa vbb vdd vc*)) **by** (*rule del-tree-less*)
with *4-2* **show** *?case* **by** *simp*
next
case (*5 x aa yy ss lta zz v rta*)
hence *tree-greater yy* (*Branch B lta zz v rta*) **by** *simp*
hence *tree-greater yy* (*del x* (*Branch B lta zz v rta*)) **by** (*rule del-tree-greater*)
with *5* **show** *?case* **by** (*simp add*: *balance-right-sorted*)
next
case (*6-2 x aa yy ss vaa vbb vdd vc*)
hence *tree-greater yy* (*Branch R vaa vbb vdd vc*) **by** *simp*
hence *tree-greater yy* (*del x* (*Branch R vaa vbb vdd vc*)) **by** (*rule del-tree-greater*)
with *6-2* **show** *?case* **by** *simp*
qed (*auto simp*: *combine-sorted*)

lemma ⟦*sorted lt*; *sorted rt*; *tree-less kt lt*; *tree-greater kt rt*; *inv1 lt*; *inv1 rt*; *inv2 lt*; *inv2 rt*; *bheight lt* = *bheight rt*; *x* < *kt*⟧ ⟹ *entry-in-tree k v* (*del-from-left x lt kt y rt*) = (*False* ∨ (*x* ≠ *k* ∧ *entry-in-tree k v* (*Branch c lt kt y rt*)))
and ⟦*sorted lt*; *sorted rt*; *tree-less kt lt*; *tree-greater kt rt*; *inv1 lt*; *inv1 rt*; *inv2 lt*; *inv2 rt*; *bheight lt* = *bheight rt*; *x* > *kt*⟧ ⟹ *entry-in-tree k v* (*del-from-right x lt kt y rt*) = (*False* ∨ (*x* ≠ *k* ∧ *entry-in-tree k v* (*Branch c lt kt y rt*)))
and *del-in-tree*: ⟦*sorted t*; *inv1 t*; *inv2 t*⟧ ⟹ *entry-in-tree k v* (*del x t*) = (*False* ∨ (*x* ≠ *k* ∧ *entry-in-tree k v t*))
proof (*induct x lt kt y rt* **and** *x t rule*: *del-from-left-del-from-right-del.induct*)
case (*2 xx c aa yy ss bb*)
have *xx* = *yy* ∨ *xx* < *yy* ∨ *xx* > *yy* **by** *auto*
from *this 2* **show** *?case* **proof** (*elim disjE*)
assume *xx* = *yy*
with *2* **show** *?thesis* **proof** (*cases xx* = *k*)
case *True*
from *2* ‹*xx* = *yy*› ‹*xx* = *k*› **have** *sorted* (*Branch c aa yy ss bb*) ∧ *k* = *yy* **by** *simp*
hence ¬ *entry-in-tree k v aa* ¬ *entry-in-tree k v bb* **by** (*auto simp*: *tree-less-nit tree-greater-prop*)
with ‹*xx* = *yy*› *2* ‹*xx* = *k*› **show** *?thesis* **by** (*simp add*: *combine-in-tree*)
qed (*simp add*: *combine-in-tree*)
qed *simp+*
next
case (*3 xx lta zz vv rta yy ss bb*)
def *mt*[*simp*]: *mt* == *Branch B lta zz vv rta*
from *3* **have** *inv2 mt* ∧ *inv1 mt* **by** *simp*
hence *inv2* (*del xx mt*) ∧ (*color-of mt* = *R* ∧ *bheight* (*del xx mt*) = *bheight mt* ∧ *inv1* (*del xx mt*) ∨ *color-of mt* = *B* ∧ *bheight* (*del xx mt*) = *bheight mt* − *1* ∧ *inv1l* (*del xx mt*)) **by** (*blast dest*: *del-inv1-inv2*)
with *3* **have** *4*: *entry-in-tree k v* (*del-from-left xx mt yy ss bb*) = (*False* ∨ *xx* ≠ *k* ∧ *entry-in-tree k v mt* ∨ (*k* = *yy* ∧ *v* = *ss*) ∨ *entry-in-tree k v bb*) **by** (*simp add*: *balance-left-in-tree*)
thus *?case* **proof** (*cases xx* = *k*)
case *True*
from *3 True* **have** *tree-greater yy bb* ∧ *yy* > *k* **by** *simp*

hence *tree-greater k bb* **by** (*blast dest*: *tree-greater-trans*)
with *3 4 True* **show** *?thesis* **by** (*auto simp*: *tree-greater-nit*)
qed *auto*
next
case (*4-1 xx yy ss bb*)
show *?case* **proof** (*cases xx = k*)
case *True*
with *4-1* **have** *tree-greater yy bb ∧ k < yy* **by** *simp*
hence *tree-greater k bb* **by** (*blast dest*: *tree-greater-trans*)
with *4-1* ‹*xx = k*›
have *entry-in-tree k v* (*Branch R Empty yy ss bb*) = *entry-in-tree k v Empty* **by** (*auto simp*: *tree-greater-nit*)
thus *?thesis* **by** *auto*
qed *simp+*
next
case (*4-2 xx vaa vbb vdd vc yy ss bb*)
thus *?case* **proof** (*cases xx = k*)
case *True*
with *4-2* **have** *k < yy ∧ tree-greater yy bb* **by** *simp*
hence *tree-greater k bb* **by** (*blast dest*: *tree-greater-trans*)
with *True 4-2* **show** *?thesis* **by** (*auto simp*: *tree-greater-nit*)
qed *auto*
next
case (*5 xx aa yy ss lta zz vv rta*)
def *mt*[*simp*]: *mt == Branch B lta zz vv rta*
from *5* **have** *inv2 mt ∧ inv1 mt* **by** *simp*
hence *inv2* (*del xx mt*) ∧ (*color-of mt = R ∧ bheight* (*del xx mt*) = *bheight mt ∧ inv1* (*del xx mt*) ∨ *color-of mt = B ∧ bheight* (*del xx mt*) = *bheight mt − 1 ∧ inv1l* (*del xx mt*)) **by** (*blast dest*: *del-inv1-inv2*)
with *5* **have** *3*: *entry-in-tree k v* (*del-from-right xx aa yy ss mt*) = (*entry-in-tree k v aa ∨* (*k = yy ∧ v = ss*) ∨ *False ∨ xx ≠ k ∧ entry-in-tree k v mt*) **by** (*simp add*: *balance-right-in-tree*)
thus *?case* **proof** (*cases xx = k*)
case *True*
from *5 True* **have** *tree-less yy aa ∧ yy < k* **by** *simp*
hence *tree-less k aa* **by** (*blast dest*: *tree-less-trans*)
with *3 5 True* **show** *?thesis* **by** (*auto simp*: *tree-less-nit*)
qed *auto*
next
case (*6-1 xx aa yy ss*)
show *?case* **proof** (*cases xx = k*)
case *True*
with *6-1* **have** *tree-less yy aa ∧ k > yy* **by** *simp*
hence *tree-less k aa* **by** (*blast dest*: *tree-less-trans*)
with *6-1* ‹*xx = k*› **show** *?thesis* **by** (*auto simp*: *tree-less-nit*)
qed *simp*
next
case (*6-2 xx aa yy ss vaa vbb vdd vc*)
thus *?case* **proof** (*cases xx = k*)

```
    case True
    with 6-2 have k > yy ∧ tree-less yy aa by simp
    hence tree-less k aa by (blast dest: tree-less-trans)
    with True 6-2 show ?thesis by (auto simp: tree-less-nit)
  qed auto
qed simp


definition delete where
  delete-def: delete k t = paint B (del k t)

theorem delete-is-rbt [simp]: assumes is-rbt t shows is-rbt (delete k t)
proof −
  from assms have inv2 t and inv1 t unfolding is-rbt-def by auto
  hence inv2 (del k t) ∧ (color-of t = R ∧ bheight (del k t) = bheight t ∧ inv1
(del k t) ∨ color-of t = B ∧ bheight (del k t) = bheight t − 1 ∧ inv1l (del k t))
by (rule del-inv1-inv2)
  hence inv2 (del k t) ∧ inv1l (del k t) by (cases color-of t) auto
  with assms show ?thesis
    unfolding is-rbt-def delete-def
    by (auto intro: paint-sorted del-sorted)
qed

lemma delete-in-tree:
  assumes is-rbt t
  shows entry-in-tree k v (delete x t) = (x ≠ k ∧ entry-in-tree k v t)
  using assms unfolding is-rbt-def delete-def
  by (auto simp: del-in-tree)

lemma lookup-delete:
  assumes is-rbt: is-rbt t
  shows lookup (delete k t) = (lookup t)|`(−{k})
proof
  fix x
  show lookup (delete k t) x = (lookup t |` (−{k})) x
  proof (cases x = k)
    assume x = k
    with is-rbt show ?thesis
      by (cases lookup (delete k t) k) (auto simp: lookup-in-tree delete-in-tree)
  next
    assume x ≠ k
    thus ?thesis
    by auto (metis is-rbt delete-is-rbt delete-in-tree is-rbt-sorted lookup-from-in-tree)
  qed
qed
```

65.5 Union

primrec

union-with-key :: ('*a*::*linorder* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
union-with-key f t Empty = *t*
| *union-with-key f t* (*Branch c lt k v rt*) = *union-with-key f* (*union-with-key f* (*insert-with-key f k v t*) *lt*) *rt*

lemma *unionwk-sorted*: *sorted lt* ⟹ *sorted* (*union-with-key f lt rt*)
by (*induct rt arbitrary*: *lt*) (*auto simp*: *insertwk-sorted*)
theorem *unionwk-is-rbt*[*simp*]: *is-rbt lt* ⟹ *is-rbt* (*union-with-key f lt rt*)
by (*induct rt arbitrary*: *lt*) (*simp add*: *insertwk-is-rbt*)+

definition
union-with **where**
union-with f = *union-with-key* (λ-. *f*)

theorem *unionw-is-rbt*: *is-rbt lt* ⟹ *is-rbt* (*union-with f lt rt*) **unfolding** *union-with-def*
by *simp*

definition *union* **where**
union = *union-with-key* (%- - *rv*. *rv*)

theorem *union-is-rbt*: *is-rbt lt* ⟹ *is-rbt* (*union lt rt*) **unfolding** *union-def* **by** *simp*

lemma *union-Branch*[*simp*]:
union t (*Branch c lt k v rt*) = *union* (*union* (*insert k v t*) *lt*) *rt*
unfolding *union-def insert-def*
by *simp*

lemma *lookup-union*:
assumes *is-rbt s sorted t*
shows *lookup* (*union s t*) = *lookup s* ++ *lookup t*
using *assms*
proof (*induct t arbitrary*: *s*)
case *Empty* **thus** *?case* **by** (*auto simp*: *union-def*)
next
case (*Branch c l k v r s*)
then have *sorted r sorted l l* |≪ *k k* ≪| *r* **by** *auto*

have *meq*: *lookup s*(*k* ↦ *v*) ++ *lookup l* ++ *lookup r* =
lookup s ++
(λ*a*. *if a* < *k then lookup l a*
else if k < *a then lookup r a else Some v*) (**is** *?m1* = *?m2*)
proof (*rule ext*)
fix *a*

have *k* < *a* ∨ *k* = *a* ∨ *k* > *a* **by** *auto*
thus *?m1 a* = *?m2 a*

proof (*elim disjE*)
assume $k < a$
with ‹l |≪ k› **have** l |≪ a **by** (*rule tree-less-trans*)
with ‹$k < a$› **show** *?thesis*
by (*auto simp*: *map-add-def split*: *option.splits*)
next
assume $k = a$
with ‹l |≪ k› ‹k ≪| r›
show *?thesis* **by** (*auto simp*: *map-add-def*)
next
assume $a < k$
from *this* ‹k ≪| r› **have** a ≪| r **by** (*rule tree-greater-trans*)
with ‹$a < k$› **show** *?thesis*
by (*auto simp*: *map-add-def split*: *option.splits*)
qed
qed

from *Branch* **have** *is-rbt*: *is-rbt* (*RBT-Impl.union* (*RBT-Impl.insert k v s*) *l*)
by (*auto intro*: *union-is-rbt insert-is-rbt*)
with *Branch* **have** *IHs*:
lookup (*union* (*union* (*insert k v s*) *l*) *r*) = *lookup* (*union* (*insert k v s*) *l*) ++ *lookup r*
lookup (*union* (*insert k v s*) *l*) = *lookup* (*insert k v s*) ++ *lookup l*
by *auto*

with *meq* **show** *?case*
by (*auto simp*: *lookup-insert*[*OF Branch*(*3*)])

qed

65.6 Modifying existing entries

primrec
map-entry :: *'a*::*linorder* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt*
where
map-entry k f Empty = *Empty*
| *map-entry k f* (*Branch c lt x v rt*) =
(*if k < x then Branch c* (*map-entry k f lt*) *x v rt*
else if k > x then (*Branch c lt x v* (*map-entry k f rt*))
else Branch c lt x (*f v*) *rt*)

lemma *map-entry-color-of*: *color-of* (*map-entry k f t*) = *color-of t* **by** (*induct t*) *simp+*
lemma *map-entry-inv1*: *inv1* (*map-entry k f t*) = *inv1 t* **by** (*induct t*) (*simp add*: *map-entry-color-of*)+
lemma *map-entry-inv2*: *inv2* (*map-entry k f t*) = *inv2 t bheight* (*map-entry k f t*) = *bheight t* **by** (*induct t*) *simp+*
lemma *map-entry-tree-greater*: *tree-greater a* (*map-entry k f t*) = *tree-greater a t* **by** (*induct t*) *simp+*

lemma *map-entry-tree-less*: *tree-less a* (*map-entry k f t*) = *tree-less a t* **by** (*induct t*) *simp+*
lemma *map-entry-sorted*: *sorted* (*map-entry k f t*) = *sorted t*
by (*induct t*) (*simp-all add*: *map-entry-tree-less map-entry-tree-greater*)

theorem *map-entry-is-rbt* [*simp*]: *is-rbt* (*map-entry k f t*) = *is-rbt t*
unfolding *is-rbt-def* **by** (*simp add*: *map-entry-inv2 map-entry-color-of map-entry-sorted map-entry-inv1*)

theorem *lookup-map-entry*:
lookup (*map-entry k f t*) = (*lookup t*)(*k* := *Option.map f* (*lookup t k*))
by (*induct t*) (*auto split*: *option.splits simp add*: *expand-fun-eq*)

65.7 Mapping all entries

primrec
map :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*c*) *rbt*
where
map f Empty = *Empty*
| *map f* (*Branch c lt k v rt*) = *Branch c* (*map f lt*) *k* (*f k v*) (*map f rt*)

lemma *map-entries* [*simp*]: *entries* (*map f t*) = *List.map* (λ(*k*, *v*). (*k*, *f k v*)) (*entries t*)
by (*induct t*) *auto*
lemma *map-keys* [*simp*]: *keys* (*map f t*) = *keys t* **by** (*simp add*: *keys-def split-def*)
lemma *map-tree-greater*: *tree-greater k* (*map f t*) = *tree-greater k t* **by** (*induct t*) *simp+*
lemma *map-tree-less*: *tree-less k* (*map f t*) = *tree-less k t* **by** (*induct t*) *simp+*
lemma *map-sorted*: *sorted* (*map f t*) = *sorted t* **by** (*induct t*) (*simp add*: *map-tree-less map-tree-greater*)+
lemma *map-color-of*: *color-of* (*map f t*) = *color-of t* **by** (*induct t*) *simp+*
lemma *map-inv1*: *inv1* (*map f t*) = *inv1 t* **by** (*induct t*) (*simp add*: *map-color-of*)+
lemma *map-inv2*: *inv2* (*map f t*) = *inv2 t bheight* (*map f t*) = *bheight t* **by** (*induct t*) *simp+*
theorem *map-is-rbt* [*simp*]: *is-rbt* (*map f t*) = *is-rbt t*
unfolding *is-rbt-def* **by** (*simp add*: *map-inv1 map-inv2 map-sorted map-color-of*)

theorem *lookup-map*: *lookup* (*map f t*) *x* = *Option.map* (*f x*) (*lookup t x*)
by (*induct t*) *auto*

65.8 Folding over entries

definition *fold* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a*, '*b*) *rbt* ⇒ '*c* ⇒ '*c* **where**
fold f t s = *foldl* (λ*s* (*k*, *v*). *f k v s*) *s* (*entries t*)

lemma *fold-simps* [*simp*, *code*]:
fold f Empty = *id*
fold f (*Branch c lt k v rt*) = *fold f rt* ∘ *f k v* ∘ *fold f lt*
by (*simp-all add*: *fold-def expand-fun-eq*)

65.9 Bulkloading a tree

definition *bulkload* :: ('*a* × '*b*) *list* ⇒ ('*a*::*linorder*, '*b*) *rbt* **where**
bulkload xs = *foldr* (λ(*k*, *v*). *insert k v*) *xs Empty*

lemma *bulkload-is-rbt* [*simp*, *intro*]:
is-rbt (*bulkload xs*)
unfolding *bulkload-def* **by** (*induct xs*) *auto*

lemma *lookup-bulkload*:
lookup (*bulkload xs*) = *map-of xs*
proof −
obtain *ys* **where** *ys* = *rev xs* **by** *simp*
have ⋀*t*. *is-rbt t* ⟹
lookup (*foldl* (λ*t* (*k*, *v*). *insert k v t*) *t ys*) = *lookup t* ++ *map-of* (*rev ys*)
by (*induct ys*) (*simp-all add*: *bulkload-def split-def lookup-insert*)
from *this Empty-is-rbt* **have**
lookup (*foldl* (λ*t* (*k*, *v*). *insert k v t*) *Empty* (*rev xs*)) = *lookup Empty* ++ *map-of xs*
by (*simp add*: ‹*ys* = *rev xs*›)
then show *?thesis* **by** (*simp add*: *bulkload-def foldl-foldr lookup-Empty split-def*)
qed

hide-const (**open**) *Empty insert delete entries keys bulkload lookup map-entry map fold union sorted*

end

66 RBT: Abstract type of Red-Black Trees

This theory defines abstract red-black trees as an efficient representation of finite maps, backed by the implementation in *RBT-Impl*.

66.1 Data type and invariant

The type ('*k*, '*v*) *RBT-Impl.rbt* denotes red-black trees with keys of type '*k* and values of type '*v*. To function properly, the key type musorted belong to the *linorder* class.

A value *t* of this type is a valid red-black tree if it satisfies the invariant *is-rbt t*. The abstract type ('*k*, '*v*) *RBT.rbt* always obeys this invariant, and for this reason you should only use this in our application. Going back to ('*k*, '*v*) *RBT-Impl.rbt* may be necessary in proofs if not yet proven properties about the operations must be established.

The interpretation function *RBT.lookup* returns the partial map represented by a red-black tree:

RBT.lookup::('*a*, '*b*) *RBT.rbt* ⇒ '*a* ⇀ '*b*

This function should be used for reasoning about the semantics of the RBT operations. Furthermore, it implements the lookup functionality for the data structure: It is executable and the lookup is performed in $O(\log n)$.

66.2 Operations

Currently, the following operations are supported:

RBT.empty::('a, 'b) *RBT.rbt*

Returns the empty tree. $O(1)$

RBT.insert::'a ⇒ 'b ⇒ ('a, 'b) *RBT.rbt* ⇒ ('a, 'b) *RBT.rbt*

Updates the map at a given position. $O(\log n)$

RBT.delete::'a ⇒ ('a, 'b) *RBT.rbt* ⇒ ('a, 'b) *RBT.rbt*

Deletes a map entry at a given position. $O(\log n)$

RBT.entries::('a, 'b) *RBT.rbt* ⇒ ('a × 'b) *list*

Return a corresponding key-value list for a tree.

RBT.bulkload::('a × 'b) *list* ⇒ ('a, 'b) *RBT.rbt*

Builds a tree from a key-value list.

RBT.map-entry::'a ⇒ ('b ⇒ 'b) ⇒ ('a, 'b) *RBT.rbt* ⇒ ('a, 'b) *RBT.rbt*

Maps a single entry in a tree.

RBT.map::('a ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) *RBT.rbt* ⇒ ('a, 'b) *RBT.rbt*

Maps all values in a tree. $O(n)$

RBT.fold::('a ⇒ 'b ⇒ 'c ⇒ 'c) ⇒ ('a, 'b) *RBT.rbt* ⇒ 'c ⇒ 'c

Folds over all entries in a tree. $O(n)$

66.3 Invariant preservation

is-rbt rbt.Empty (*Empty-is-rbt*)
is-rbt ?t ⟹ *is-rbt* (*RBT-Impl.insert ?k ?v ?t*) (*insert-is-rbt*)
is-rbt ?t ⟹ *is-rbt* (*RBT-Impl.delete ?k ?t*) (*delete-is-rbt*)
is-rbt (*RBT-Impl.bulkload ?xs*) (*bulkload-is-rbt*)
is-rbt (*RBT-Impl.map-entry ?k ?f ?t*) = *is-rbt ?t* (*map-entry-is-rbt*)
is-rbt (*RBT-Impl.map ?f ?t*) = *is-rbt ?t* (*map-is-rbt*)
is-rbt ?lt ⟹ *is-rbt* (*RBT-Impl.union ?lt ?rt*) (*union-is-rbt*)

66.4 Map Semantics

lookup-empty

$RBT.lookup\ RBT.empty = Map.empty$

lookup-insert

$RBT.lookup\ (RBT.insert\ ?k\ ?v\ ?t) = RBT.lookup\ ?t(?k \mapsto ?v)$

lookup-delete

$RBT.lookup\ (RBT.delete\ ?k\ ?t) = (RBT.lookup\ ?t)(?k := None)$

lookup-bulkload

$RBT.lookup\ (RBT.bulkload\ ?xs) = map\text{-}of\ ?xs$

lookup-map

$RBT.lookup\ (RBT.map\ ?f\ ?t)\ ?k = Option.map\ (?f\ ?k)\ (RBT.lookup\ ?t\ ?k)$

end

67 SML-Quickcheck: Install quickcheck of SML code generator

theory *SML-Quickcheck*
imports *Main*
begin

setup ⟪
InductiveCodegen.quickcheck-setup #>
Quickcheck.add-generator (SML, Codegen.test-term)
⟫

end

68 State-Monad: Combinator syntax for generic, open state monads (single threaded monads)

theory *State-Monad*
imports *Main*
begin

68.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is transformed single-threaded).

To enter from the Haskell world, http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

68.2 State transformations and combinators

We classify functions operating on states into two categories:

transformations with type signature $\sigma \Rightarrow \sigma'$, transforming a state.

"yielding" transformations with type signature $\sigma \Rightarrow \alpha \times \sigma'$, "yielding" a side result while transforming a state.

queries with type signature $\sigma \Rightarrow \alpha$, computing a result dependent on a state.

By convention we write σ for types representing states and α, β, γ, ... for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type σ are used in a single-threaded way: after application of a transformation on a value of type σ, the former value should not be used again. To achieve this, we use a set of monad combinators:

notation *fcomp* (**infixl** *o>* *60*)
notation (*xsymbols*) *fcomp* (**infixl** *o>* *60*)
notation *scomp* (**infixl** *o->* *60*)
notation (*xsymbols*) *scomp* (**infixl** *o→* *60*)

abbreviation (*input*)
return ≡ *Pair*

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

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

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *Pair* for this purpose.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.
- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units ("open monad").
- The type of states may change due to a transformation.

68.3 Monad laws

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

lemmas *monad-simp = Pair-scomp scomp-Pair id-fcomp fcomp-id scomp-scomp scomp-fcomp fcomp-scomp fcomp-assoc*

Evaluation of monadic expressions by force:

lemmas *monad-collapse = monad-simp fcomp-apply scomp-apply split-beta*

68.4 Syntax

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

nonterminals *do-expr*

syntax
-do :: *do-expr* ⇒ *'a*
(*do - done* [*12*] *12*)
-scomp :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*- <− -;// -* [*1000*, *13*, *12*] *12*)
-fcomp :: *'a* ⇒ *do-expr* ⇒ *do-expr*
(*-;// -* [*13*, *12*] *12*)
-let :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*let - = -;// -* [*1000*, *13*, *12*] *12*)
-done :: *'a* ⇒ *do-expr*
(*-* [*12*] *12*)

syntax (*xsymbols*)

```
  -scomp :: pttrn ⇒ 'a ⇒ do-expr ⇒ do-expr
    (- ← -;// - [1000, 13, 12] 12)

translations
  -do f => f
  -scomp x f g => f o→ (λx. g)
  -fcomp f g => f o> g
  -let x t f => CONST Let t (λx. f)
  -done f => f

print-translation ⟪
let
  fun dest-abs-eta (Abs (abs as (-, ty, -))) =
        let
          val (v, t) = Syntax.variant-abs abs;
        in (Free (v, ty), t) end
    | dest-abs-eta t =
        let
          val (v, t) = Syntax.variant-abs (, dummyT, t $ Bound 0);
        in (Free (v, dummyT), t) end;
  fun unfold-monad (Const (@{const-syntax scomp}, -) $ f $ g) =
        let
          val (v, g') = dest-abs-eta g;
        in Const (@{syntax-const -scomp}, dummyT) $ v $ f $ unfold-monad g' end
    | unfold-monad (Const (@{const-syntax fcomp}, -) $ f $ g) =
        Const (@{syntax-const -fcomp}, dummyT) $ f $ unfold-monad g
    | unfold-monad (Const (@{const-syntax Let}, -) $ f $ g) =
        let
          val (v, g') = dest-abs-eta g;
        in Const (@{syntax-const -let}, dummyT) $ v $ f $ unfold-monad g' end
    | unfold-monad (Const (@{const-syntax Pair}, -) $ f) =
        Const (@{const-syntax return}, dummyT) $ f
    | unfold-monad f = f;
  fun contains-scomp (Const (@{const-syntax scomp}, -) $ - $ -) = true
    | contains-scomp (Const (@{const-syntax fcomp}, -) $ - $ t) =
        contains-scomp t
    | contains-scomp (Const (@{const-syntax Let}, -) $ - $ Abs (-, -, t)) =
        contains-scomp t;
  fun scomp-monad-tr' (f::g::ts) = list-comb
    (Const (@{syntax-const -do}, dummyT) $
      unfold-monad (Const (@{const-syntax scomp}, dummyT) $ f $ g), ts);
  fun fcomp-monad-tr' (f::g::ts) =
    if contains-scomp g then list-comb
      (Const (@{syntax-const -do}, dummyT) $
        unfold-monad (Const (@{const-syntax fcomp}, dummyT) $ f $ g), ts)
    else raise Match;
  fun Let-monad-tr' (f :: (g as Abs (-, -, g')) :: ts) =
    if contains-scomp g' then list-comb
      (Const (@{syntax-const -do}, dummyT) $
```

```
      unfold-monad (Const (@{const-syntax Let}, dummyT) $ f $ g), ts)
    else raise Match;
in
 [(@{const-syntax scomp}, scomp-monad-tr'),
  (@{const-syntax fcomp}, fcomp-monad-tr'),
  (@{const-syntax Let}, Let-monad-tr')]
end;
⟫
```

For an example, see HOL/Extraction/Higman.thy.

end

69 Sum-Of-Squares: A decision method for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming

theory *Sum-Of-Squares*
imports *Complex-Main*
uses
positivstellensatz.ML
Sum-Of-Squares/sum-of-squares.ML
Sum-Of-Squares/positivstellensatz-tools.ML
Sum-Of-Squares/sos-wrapper.ML
begin

In order to use the method sos, call it with (*sos remote-csdp*) to use the remote solver. Or install CSDP (https://projects.coin-or.org/Csdp), configure the Isabelle setting *CSDP-EXE*, and call it with (*sos csdp*). By default, sos calls *remote-csdp*. This can take of the order of a minute for one sos call, because sos calls CSDP repeatedly. If you install CSDP locally, sos calls typically takes only a few seconds. sos generates a certificate which can be used to repeat the proof without calling an external prover.

setup *SOS-Wrapper.setup*

Tests

lemma (*3*::*real*) * *x* + *7* * *a* < *4* & *3* < *2* * *x* ⟹ *a* < *0*
by (*sos-cert* ((*R*<*1* + (((*A*<*1* * *R*<*1*) * (*R*<*2* * [*1*] ^ *2*)) + (((*A*<*0* * *R*<*1*) * (*R*<*3* * [*1*] ^ *2*)) + ((*A*<=*0* * *R*<*1*) * (*R*<*14* * [*1*] ^ *2*)))))))

lemma *a1* >= *0* & *a2* >= *0* ∧ (*a1* * *a1* + *a2* * *a2* = *b1* * *b1* + *b2* * *b2* + *2*) ∧ (*a1* * *b1* + *a2* * *b2* = *0*) −−> *a1* * *a2* − *b1* * *b2* >= (*0*::*real*)
by (*sos-cert* (((*A*<*0* * *R*<*1*) + (([~ *1/2*a1*b2* + ~ *1/2*a2*b1*] * *A*=*0*) + (([~ *1/2*a1*a2* + *1/2*b1*b2*] * *A*=*1*) + (((*A*<*0* * *R*<*1*) * ((*R*<*1/2* * [*b2*] ^ *2*) + (*R*<*1/2* * [*b1*] ^ *2*))) + ((*A*<=*0* * (*A*<=*1* * *R*<*1*)) * ((*R*<*1/2* * [*b2*] ^ *2*) + ((*R*<*1/2* * [*b1*] ^ *2*) + ((*R*<*1/2* * [*a2*] ^ *2*) + (*R*<*1/2* * [*a1*] ^ *2*)))))))))))

lemma (*3*::*real*) * *x* + *7* * *a* < *4* & *3* < *2* * *x* --> *a* < *0*
by (*sos-cert* ((*R*<*1* + (((*A*<*1* * *R*<*1*) * (*R*<*2* * [*1*] ^ *2*)) + (((*A*<*0* * *R*<*1*) * (*R*<*3* * [*1*] ^ *2*)) + ((*A*<=*0* * *R*<*1*) * (*R*<*14* * [*1*] ^ *2*)))))))

lemma (*0*::*real*) <= *x* & *x* <= *1* & *0* <= *y* & *y* <= *1* --> *x* ^ *2* + *y* ^ *2* < *1* |(*x* - *1*) ^ *2* + *y* ^ *2* < *1* | *x* ^ *2* + (*y* - *1*) ^ *2* < *1* | (*x* - *1*) ^ *2* + (*y* - *1*) ^ *2* < *1*
by (*sos-cert* ((*R*<*1* + (((*A*<=*3* * (*A*<=*4* * *R*<*1*)) * (*R*<*1* * [*1*] ^ *2*)) + (((*A*<=*2* * (*A*<=*7* * *R*<*1*)) * (*R*<*1* * [*1*] ^ *2*)) + (((*A*<=*1* * (*A*<=*6* * *R*<*1*)) * (*R*<*1* * [*1*] ^ *2*)) + ((*A*<=*0* * (*A*<=*5* * *R*<*1*)) * (*R*<*1* * [*1*] ^ *2*)))))))

lemma (*0*::*real*) <= *x* & *0* <= *y* & *0* <= *z* & *x* + *y* + *z* <= *3* --> *x* * *y* + *x* * *z* + *y* * *z* >= *3* * *x* * *y* * *z*
by (*sos-cert* (((*A*<*0* * *R*<*1*) + (((*A*<*0* * *R*<*1*) * (*R*<*1/2* * [*1*] ^ *2*)) + (((*A*<=*2* * *R*<*1*) * (*R*<*1/2* * [~*1**x + *y*] ^ *2*)) + (((*A*<=*1* * *R*<*1*) * (*R*<*1/2* * [~*1**x + *z*] ^ *2*)) + (((*A*<=*1* * (*A*<=*2* * (*A*<=*3* * *R*<*1*))) * (*R*<*1/2* * [*1*] ^ *2*)) + (((*A*<=*0* * *R*<*1*) * (*R*<*1/2* * [~*1**y + *z*] ^ *2*)) + (((*A*<=*0* * (*A*<=*2* * (*A*<=*3* * *R*<*1*))) * (*R*<*1/2* * [*1*] ^ *2*)) + ((*A*<=*0* * (*A*<=*1* * (*A*<=*3* * *R*<*1*))) * (*R*<*1/2* * [*1*] ^ *2*)))))))))))

lemma ((*x*::*real*) ^ *2* + *y* ^ *2* + *z* ^ *2* = *1*) --> (*x* + *y* + *z*) ^ *2* <= *3*
by (*sos-cert* (((*A*<*0* * *R*<*1*) + (([~*3*] * *A*=*0*) + (*R*<*1* * ((*R*<*2* * [~*1/2**x + ~*1/2**y + *z*] ^ *2*) + (*R*<*3/2* * [~*1**x + *y*] ^ *2*)))))))

lemma (*w* ^ *2* + *x* ^ *2* + *y* ^ *2* + *z* ^ *2* = *1*) --> (*w* + *x* + *y* + *z*) ^ *2* <= (*4*::*real*)
by (*sos-cert* (((*A*<*0* * *R*<*1*) + (([~*4*] * *A*=*0*) + (*R*<*1* * ((*R*<*3* * [~*1/3**w + ~*1/3**x + ~*1/3**y + *z*] ^ *2*) + ((*R*<*8/3* * [~*1/2**w + ~*1/2**x + *y*] ^ *2*) + (*R*<*2* * [~*1**w + *x*] ^ *2*)))))))))

lemma (*x*::*real*) >= *1* & *y* >= *1* --> *x* * *y* >= *x* + *y* - *1*
by (*sos-cert* (((*A*<*0* * *R*<*1*) + ((*A*<=*0* * (*A*<=*1* * *R*<*1*)) * (*R*<*1* * [*1*] ^ *2*)))))

lemma (*x*::*real*) > *1* & *y* > *1* --> *x* * *y* > *x* + *y* - *1*
by (*sos-cert* ((((*A*<*0* * *A*<*1*) * *R*<*1*) + ((*A*<=*0* * *R*<*1*) * (*R*<*1* * [*1*] ^ *2*)))))

lemma *abs*(*x*) <= *1* --> *abs*(*64* * *x* ^ *7* - *112* * *x* ^ *5* + *56* * *x* ^ *3* - *7* * *x*) <= (*1*::*real*)
by (*sos-cert* ((((*A*<*0* * *R*<*1*) + ((*A*<=*1* * *R*<*1*) * (*R*<*1* * [~*8**x ^ *3* + ~*4**x ^ *2* + *4**x + *1*] ^ *2*)))) & ((((*A*<*0* * *A*<*1*) * *R*<*1*) + ((*A*<=*1* * (*A*<*0* * *R*<*1*)) * (*R*<*1* * [*8**x ^ *3* + ~*4**x ^ *2* + ~*4**x + *1*] ^ *2*))))))

lemma *2* <= *x* & *x* <= *125841* / *50000* & *2* <= *y* & *y* <= *125841* / *50000* & *2* <= *z* & *z* <= *125841* / *50000* --> *2* * (*x* * *z* + *x* * *y* + *y* * *z*) - (*x* * *x* + *y* * *y* + *z* * *z*) >= (*0*::*real*)
by (*sos-cert* (((*A*<*0* * *R*<*1*) + ((*R*<*1* * ((*R*<*5749028157/5000000000* * [~*25000/222477**x

+ ~25000/222477*y + ~25000/222477*z + 1]^2) + ((R<864067/1779816 * [419113/864067*x + 419113/864067*y + z]^2) + ((R<320795/864067 * [419113/1283180*x + y]^2) + (R<1702293/5132720 * [x]^2))))) + (((A<=4 * (A<=5 * R<1)) * (R<3/2 * [1]^2)) + (((A<=3 * (A<=5 * R<1)) * (R<1/2 * [1]^2)) + (((A<=2 * (A<=4 * R<1)) * (R<1 * [1]^2)) + (((A<=2 * (A<=3 * R<1)) * (R<3/2 * [1]^2)) + (((A<=1 * (A<=5 * R<1)) * (R<1/2 * [1]^2)) + (((A<=1 * (A<=3 * R<1)) * (R<1/2 * [1]^2)) + (((A<=0 * (A<=4 * R<1)) * (R<1 * [1]^2)) + (((A<=0 * (A<=2 * R<1)) * (R<1 * [1]^2)) + ((A<=0 * (A<=1 * R<1)) * (R<3/2 * [1]^2)))))))))))))

lemma (2::*real*) <= x & x <= 4 & 2 <= y & y <= 4 & 2 <= z & z <= 4 --> 0 <= 2 * (x * z + x * y + y * z) - (x * x + y * y + z * z)
by (*sos-cert* ((R<1 + ((R<1 * ((R<1 * [~1/6*x + ~1/6*y + ~1/6*z + 1]^2) + ((R<1/18 * [~1/2*x + ~1/2*y + z]^2) + (R<1/24 * [~1*x + y]^2)))) + (((A<0 * R<1) * (R<1/12 * [1]^2)) + (((A<=4 * (A<=5 * R<1)) * (R<1/6 * [1]^2)) + (((A<=2 * (A<=4 * R<1)) * (R<1/6 * [1]^2)) + (((A<=2 * (A<=3 * R<1)) * (R<1/6 * [1]^2)) + (((A<=0 * (A<=4 * R<1)) * (R<1/6 * [1]^2)) + (((A<=0 * (A<=2 * R<1)) * (R<1/6 * [1]^2)) + ((A<=0 * (A<=1 * R<1)) * (R<1/6 * [1]^2)))))))))))

lemma 2 <= (x::*real*) & x <= 4 & 2 <= y & y <= 4 & 2 <= z & z <= 4 --> 12 <= 2 * (x * z + x * y + y * z) - (x * x + y * y + z * z)
by (*sos-cert* (((A<0 * R<1) + (((A<=4 * R<1) * (R<2/3 * [1]^2)) + (((A<=4 * (A<=5 * R<1)) * (R<1 * [1]^2)) + (((A<=3 * (A<=4 * R<1)) * (R<1/3 * [1]^2)) + (((A<=2 * R<1) * (R<2/3 * [1]^2)) + (((A<=2 * (A<=5 * R<1)) * (R<1/3 * [1]^2)) + (((A<=2 * (A<=4 * R<1)) * (R<8/3 * [1]^2)) + (((A<=2 * (A<=3 * R<1)) * (R<1 * [1]^2)) + (((A<=1 * (A<=4 * R<1)) * (R<1/3 * [1]^2)) + (((A<=1 * (A<=2 * R<1)) * (R<1/3 * [1]^2)) + (((A<=0 * R<1) * (R<2/3 * [1]^2)) + (((A<=0 * (A<=5 * R<1)) * (R<1/3 * [1]^2)) + (((A<=0 * (A<=4 * R<1)) * (R<8/3 * [1]^2)) + (((A<=0 * (A<=3 * R<1)) * (R<1/3 * [1]^2)) + (((A<=0 * (A<=2 * R<1)) * (R<8/3 * [1]^2)) + ((A<=0 * (A<=1 * R<1)) * (R<1 * [1]^2)))))))))))))))))

lemma 0 <= (x::*real*) & 0 <= y & (x * y = 1) --> x + y <= x^2 + y^2
by (*sos-cert* (((A<0 * R<1) + (([1] * A=0) + (R<1 * ((R<1 * [~1/2*x + ~1/2*y + 1]^2) + (R<3/4 * [~1*x + y]^2)))))))

lemma *0 <= (x::real) & 0 <= y & (x * y = 1) --> x * y * (x + y) <= x^2 + y^2*
by (*sos-cert* (((*A<0 * R<1*) + (([~*1*x + ~*1*y + 1] * *A=0*) + (*R<1 * ((R<1 * [~*1/2*x + ~*1/2*y + 1]^2) + (R<3/4 * [~*1*x + y]^2)))))))

lemma *0 <= (x::real) & 0 <= y --> x * y * (x + y)^2 <= (x^2 + y^2)^2*
by (*sos-cert* (((*A<0 * R<1*) + (*R<1 * ((R<1 * [~*1/2*x^2 + y^2 + ~*1/2*x*y]^2) + (R<3/4 * [~*1*x^2 + x*y]^2))))))

lemma (*0::real*) *<= a & 0 <= b & 0 <= c & c * (2 * a + b)^3/ 27 <= x --> c * a^2 * b <= x*
by (*sos-cert* (((*A<0 * R<1*) + (((*A<=3 * R<1*) * (*R<1 * [1]^2*)) + (((*A<=1 * (A<=2 * R<1*)) * (*R<1/27 * [~*1*a + b]^2)) + ((*A<=0 * (A<=2 * R<1*)) * (*R<8/27 * [~*1*a + b]^2)))))))

lemma (*0::real*) *< x --> 0 < 1 + x + x^2*
by (*sos-cert* ((*R<1 + ((R<1 * (R<1 * [x]^2)) + (((A<0 * R<1*) * (*R<1 * [1]^2)) + ((A<=0 * R<1*) * (*R<1 * [1]^2)))))))

lemma (*0::real*) *<= x --> 0 < 1 + x + x^2*
by (*sos-cert* ((*R<1 + ((R<1 * (R<1 * [x]^2)) + (((A<=1 * R<1*) * (*R<1 * [1]^2)) + ((A<=0 * R<1*) * (*R<1 * [1]^2)))))))

lemma (*0::real*) *< 1 + x^2*
by (*sos-cert* ((*R<1 + ((R<1 * (R<1 * [x]^2)) + ((A<=0 * R<1*) * (*R<1 * [1]^2))))))

lemma (*0::real*) *<= 1 + 2 * x + x^2*
by (*sos-cert* (((*A<0 * R<1*) + (*R<1 * (R<1 * [x + 1]^2)))))

lemma (*0::real*) *< 1 + abs x*
by (*sos-cert* ((*R<1 + (((A<=1 * R<1*) * (*R<1/2 * [1]^2)) + ((A<=0 * R<1*) * (*R<1/2 * [1]^2))))))

lemma (*0::real*) *< 1 + (1 + x)^2 * (abs x)*
by (*sos-cert* (((*R<1 + (((A<=1 * R<1*) * (*R<1 * [1]^2)) + ((A<=0 * R<1*) * (*R<1 * [x + 1]^2))))) & ((*R<1 + (((A<0 * R<1*) * (*R<1 * [x + 1]^2)) + ((A<=0 * R<1*) * (*R<1 * [1]^2)))))))

lemma *abs ((1::real) + x^2) = (1::real) + x^2*
by (*sos-cert* (() & (((*R<1 + ((R<1 * (R<1 * [x]^2)) + ((A<1 * R<1*) * (*R<1/2 * [1]^2))))) & ((*R<1 + ((R<1 * (R<1 * [x]^2)) + ((A<0 * R<1*) * (*R<1 * [1]^2)))))))))
lemma (*3::real*) ** x + 7 * a < 4 ∧ 3 < 2 * x ⟶ a < 0*
by (*sos-cert* ((*R<1 + (((A<1 * R<1*) * (*R<2 * [1]^2)) + (((A<0 * R<1*) * (*R<3 * [1]^2)) + ((A<=0 * R<1*) * (*R<14 * [1]^2)))))))

lemma (*0*::*real*) < *x* −−> *1* < *y* −−> *y* * *x* <= *z* −−> *x* < *z*
by (*sos-cert* ((((*A*<*0* * *A*<*1*) * *R*<*1*) + (((*A*<=*1* * *R*<*1*) * (*R*<*1* * [*1*] ^ *2*)) + ((*A*<=*0* * *R*<*1*) * (*R*<*1* * [*1*] ^ *2*))))))
lemma (*1*::*real*) < *x* −−> *x* ^ *2* < *y* −−> *1* < *y*
by (*sos-cert* ((((*A*<*0* * *A*<*1*) * *R*<*1*) + ((*R*<*1* * ((*R*<*1*/*10* * [~ *2**x + *y* + *1*] ^ *2*) + (*R*<*1*/*10* * [~ *1**x + *y*] ^ *2*))) + (((*A*<*1* * *R*<*1*) * (*R*<*1*/*2* * [*1*] ^ *2*)) + (((*A*<*0* * *R*<*1*) * (*R*<*1* * [*x*] ^ *2*)) + (((*A*<=*0* * *R*<*1*) * ((*R*<*1*/*10* * [*x* + *1*] ^ *2*) + (*R*<*1*/*10* * [*x*] ^ *2*))) + (((*A*<=*0* * (*A*<*1* * *R*<*1*)) * (*R*<*1*/*5* * [*1*] ^ *2*)) + ((*A*<=*0* * (*A*<*0* * *R*<*1*)) * (*R*<*1*/*5* * [*1*] ^ *2*)))))))))
lemma (*b*::*real*) ^ *2* < *4* * *a* * *c* −−> ~(*a* * *x* ^ *2* + *b* * *x* + *c* = *0*)
by (*sos-cert* (((*A*<*0* * *R*<*1*) + (*R*<*1* * (*R*<*1* * [*2***a***x* + *b*] ^ *2*)))))
lemma (*b*::*real*) ^ *2* < *4* * *a* * *c* −−> ~(*a* * *x* ^ *2* + *b* * *x* + *c* = *0*)
by (*sos-cert* (((*A*<*0* * *R*<*1*) + (*R*<*1* * (*R*<*1* * [*2***a***x* + *b*] ^ *2*)))))
lemma ((*a*::*real*) * *x* ^ *2* + *b* * *x* + *c* = *0*) −−> *b* ^ *2* >= *4* * *a* * *c*
by (*sos-cert* (((*A*<*0* * *R*<*1*) + (*R*<*1* * (*R*<*1* * [*2***a***x* + *b*] ^ *2*)))))
lemma (*0*::*real*) <= *b* & *0* <= *c* & *0* <= *x* & *0* <= *y* & (*x* ^ *2* = *c*) & (*y* ^ *2* = *a* ^ *2* * *c* + *b*) −−> *a* * *c* <= *y* * *x*
by (*sos-cert* (((*A*<*0* * (*A*<*0* * *R*<*1*)) + (((*A*<=*2* * (*A*<=*3* * (*A*<*0* * *R*<*1*))) * (*R*<*2* * [*1*] ^ *2*)) + ((*A*<=*0* * (*A*<=*1* * *R*<*1*)) * (*R*<*1* * [*1*] ^ *2*))))))
lemma *abs*(*x* − *z*) <= *e* & *abs*(*y* − *z*) <= *e* & *0* <= *u* & *0* <= *v* & (*u* + *v* = *1*) −−> *abs*((*u* * *x* + *v* * *y*) − *z*) <= (*e*::*real*)
by (*sos-cert* ((((*A*<*0* * *R*<*1*) + (((*A*<=*3* * (*A*<=*6* * *R*<*1*)) * (*R*<*1* * [*1*] ^ *2*)) + ((*A*<=*1* * (*A*<=*5* * *R*<*1*)) * (*R*<*1* * [*1*] ^ *2*))))) & ((((*A*<*0* * *A*<*1*) * *R*<*1*) + (((*A*<=*3* * (*A*<=*5* * (*A*<*0* * *R*<*1*))) * (*R*<*1* * [*1*] ^ *2*)) + ((*A*<=*1* * (*A*<=*4* * (*A*<*0* * *R*<*1*))) * (*R*<*1* * [*1*] ^ *2*)))))))

lemma (*0*::*real*) <= *x* −−> (*1* + *x* + *x* ^ *2*)/(*1* + *x* ^ *2*) <= *1* + *x*
by (*sos-cert* (((((*A*<*0* * *A*<*1*) * *R*<*1*) + ((*A*<=*0* * (*A*<*0* * *R*<*1*)) * (*R*<*1* * [*x*] ^ *2*)))) & ((*R*<*1* + ((*R*<*1* * (*R*<*1* * [*x*] ^ *2*)) + ((*A*<*0* * *R*<*1*) * (*R*<*1* * [*1*] ^ *2*)))))))

lemma (*0*::*real*) <= *x* −−> *1* − *x* <= *1* / (*1* + *x* + *x* ^ *2*)
by (*sos-cert* (((*R*<*1* + (([~ *4*/*3*] * *A*=*0*) + ((*R*<*1* * ((*R*<*1*/*3* * [*3*/*2***x* + *1*] ^ *2*) + (*R*<*7*/*12* * [*x*] ^ *2*))) + ((*A*<=*0* * *R*<*1*) * (*R*<*1*/*3* * [*1*] ^ *2*)))))) & (((((*A*<*0* * *A*<*1*) * *R*<*1*) + ((*A*<=*0* * (*A*<*0* * *R*<*1*)) * (*R*<*1* * [*x*] ^ *2*)))) & ((*R*<*1* + ((*R*<*1* * (*R*<*1* * [*x*] ^ *2*)) + (((*A*<*0* * *R*<*1*) * (*R*<*1* * [*1*] ^ *2*)) + ((*A*<=*0* * *R*<*1*) * (*R*<*1* * [*1*] ^ *2*)))))))))

lemma (*x*::*real*) <= *1* / *2* −−> − *x* − *2* * *x* ^ *2* <= − *x* / (*1* − *x*)
by (*sos-cert* ((((*A*<*0* * *A*<*1*) * *R*<*1*) + ((*A*<=*0* * (*A*<*0* * *R*<*1*)) * (*R*<*1* * [*x*] ^ *2*)))))

lemma *4***r* ^ *2* = *p* ^ *2* − *4***q* & *r* >= (*0*::*real*) & *x* ^ *2* + *p***x* + *q* = *0* −−> *2**(*x*::*real*) = − *p* + *2***r* | *2***x* = −*p* − *2***r*
by (*sos-cert* ((((((*A*<*0* * *A*<*1*) * *R*<*1*) + ([~ *4*] * *A*=*0*))) & ((((*A*<*0* * *A*<*1*) * *R*<*1*) + ([*4*] * *A*=*0*)))) & (((((*A*<*0* * *A*<*1*) * *R*<*1*) + ([*4*] * *A*=*0*))) & ((((*A*<*0*

$* A{<}1) * R{<}1) + ([^\sim 4] * A{=}0))))))$

end

70 Transitive-Closure-Table: A tabled implementation of the reflexive transitive closure

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

inductive *rtrancl-path* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow bool$
  for $r :: 'a \Rightarrow 'a \Rightarrow bool$
where
  base: *rtrancl-path r x* [] *x*
| *step*: $r\ x\ y \Longrightarrow$ *rtrancl-path r y ys z* $\Longrightarrow$ *rtrancl-path r x* $(y \# ys)\ z$

lemma *rtranclp-eq-rtrancl-path*: $r^{**}\ x\ y = (\exists xs.$ *rtrancl-path r x xs y*$)$
proof
  assume $r^{**}\ x\ y$
  then show $\exists xs.$ *rtrancl-path r x xs y*
  proof (*induct rule*: *converse-rtranclp-induct*)
    case *base*
    have *rtrancl-path r y* [] *y* **by** (*rule rtrancl-path.base*)
    then show *?case* **..**
  next
    case (*step x z*)
    from ‹$\exists xs.$ *rtrancl-path r z xs y*›
    obtain *xs* **where** *rtrancl-path r z xs y* **..**
    with ‹*r x z*› **have** *rtrancl-path r x* $(z \# xs)\ y$
      by (*rule rtrancl-path.step*)
    then show *?case* **..**
  qed
next
  assume $\exists xs.$ *rtrancl-path r x xs y*
  then obtain *xs* **where** *rtrancl-path r x xs y* **..**
  then show $r^{**}\ x\ y$
  proof *induct*
    case (*base x*)
    show *?case* **by** (*rule rtranclp.rtrancl-refl*)
  next
    case (*step x y ys z*)
    from ‹*r x y*› ‹$r^{**}\ y\ z$› **show** *?case*
      by (*rule converse-rtranclp-into-rtranclp*)
  qed
qed

lemma *rtrancl-path-trans*:
assumes *xy*: *rtrancl-path r x xs y*
and *yz*: *rtrancl-path r y ys z*
shows *rtrancl-path r x* (*xs* @ *ys*) *z* **using** *xy yz*
proof (*induct arbitrary*: *z*)
case (*base x*)
then show *?case* **by** *simp*
next
case (*step x y xs*)
then have *rtrancl-path r y* (*xs* @ *ys*) *z*
by *simp*
with ‹*r x y*› **have** *rtrancl-path r x* (*y* # (*xs* @ *ys*)) *z*
by (*rule rtrancl-path.step*)
then show *?case* **by** *simp*
qed

lemma *rtrancl-path-appendE*:
assumes *xz*: *rtrancl-path r x* (*xs* @ *y* # *ys*) *z*
obtains *rtrancl-path r x* (*xs* @ [*y*]) *y* **and** *rtrancl-path r y ys z* **using** *xz*
proof (*induct xs arbitrary*: *x*)
case *Nil*
then have *rtrancl-path r x* (*y* # *ys*) *z* **by** *simp*
then obtain *xy*: *r x y* **and** *yz*: *rtrancl-path r y ys z*
by *cases auto*
from *xy* **have** *rtrancl-path r x* [*y*] *y*
by (*rule rtrancl-path.step* [*OF - rtrancl-path.base*])
then have *rtrancl-path r x* ([] @ [*y*]) *y* **by** *simp*
then show *?thesis* **using** *yz* **by** (*rule Nil*)
next
case (*Cons a as*)
then have *rtrancl-path r x* (*a* # (*as* @ *y* # *ys*)) *z* **by** *simp*
then obtain *xa*: *r x a* **and** *az*: *rtrancl-path r a* (*as* @ *y* # *ys*) *z*
by *cases auto*
show *?thesis*
proof (*rule Cons*(*1*) [*OF - az*])
assume *rtrancl-path r y ys z*
assume *rtrancl-path r a* (*as* @ [*y*]) *y*
with *xa* **have** *rtrancl-path r x* (*a* # (*as* @ [*y*])) *y*
by (*rule rtrancl-path.step*)
then have *rtrancl-path r x* ((*a* # *as*) @ [*y*]) *y*
by *simp*
then show *?thesis* **using** ‹*rtrancl-path r y ys z*›
by (*rule Cons*(*2*))
qed
qed

lemma *rtrancl-path-distinct*:
assumes *xy*: *rtrancl-path r x xs y*

```
  obtains xs' where rtrancl-path r x xs' y and distinct (x # xs') using xy
proof (induct xs rule: measure-induct-rule [of length])
  case (less xs)
  show ?case
  proof (cases distinct (x # xs))
    case True
    with ‹rtrancl-path r x xs y› show ?thesis by (rule less)
  next
    case False
    then have ∃ as bs cs a. x # xs = as @ [a] @ bs @ [a] @ cs
      by (rule not-distinct-decomp)
    then obtain as bs cs a where xxs: x # xs = as @ [a] @ bs @ [a] @ cs
      by iprover
    show ?thesis
    proof (cases as)
      case Nil
      with xxs have x: x = a and xs: xs = bs @ a # cs
        by auto
      from x xs ‹rtrancl-path r x xs y› have cs: rtrancl-path r x cs y
        by (auto elim: rtrancl-path-appendE)
      from xs have length cs < length xs by simp
      then show ?thesis
        by (rule less(1)) (iprover intro: cs less(2))+
    next
      case (Cons d ds)
      with xxs have xs: xs = ds @ a # (bs @ [a] @ cs)
        by auto
      with ‹rtrancl-path r x xs y› obtain xa: rtrancl-path r x (ds @ [a]) a
        and ay: rtrancl-path r a (bs @ a # cs) y
        by (auto elim: rtrancl-path-appendE)
      from ay have rtrancl-path r a cs y by (auto elim: rtrancl-path-appendE)
      with xa have xy: rtrancl-path r x ((ds @ [a]) @ cs) y
        by (rule rtrancl-path-trans)
      from xs have length ((ds @ [a]) @ cs) < length xs by simp
      then show ?thesis
        by (rule less(1)) (iprover intro: xy less(2))+
    qed
  qed
qed

inductive rtrancl-tab :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a ⇒ 'a ⇒ bool
  for r :: 'a ⇒ 'a ⇒ bool
where
  base: rtrancl-tab r xs x x
| step: x ∉ set xs ⟹ r x y ⟹ rtrancl-tab r (x # xs) y z ⟹ rtrancl-tab r xs x z

lemma rtrancl-path-imp-rtrancl-tab:
  assumes path: rtrancl-path r x xs y
  and x: distinct (x # xs)
```

and *ys*: $(\{x\} \cup set\ xs) \cap set\ ys = \{\}$
shows *rtrancl-tab r ys x y* **using** *path x ys*
proof (*induct arbitrary*: *ys*)
case *base*
show *?case* **by** (*rule rtrancl-tab.base*)
next
case (*step x y zs z*)
then have $x \notin set\ ys$ **by** *auto*
from *step* **have** *distinct* $(y \# zs)$ **by** *simp*
moreover from *step* **have** $(\{y\} \cup set\ zs) \cap set\ (x \# ys) = \{\}$
by *auto*
ultimately have *rtrancl-tab r* $(x \# ys)$ *y z*
by (*rule step*)
with ‹$x \notin set\ ys$› ‹*r x y*›
show *?case* **by** (*rule rtrancl-tab.step*)
qed

lemma *rtrancl-tab-imp-rtrancl-path*:
assumes *tab*: *rtrancl-tab r ys x y*
obtains *xs* **where** *rtrancl-path r x xs y* **using** *tab*
proof *induct*
case *base*
from *rtrancl-path.base* **show** *?case* **by** (*rule base*)
next
case *step* **show** *?case* **by** (*iprover intro*: *step rtrancl-path.step*)
qed

lemma *rtranclp-eq-rtrancl-tab-nil*: $r^{**}\ x\ y = rtrancl\text{-}tab\ r\ []\ x\ y$
proof
assume $r^{**}\ x\ y$
then obtain *xs* **where** *rtrancl-path r x xs y*
by (*auto simp add*: *rtranclp-eq-rtrancl-path*)
then obtain xs' **where** xs': *rtrancl-path r x xs' y*
and *distinct*: *distinct* $(x \# xs')$
by (*rule rtrancl-path-distinct*)
have $(\{x\} \cup set\ xs') \cap set\ [] = \{\}$ **by** *simp*
with xs' *distinct* **show** *rtrancl-tab r [] x y*
by (*rule rtrancl-path-imp-rtrancl-tab*)
next
assume *rtrancl-tab r [] x y*
then obtain *xs* **where** *rtrancl-path r x xs y*
by (*rule rtrancl-tab-imp-rtrancl-path*)
then show $r^{**}\ x\ y$
by (*auto simp add*: *rtranclp-eq-rtrancl-path*)
qed

declare *rtranclp-eq-rtrancl-tab-nil* [*code-unfold*, *code-inline del*]

declare *rtranclp-eq-rtrancl-tab-nil*[*THEN iffD2*, *code-pred-intro*]

code-pred *rtranclp* **using** *rtranclp-eq-rtrancl-tab-nil*[*THEN iffD1*] **by** *fastsimp*

70.1 A simple example

datatype *ty* = *A* | *B* | *C*

inductive *test* :: *ty* ⇒ *ty* ⇒ *bool*
where
  test A B
| *test B A*
| *test B C*

70.1.1 Invoking with the SML code generator

code-module *Test*
contains
test1 = *test*** *A C*
test2 = *test*** *C A*
test3 = *test*** *A* -
test4 = *test*** - *C*

ML *Test.test1*
ML *Test.test2*
ML *DSeq.list-of Test.test3*
ML *DSeq.list-of Test.test4*

70.1.2 Invoking with the predicate compiler and the generic code generator

code-pred *test* **.**

values {*x*. *test*** *A C*}
values {*x*. *test*** *C A*}
values {*x*. *test*** *A x*}
values {*x*. *test*** *x C*}

value *test*** *A C*
value *test*** *C A*

hide-type *ty*
hide-const *test A B C*

end

71 Univ-Poly: Univariate Polynomials

theory *Univ-Poly*
imports *Main*
begin

Application of polynomial as a function.

primrec (**in** *semiring-0*) *poly* :: *'a list => 'a => 'a* **where**
poly-Nil: *poly [] x = 0*
| *poly-Cons*: *poly (h#t) x = h + x * poly t x*

71.1 Arithmetic Operations on Polynomials

addition

primrec (**in** *semiring-0*) *padd* :: *'a list ⇒ 'a list ⇒ 'a list* (**infixl** *+++ 65*)
where
padd-Nil: *[] +++ l2 = l2*
| *padd-Cons*: *(h#t) +++ l2 = (if l2 = [] then h#t*
else (h + hd l2)#(t +++ tl l2))

Multiplication by a constant

primrec (**in** *semiring-0*) *cmult* :: *'a ⇒ 'a list ⇒ 'a list* (**infixl** *%** *70*) **where**
cmult-Nil: *c %* [] = []*
| *cmult-Cons*: *c %* (h#t) = (c * h)#(c %* t)*

Multiplication by a polynomial

primrec (**in** *semiring-0*) *pmult* :: *'a list ⇒ 'a list ⇒ 'a list* (**infixl** *** *70*)
where
pmult-Nil: *[] *** l2 = []*
| *pmult-Cons*: *(h#t) *** l2 = (if t = [] then h %* l2*
else (h % l2) +++ ((0) # (t *** l2)))*

Repeated multiplication by a polynomial

primrec (**in** *semiring-0*) *mulexp* :: *nat ⇒ 'a list ⇒ 'a list ⇒ 'a list* **where**
mulexp-zero: *mulexp 0 p q = q*
| *mulexp-Suc*: *mulexp (Suc n) p q = p *** mulexp n p q*

Exponential

primrec (**in** *semiring-1*) *pexp* :: *'a list ⇒ nat ⇒ 'a list* (**infixl** *%^ 80*) **where**
pexp-0: *p %^ 0 = [1]*
| *pexp-Suc*: *p %^ (Suc n) = p *** (p %^ n)*

Quotient related value of dividing a polynomial by x + a

primrec (**in** *field*) *pquot* :: *'a list ⇒ 'a ⇒ 'a list* **where**
pquot-Nil: *pquot [] a= []*
| *pquot-Cons*: *pquot (h#t) a = (if t = [] then [h]*
*else (inverse(a) * (h − hd(pquot t a)))#(pquot t a))*

normalization of polynomials (remove extra 0 coeff)

primrec (**in** *semiring-0*) *pnormalize* :: *'a list* ⇒ *'a list* **where**
  pnormalize-Nil: *pnormalize* [] = []
| *pnormalize-Cons*: *pnormalize* (*h*#*p*) = (*if* ((*pnormalize p*) = [])
                                then (*if* (*h* = *0*) *then* [] *else* [*h*])
                                else (*h*#(*pnormalize p*)))

definition (**in** *semiring-0*) *pnormal p* = ((*pnormalize p* = *p*) ∧ *p* ≠ [])
definition (**in** *semiring-0*) *nonconstant p* = (*pnormal p* ∧ (∀ *x*. *p* ≠ [*x*]))

Other definitions

definition (**in** *ring-1*)
  poly-minus :: *'a list* => *'a list* (−− - [*80*] *80*) **where**
  −− *p* = (− *1*) %* *p*

definition (**in** *semiring-0*)
  divides :: *'a list* ⇒ *'a list* ⇒ *bool* (**infixl** *divides 70*) **where**
  [*code del*]: *p1 divides p2* = (∃ *q*. *poly p2* = *poly*(*p1* *** *q*))

— order of a polynomial
definition (**in** *ring-1*) *order* :: *'a* => *'a list* => *nat* **where**
  order a p = (*SOME n*. ([−*a*, *1*] %^ *n*) *divides p* &
                ~ (([−*a*, *1*] %^ (*Suc n*)) *divides p*))

— degree of a polynomial
definition (**in** *semiring-0*) *degree* :: *'a list* => *nat* **where**
  degree p = *length* (*pnormalize p*) − *1*

— squarefree polynomials — NB with respect to real roots only.
definition (**in** *ring-1*)
  rsquarefree :: *'a list* => *bool* **where**
  rsquarefree p = (*poly p* ≠ *poly* [] &
                (∀ *a*. (*order a p* = *0*) | (*order a p* = *1*)))

context *semiring-0*
begin

lemma *padd-Nil2*[*simp*]: *p* +++ [] = *p*
by (*induct p*) *auto*

lemma *padd-Cons-Cons*: (*h1* # *p1*) +++ (*h2* # *p2*) = (*h1* + *h2*) # (*p1* +++ *p2*)
by *auto*

lemma *pminus-Nil*[*simp*]: −− [] = []
by (*simp add*: *poly-minus-def*)

lemma *pmult-singleton*: [*h1*] *** *p1* = *h1* %* *p1* **by** *simp*
end

lemma (**in** *semiring-1*) *poly-ident-mult*[*simp*]: *1* %* *t* = *t* **by** (*induct t*, *auto*)

lemma (**in** *semiring-0*) *poly-simple-add-Cons*[*simp*]: [*a*] +++ ((*0*)#*t*) = (*a*#*t*)
by *simp*

Handy general properties

lemma (**in** *comm-semiring-0*) *padd-commut*: *b* +++ *a* = *a* +++ *b*
proof(*induct b arbitrary*: *a*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons b bs a*) **thus** *?case* **by** (*cases a*, *simp-all add*: *add-commute*)
qed

lemma (**in** *comm-semiring-0*) *padd-assoc*: ∀ *b c*. (*a* +++ *b*) +++ *c* = *a* +++ (*b* +++ *c*)
apply (*induct a arbitrary*: *b c*)
apply (*simp*, *clarify*)
apply (*case-tac b*, *simp-all add*: *add-ac*)
done

lemma (**in** *semiring-0*) *poly-cmult-distr*: *a* %* (*p* +++ *q*) = (*a* %* *p* +++ *a* %* *q*)
apply (*induct p arbitrary*: *q*,*simp*)
apply (*case-tac q*, *simp-all add*: *right-distrib*)
done

lemma (**in** *ring-1*) *pmult-by-x*[*simp*]: [*0*, *1*] *** *t* = ((*0*)#*t*)
apply (*induct t*, *simp*)
apply (*auto simp add*: *mult-zero-left poly-ident-mult padd-commut*)
apply (*case-tac t*, *auto*)
done

properties of evaluation of polynomials.

lemma (**in** *semiring-0*) *poly-add*: *poly* (*p1* +++ *p2*) *x* = *poly p1 x* + *poly p2 x*
proof(*induct p1 arbitrary*: *p2*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a as p2*) **thus** *?case*
by (*cases p2*, *simp-all add*: *add-ac right-distrib*)
qed

lemma (**in** *comm-semiring-0*) *poly-cmult*: *poly* (*c* %* *p*) *x* = *c* * *poly p x*
apply (*induct p*)
apply (*case-tac* [*2*] *x*=*zero*)
apply (*auto simp add*: *right-distrib mult-ac*)
done

lemma (**in** *comm-semiring-0*) *poly-cmult-map*: *poly* (*map* (*op* * *c*) *p*) *x* = *c*∗*poly p x*

by (*induct p*, *auto simp add*: *right-distrib mult-ac*)

lemma (**in** *comm-ring-1*) *poly-minus*: *poly* (−− *p*) *x* = − (*poly p x*)
apply (*simp add*: *poly-minus-def*)
apply (*auto simp add*: *poly-cmult minus-mult-left*[*symmetric*])
done

lemma (**in** *comm-semiring-0*) *poly-mult*: *poly* (*p1* *** *p2*) *x* = *poly p1 x* * *poly p2 x*
proof(*induct p1 arbitrary*: *p2*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a as p2*)
thus *?case* **by** (*cases as*,
simp-all add: *poly-cmult poly-add left-distrib right-distrib mult-ac*)
qed

class *idom-char-0* = *idom* + *ring-char-0*

lemma (**in** *comm-ring-1*) *poly-exp*: *poly* (*p* %^ *n*) *x* = (*poly p x*) ^ *n*
apply (*induct n*)
apply (*auto simp add*: *poly-cmult poly-mult power-Suc*)
done

More Polynomial Evaluation Lemmas

lemma (**in** *semiring-0*) *poly-add-rzero*[*simp*]: *poly* (*a* +++ []) *x* = *poly a x*
by *simp*

lemma (**in** *comm-semiring-0*) *poly-mult-assoc*: *poly* ((*a* *** *b*) *** *c*) *x* = *poly* (*a* *** (*b* *** *c*)) *x*
by (*simp add*: *poly-mult mult-assoc*)

lemma (**in** *semiring-0*) *poly-mult-Nil2*[*simp*]: *poly* (*p* *** []) *x* = *0*
by (*induct p*, *auto*)

lemma (**in** *comm-semiring-1*) *poly-exp-add*: *poly* (*p* %^ (*n* + *d*)) *x* = *poly*(*p* %^ *n* *** *p* %^ *d*) *x*
apply (*induct n*)
apply (*auto simp add*: *poly-mult mult-assoc*)
done

71.2 Key Property: if *f a* = (*0*::'*a*) then *x* − *a* divides *p x*

lemma (**in** *comm-ring-1*) *lemma-poly-linear-rem*: ∀ *h*. ∃ *q r*. *h*#*t* = [*r*] +++ [−*a*, *1*] *** *q*
proof(*induct t*)
case *Nil*
{fix *h* **have** [*h*] = [*h*] +++ [− *a*, *1*] *** [] **by** *simp***}**
thus *?case* **by** *blast*
next

```
  case (Cons  x xs)
  {fix h
    from Cons.hyps[rule-format, of x]
    obtain q r where qr: x#xs = [r] +++ [- a, 1] *** q by blast
    have h#x#xs = [a*r + h] +++ [-a, 1] *** (r#q)
      using qr by(cases q, simp-all add: algebra-simps diff-def[symmetric]
        minus-mult-left[symmetric] right-minus)
    hence ∃ q r. h#x#xs = [r] +++ [-a, 1] *** q by blast}
  thus ?case by blast
qed

lemma (in comm-ring-1) poly-linear-rem: ∃ q r. h#t = [r] +++ [-a, 1] *** q
by (cut-tac t = t and a = a in lemma-poly-linear-rem, auto)


lemma (in comm-ring-1) poly-linear-divides: (poly p a = 0) = ((p = []) | (∃ q. p
= [-a, 1] *** q))
proof-
  {assume p: p = [] hence ?thesis by simp}
  moreover
  {fix x xs assume p: p = x#xs
    {fix q assume p = [-a, 1] *** q hence poly p a = 0
       by (simp add: poly-add poly-cmult minus-mult-left[symmetric])}
    moreover
    {assume p0: poly p a = 0
      from poly-linear-rem[of x xs a] obtain q r
      where qr: x#xs = [r] +++ [- a, 1] *** q by blast
      have r = 0 using p0 by (simp only: p qr poly-mult poly-add) simp
      hence ∃ q. p = [- a, 1] *** q using p qr  apply - apply (rule exI[where
x=q])apply auto apply (cases q) apply auto done}
    ultimately have ?thesis using p by blast}
  ultimately show ?thesis by (cases p, auto)
qed

lemma (in semiring-0) lemma-poly-length-mult[simp]: ∀ h k a. length (k %* p
+++  (h # (a %* p))) = Suc (length p)
by (induct p, auto)

lemma (in semiring-0) lemma-poly-length-mult2[simp]: ∀ h k. length (k %* p
+++  (h # p)) = Suc (length p)
by (induct p, auto)

lemma (in ring-1) poly-length-mult[simp]: length([-a,1] *** q) = Suc (length q)
by auto
```

71.3 Polynomial length

```
lemma (in semiring-0) poly-cmult-length[simp]: length (a %* p) = length p
by (induct p, auto)
```

lemma (**in** *semiring-0*) *poly-add-length*: *length* (*p1* +++ *p2*) = *max* (*length p1*) (*length p2*)
apply (*induct p1 arbitrary*: *p2*, *simp-all*)
apply *arith*
done

lemma (**in** *semiring-0*) *poly-root-mult-length*[*simp*]: *length*([*a*,*b*] *** *p*) = *Suc* (*length p*)
by (*simp add*: *poly-add-length*)

lemma (**in** *idom*) *poly-mult-not-eq-poly-Nil*[*simp*]:
poly (*p* *** *q*) *x* ≠ *poly* [] *x* ⟷ *poly p x* ≠ *poly* [] *x* ∧ *poly q x* ≠ *poly* [] *x*
by (*auto simp add*: *poly-mult*)

lemma (**in** *idom*) *poly-mult-eq-zero-disj*: *poly* (*p* *** *q*) *x* = *0* ⟷ *poly p x* = *0* ∨ *poly q x* = *0*
by (*auto simp add*: *poly-mult*)

Normalisation Properties

lemma (**in** *semiring-0*) *poly-normalized-nil*: (*pnormalize p* = []) −−> (*poly p x* = *0*)
by (*induct p*, *auto*)

A nontrivial polynomial of degree n has no more than n roots

lemma (**in** *idom*) *poly-roots-index-lemma*:
assumes *p*: *poly p x* ≠ *poly* [] *x* **and** *n*: *length p* = *n*
shows ∃ *i*. ∀ *x*. *poly p x* = *0* ⟶ (∃ *m*≤*n*. *x* = *i m*)
using *p n*
proof(*induct n arbitrary*: *p x*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n p x*)
{assume *C*: ⋀*i*. ∃ *x*. *poly p x* = *0* ∧ (∀ *m*≤*Suc n*. *x* ≠ *i m*)
from *Suc.prems* **have** *p0*: *poly p x* ≠ *0* *p*≠ [] **by** *auto*
from *p0*(*1*)[*unfolded poly-linear-divides*[*of p x*]]
have ∀ *q*. *p* ≠ [− *x*, *1*] *** *q* **by** *blast*
from *C* **obtain** *a* **where** *a*: *poly p a* = *0* **by** *blast*
from *a*[*unfolded poly-linear-divides*[*of p a*]] *p0*(*2*)
obtain *q* **where** *q*: *p* = [−*a*, *1*] *** *q* **by** *blast*
have *lg*: *length q* = *n* **using** *q Suc.prems*(*2*) **by** *simp*
from *q p0* **have** *qx*: *poly q x* ≠ *poly* [] *x*
by (*auto simp add*: *poly-mult poly-add poly-cmult*)
from *Suc.hyps*[*OF qx lg*] **obtain** *i* **where**
i: ∀ *x*. *poly q x* = *0* ⟶ (∃ *m*≤*n*. *x* = *i m*) **by** *blast*
let *?i* = λ*m*. *if m* = *Suc n then a else i m*
from *C*[*of ?i*] **obtain** *y* **where** *y*: *poly p y* = *0* ∀ *m*≤ *Suc n*. *y* ≠ *?i m*
by *blast*
from *y* **have** *y* = *a* ∨ *poly q y* = *0*

```
    by (simp only: q poly-mult-eq-zero-disj poly-add) (simp add: algebra-simps)
   with i[rule-format, of y] y(1) y(2) have False apply auto
    apply (erule-tac x=m in allE)
    apply auto
    done}
 thus ?case by blast
qed


lemma (in idom) poly-roots-index-length: poly p x ≠ poly [] x ==>
     ∃ i. ∀ x. (poly p x = 0) --> (∃ n. n ≤ length p & x = i n)
by (blast intro: poly-roots-index-lemma)

lemma (in idom) poly-roots-finite-lemma1: poly p x ≠ poly [] x ==>
     ∃ N i. ∀ x. (poly p x = 0) --> (∃ n. (n::nat) < N & x = i n)
apply (drule poly-roots-index-length, safe)
apply (rule-tac x = Suc (length p) in exI)
apply (rule-tac x = i in exI)
apply (simp add: less-Suc-eq-le)
done


lemma (in idom) idom-finite-lemma:
 assumes P: ∀ x. P x --> (∃ n. n < length j & x = j!n)
 shows finite {x. P x}
proof-
 let ?M = {x. P x}
 let ?N = set j
 have ?M ⊆ ?N using P by auto
 thus ?thesis using finite-subset by auto
qed


lemma (in idom) poly-roots-finite-lemma2: poly p x ≠ poly [] x ==>
     ∃ i. ∀ x. (poly p x = 0) --> x ∈ set i
apply (drule poly-roots-index-length, safe)
apply (rule-tac x=map (λn. i n) [0 ..< Suc (length p)] in exI)
apply (auto simp add: image-iff)
apply (erule-tac x=x in allE, clarsimp)
by (case-tac n=length p, auto simp add: order-le-less)

lemma (in ring-char-0) UNIV-ring-char-0-infinte:
 ¬ (finite (UNIV:: 'a set))
proof
 assume F: finite (UNIV :: 'a set)
 have finite (UNIV :: nat set)
 proof (rule finite-imageD)
  have of-nat ` UNIV ⊆ UNIV by simp
  then show finite (of-nat ` UNIV :: 'a set) using F by (rule finite-subset)
```

show *inj* (*of-nat* :: *nat* ⇒ *'a*) **by** (*simp add*: *inj-on-def*)
qed
with *infinite-UNIV-nat* **show** *False* **..**
qed

lemma (**in** *idom-char-0*) *poly-roots-finite*: (*poly p* ≠ *poly* []) =
finite {*x*. *poly p x* = *0*}
proof
assume *H*: *poly p* ≠ *poly* []
show *finite* {*x*. *poly p x* = (*0*::*'a*)}
using *H*
apply −
apply (*erule contrapos-np*, *rule ext*)
apply (*rule ccontr*)
apply (*clarify dest*!: *poly-roots-finite-lemma2*)
using *finite-subset*
proof−
fix *x i*
assume *F*: ¬ *finite* {*x*. *poly p x* = (*0*::*'a*)}
and *P*: ∀ *x*. *poly p x* = (*0*::*'a*) ⟶ *x* ∈ *set i*
let *?M*= {*x*. *poly p x* = (*0*::*'a*)}
from *P* **have** *?M* ⊆ *set i* **by** *auto*
with *finite-subset F* **show** *False* **by** *auto*
qed
next
assume *F*: *finite* {*x*. *poly p x* = (*0*::*'a*)}
show *poly p* ≠ *poly* [] **using** *F UNIV-ring-char-0-infinte* **by** *auto*
qed

Entirety and Cancellation for polynomials

lemma (**in** *idom-char-0*) *poly-entire-lemma2*:
assumes *p0*: *poly p* ≠ *poly* [] **and** *q0*: *poly q* ≠ *poly* []
shows *poly* (*p*∗∗∗*q*) ≠ *poly* []
proof−
let *?S* = λ*p*. {*x*. *poly p x* = *0*}
have *?S* (*p* ∗∗∗ *q*) = *?S p* ∪ *?S q* **by** (*auto simp add*: *poly-mult*)
with *p0 q0* **show** *?thesis* **unfolding** *poly-roots-finite* **by** *auto*
qed

lemma (**in** *idom-char-0*) *poly-entire*:
poly (*p* ∗∗∗ *q*) = *poly* [] ⟷ *poly p* = *poly* [] ∨ *poly q* = *poly* []
using *poly-entire-lemma2*[*of p q*]
by (*auto simp add*: *expand-fun-eq poly-mult*)

lemma (**in** *idom-char-0*) *poly-entire-neg*: (*poly* (*p* ∗∗∗ *q*) ≠ *poly* []) = ((*poly p* ≠ *poly* []) & (*poly q* ≠ *poly* []))
by (*simp add*: *poly-entire*)

lemma *fun-eq*: (*f* = *g*) = (∀ *x*. *f x* = *g x*)

by (*auto intro*!: *ext*)

lemma (**in** *comm-ring-1*) *poly-add-minus-zero-iff*: (*poly* (*p* +++ −− *q*) = *poly* []) = (*poly p* = *poly q*)
by (*auto simp add*: *algebra-simps poly-add poly-minus-def fun-eq poly-cmult minus-mult-left*[*symmetric*])

lemma (**in** *comm-ring-1*) *poly-add-minus-mult-eq*: *poly* (*p* *** *q* +++ −−(*p* *** *r*)) = *poly* (*p* *** (*q* +++ −− *r*))
by (*auto simp add*: *poly-add poly-minus-def fun-eq poly-mult poly-cmult right-distrib minus-mult-left*[*symmetric*] *minus-mult-right*[*symmetric*])

subclass (**in** *idom-char-0*) *comm-ring-1* **..**
lemma (**in** *idom-char-0*) *poly-mult-left-cancel*: (*poly* (*p* *** *q*) = *poly* (*p* *** *r*)) = (*poly p* = *poly* [] | *poly q* = *poly r*)
proof−
  have *poly* (*p* *** *q*) = *poly* (*p* *** *r*) ⟷ *poly* (*p* *** *q* +++ −− (*p* *** *r*)) = *poly* [] **by** (*simp only*: *poly-add-minus-zero-iff*)
  also have . . . ⟷ *poly p* = *poly* [] | *poly q* = *poly r*
   by (*auto intro*: *ext simp add*: *poly-add-minus-mult-eq poly-entire poly-add-minus-zero-iff*)
  finally show *?thesis* **.**
qed

lemma (**in** *idom*) *poly-exp-eq-zero*[*simp*]:
    (*poly* (*p* %^ *n*) = *poly* []) = (*poly p* = *poly* [] & *n* ≠ *0*)
apply (*simp only*: *fun-eq add*: *all-simps* [*symmetric*])
apply (*rule arg-cong* [**where** *f* = *All*])
apply (*rule ext*)
apply (*induct n*)
apply (*auto simp add*: *poly-exp poly-mult*)
done

lemma (**in** *semiring-1*) *one-neq-zero*[*simp*]: *1* ≠ *0* **using** *zero-neq-one* **by** *blast*
lemma (**in** *comm-ring-1*) *poly-prime-eq-zero*[*simp*]: *poly* [*a*,*1*] ≠ *poly* []
apply (*simp add*: *fun-eq*)
apply (*rule-tac x* = *minus one a* **in** *exI*)
apply (*unfold diff-minus*)
apply (*subst add-commute*)
apply (*subst add-assoc*)
apply *simp*
done

lemma (**in** *idom*) *poly-exp-prime-eq-zero*: (*poly* ([*a*, *1*] %^ *n*) ≠ *poly* [])
by *auto*

A more constructive notion of polynomials being trivial

lemma (**in** *idom-char-0*) *poly-zero-lemma'*: *poly* (*h* # *t*) = *poly* [] ==> *h* = *0* & *poly t* = *poly* []
apply(*simp add*: *fun-eq*)
apply (*case-tac h* = *zero*)

apply (*drule-tac* [*2*] *x* = *zero* **in** *spec*, *auto*)
apply (*cases poly t* = *poly* [], *simp*)
proof−
fix *x*
assume *H*: ∀ *x*. *x* = (*0*::'*a*) ∨ *poly t x* = (*0*::'*a*) **and** *pnz*: *poly t* ≠ *poly* []
let *?S* = {*x*. *poly t x* = *0*}
from *H* **have** ∀ *x*. *x* ≠*0* ⟶ *poly t x* = *0* **by** *blast*
hence *th*: *?S* ⊇ *UNIV* − {*0*} **by** *auto*
from *poly-roots-finite pnz* **have** *th'*: *finite ?S* **by** *blast*
from *finite-subset*[*OF th th'*] *UNIV-ring-char-0-infinte*
show *poly t x* = (*0*::'*a*) **by** *simp*
qed

lemma (**in** *idom-char-0*) *poly-zero*: (*poly p* = *poly* []) = *list-all* (%*c*. *c* = *0*) *p*
apply (*induct p*, *simp*)
apply (*rule iffI*)
apply (*drule poly-zero-lemma'*, *auto*)
done

lemma (**in** *idom-char-0*) *poly-0*: *list-all* (λ*c*. *c* = *0*) *p* ⟹ *poly p x* = *0*
unfolding *poly-zero*[*symmetric*] **by** *simp*

Basics of divisibility.

lemma (**in** *idom*) *poly-primes*: ([*a*, *1*] *divides* (*p* ∗∗∗ *q*)) = ([*a*, *1*] *divides p* | [*a*, *1*] *divides q*)
apply (*auto simp add*: *divides-def fun-eq poly-mult poly-add poly-cmult left-distrib* [*symmetric*])
apply (*drule-tac x* = *uminus a* **in** *spec*)
apply (*simp add*: *poly-linear-divides poly-add poly-cmult left-distrib* [*symmetric*])
apply (*cases p* = [])
apply (*rule exI*[**where** *x*=[]])
apply *simp*
apply (*cases q* = [])
apply (*erule allE*[**where** *x*=[]], *simp*)

apply *clarsimp*
apply (*cases* ∃ *q*::'*a list*. *p* = *a* %∗ *q* +++ ((*0*::'*a*) # *q*))
apply (*clarsimp simp add*: *poly-add poly-cmult*)
apply (*rule-tac x*=*qa* **in** *exI*)
apply (*simp add*: *left-distrib* [*symmetric*])
apply *clarsimp*

apply (*auto simp add*: *right-minus poly-linear-divides poly-add poly-cmult left-distrib* [*symmetric*])
apply (*rule-tac x* = *pmult qa q* **in** *exI*)
apply (*rule-tac* [*2*] *x* = *pmult p qa* **in** *exI*)
apply (*auto simp add*: *poly-add poly-mult poly-cmult mult-ac*)
done

lemma (**in** *comm-semiring-1*) *poly-divides-refl*[*simp*]: *p divides p*
apply (*simp add*: *divides-def*)
apply (*rule-tac x* = [*one*] **in** *exI*)
apply (*auto simp add*: *poly-mult fun-eq*)
done

lemma (**in** *comm-semiring-1*) *poly-divides-trans*: [| *p divides q*; *q divides r* |] ==> *p divides r*
apply (*simp add*: *divides-def*, *safe*)
apply (*rule-tac x* = *pmult qa qaa* **in** *exI*)
apply (*auto simp add*: *poly-mult fun-eq mult-assoc*)
done

lemma (**in** *comm-semiring-1*) *poly-divides-exp*: *m* ≤ *n* ==> (*p* % ^ *m*) *divides* (*p* % ^ *n*)
apply (*auto simp add*: *le-iff-add*)
apply (*induct-tac k*)
apply (*rule-tac* [*2*] *poly-divides-trans*)
apply (*auto simp add*: *divides-def*)
apply (*rule-tac x* = *p* **in** *exI*)
apply (*auto simp add*: *poly-mult fun-eq mult-ac*)
done

lemma (**in** *comm-semiring-1*) *poly-exp-divides*: [| (*p* % ^ *n*) *divides q*; *m*≤*n* |] ==> (*p* % ^ *m*) *divides q*
by (*blast intro*: *poly-divides-exp poly-divides-trans*)

lemma (**in** *comm-semiring-0*) *poly-divides-add*:
[| *p divides q*; *p divides r* |] ==> *p divides* (*q* +++ *r*)
apply (*simp add*: *divides-def*, *auto*)
apply (*rule-tac x* = *padd qa qaa* **in** *exI*)
apply (*auto simp add*: *poly-add fun-eq poly-mult right-distrib*)
done

lemma (**in** *comm-ring-1*) *poly-divides-diff*:
[| *p divides q*; *p divides* (*q* +++ *r*) |] ==> *p divides r*
apply (*simp add*: *divides-def*, *auto*)
apply (*rule-tac x* = *padd qaa* (*poly-minus qa*) **in** *exI*)
apply (*auto simp add*: *poly-add fun-eq poly-mult poly-minus algebra-simps*)
done

lemma (**in** *comm-ring-1*) *poly-divides-diff2*: [| *p divides r*; *p divides* (*q* +++ *r*) |] ==> *p divides q*
apply (*erule poly-divides-diff*)
apply (*auto simp add*: *poly-add fun-eq poly-mult divides-def add-ac*)
done

lemma (**in** *semiring-0*) *poly-divides-zero*: *poly p* = *poly* [] ==> *q divides p*

apply (*simp add*: *divides-def*)
apply (*rule exI*[**where** *x*=[]])
apply (*auto simp add*: *fun-eq poly-mult*)
done

lemma (**in** *semiring-0*) *poly-divides-zero2*[*simp*]: *q divides* []
apply (*simp add*: *divides-def*)
apply (*rule-tac x* = [] **in** *exI*)
apply (*auto simp add*: *fun-eq*)
done

At last, we can consider the order of a root.

lemma (**in** *idom-char-0*) *poly-order-exists-lemma*:
assumes *lp*: *length p* = *d* **and** *p*: *poly p* ≠ *poly* []
shows ∃ *n q*. *p* = *mulexp n* [−*a*, *1*] *q* ∧ *poly q a* ≠ *0*
using *lp p*
proof(*induct d arbitrary*: *p*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n p*)
{assume *p0*: *poly p a* = *0*
from *Suc.prems* **have** *h*: *length p* = *Suc n poly p* ≠ *poly* [] **by** *auto*
hence *pN*: *p* ≠ [] **by** *auto*
from *p0*[*unfolded poly-linear-divides*] *pN* **obtain** *q* **where**
q: *p* = [−*a*, *1*] *** *q* **by** *blast*
from *q h p0* **have** *qh*: *length q* = *n poly q* ≠ *poly* []
apply −
apply *simp*
apply (*simp only*: *fun-eq*)
apply (*rule ccontr*)
apply (*simp add*: *fun-eq poly-add poly-cmult minus-mult-left*[*symmetric*])
done
from *Suc.hyps*[*OF qh*] **obtain** *m r* **where**
mr: *q* = *mulexp m* [−*a*,*1*] *r poly r a* ≠ *0* **by** *blast*
from *mr q* **have** *p* = *mulexp* (*Suc m*) [−*a*,*1*] *r* ∧ *poly r a* ≠ *0* **by** *simp*
hence *?case* **by** *blast***}**
moreover
{assume *p0*: *poly p a* ≠ *0*
hence *?case* **using** *Suc.prems* **apply** *simp* **by** (*rule exI*[**where** *x*=*0*::*nat*], *simp*)**}**
ultimately show *?case* **by** *blast*
qed

lemma (**in** *comm-semiring-1*) *poly-mulexp*: *poly* (*mulexp n p q*) *x* = (*poly p x*) ^ *n* ∗ *poly q x*
by(*induct n*, *auto simp add*: *poly-mult power-Suc mult-ac*)

lemma (**in** *comm-semiring-1*) *divides-left-mult*:

assumes *d*:(*p*∗∗∗*q*) *divides r* **shows** *p divides r* ∧ *q divides r*
proof−
from *d* **obtain** *t* **where** *r*:*poly r* = *poly* (*p*∗∗∗*q* ∗∗∗ *t*)
unfolding *divides-def* **by** *blast*
hence *poly r* = *poly* (*p* ∗∗∗ (*q* ∗∗∗ *t*))
poly r = *poly* (*q* ∗∗∗ (*p*∗∗∗*t*)) **by**(*auto simp add*: *fun-eq poly-mult mult-ac*)
thus *?thesis* **unfolding** *divides-def* **by** *blast*
qed

lemma (**in** *semiring-1*)
zero-power-iff: *0* ^ *n* = (*if n* = *0 then 1 else 0*)
by (*induct n*, *simp-all add*: *power-Suc*)

lemma (**in** *idom-char-0*) *poly-order-exists*:
assumes *lp*: *length p* = *d* **and** *p0*: *poly p* ≠ *poly* []
shows ∃ *n*. ([−*a*, *1*] %^ *n*) *divides p* & ~(([−*a*, *1*] %^ (*Suc n*)) *divides p*)
proof−
let *?poly* = *poly*
let *?mulexp* = *mulexp*
let *?pexp* = *pexp*
from *lp p0*
show *?thesis*
apply −
apply (*drule poly-order-exists-lemma* [**where** *a*=*a*], *assumption*, *clarify*)
apply (*rule-tac x* = *n* **in** *exI*, *safe*)
apply (*unfold divides-def*)
apply (*rule-tac x* = *q* **in** *exI*)
apply (*induct-tac n*, *simp*)
apply (*simp* (*no-asm-simp*) *add*: *poly-add poly-cmult poly-mult right-distrib mult-ac*)
apply *safe*
apply (*subgoal-tac ?poly* (*?mulexp n* [*uminus a*, *one*] *q*) ≠ *?poly* (*pmult* (*?pexp* [*uminus a*, *one*] (*Suc n*)) *qa*))
apply *simp*
apply (*induct-tac n*)
apply (*simp del*: *pmult-Cons pexp-Suc*)
apply (*erule-tac Q* = *?poly q a* = *zero* **in** *contrapos-np*)
apply (*simp add*: *poly-add poly-cmult minus-mult-left*[*symmetric*])
apply (*rule pexp-Suc* [*THEN ssubst*])
apply (*rule ccontr*)
apply (*simp add*: *poly-mult-left-cancel poly-mult-assoc del*: *pmult-Cons pexp-Suc*)
done
qed

lemma (**in** *semiring-1*) *poly-one-divides*[*simp*]: [*1*] *divides p*

by (*simp add*: *divides-def*, *auto*)

lemma (**in** *idom-char-0*) *poly-order*: *poly p* ≠ *poly* []
    ==> *EX*! *n*. ([−*a*, *1*] % ^ *n*) *divides p* &
            ~(([−*a*, *1*] % ^ (*Suc n*)) *divides p*)
apply (*auto intro*: *poly-order-exists simp add*: *less-linear simp del*: *pmult-Cons pexp-Suc*)
apply (*cut-tac x* = *y* **and** *y* = *n* **in** *less-linear*)
apply (*drule-tac m* = *n* **in** *poly-exp-divides*)
apply (*auto dest*: *Suc-le-eq* [*THEN iffD2*, *THEN* [*2*] *poly-exp-divides*]
        simp del: *pmult-Cons pexp-Suc*)
done

Order

lemma *some1-equalityD*: [| *n* = (@*n*. *P n*); *EX*! *n*. *P n* |] ==> *P n*
by (*blast intro*: *someI2*)

lemma (**in** *idom-char-0*) *order*:
    (([−*a*, *1*] % ^ *n*) *divides p* &
     ~(([−*a*, *1*] % ^ (*Suc n*)) *divides p*)) =
     ((*n* = *order a p*) & ~(*poly p* = *poly* []))
apply (*unfold order-def*)
apply (*rule iffI*)
apply (*blast dest*: *poly-divides-zero intro*!: *some1-equality* [*symmetric*] *poly-order*)
apply (*blast intro*!: *poly-order* [*THEN* [*2*] *some1-equalityD*])
done

lemma (**in** *idom-char-0*) *order2*: [| *poly p* ≠ *poly* [] |]
    ==> ([−*a*, *1*] % ^ (*order a p*)) *divides p* &
            ~(([−*a*, *1*] % ^ (*Suc*(*order a p*))) *divides p*)
by (*simp add*: *order del*: *pexp-Suc*)

lemma (**in** *idom-char-0*) *order-unique*: [| *poly p* ≠ *poly* []; ([−*a*, *1*] % ^ *n*) *divides p*;
        ~(([−*a*, *1*] % ^ (*Suc n*)) *divides p*)
    |] ==> (*n* = *order a p*)
by (*insert order* [*of a n p*], *auto*)

lemma (**in** *idom-char-0*) *order-unique-lemma*: (*poly p* ≠ *poly* [] & ([−*a*, *1*] % ^ *n*) *divides p* &
        ~(([−*a*, *1*] % ^ (*Suc n*)) *divides p*))
    ==> (*n* = *order a p*)
by (*blast intro*: *order-unique*)

lemma (**in** *ring-1*) *order-poly*: *poly p* = *poly q* ==> *order a p* = *order a q*
by (*auto simp add*: *fun-eq divides-def poly-mult order-def*)

lemma (**in** *semiring-1*) *pexp-one*[*simp*]: *p* % ^ (*Suc 0*) = *p*
apply (*induct p*)

apply (*auto simp add*: *numeral-1-eq-1*)
done

lemma (**in** *comm-ring-1*) *lemma-order-root*:
    0 < n & [− a, 1] %^ n divides p & ~ [− a, 1] %^ (Suc n) divides p
      ⟹ poly p a = 0
apply (*induct n arbitrary*: *a p*, *blast*)
apply (*auto simp add*: *divides-def poly-mult simp del*: *pmult-Cons*)
done

lemma (**in** *idom-char-0*) *order-root*: (*poly p a = 0*) = ((*poly p = poly []*) | *order a p ≠ 0*)
proof−
  let *?poly = poly*
  show *?thesis*
apply (*case-tac ?poly p = ?poly []*, *auto*)
apply (*simp add*: *poly-linear-divides del*: *pmult-Cons*, *safe*)
apply (*drule-tac* [!] *a = a* **in** *order2*)
apply (*rule ccontr*)
apply (*simp add*: *divides-def poly-mult fun-eq del*: *pmult-Cons*, *blast*)
using *neq0-conv*
apply (*blast intro*: *lemma-order-root*)
done
qed

lemma (**in** *idom-char-0*) *order-divides*: (([*−a, 1*] *%^ n*) *divides p*) = ((*poly p = poly []*) | *n ≤ order a p*)
proof−
  let *?poly = poly*
  show *?thesis*
apply (*case-tac ?poly p = ?poly []*, *auto*)
apply (*simp add*: *divides-def fun-eq poly-mult*)
apply (*rule-tac x = []* **in** *exI*)
apply (*auto dest*!: *order2* [**where** *a=a*]
      intro: *poly-exp-divides simp del*: *pexp-Suc*)
done
qed

lemma (**in** *idom-char-0*) *order-decomp*:
    poly p ≠ poly []
    *==> ∃ q. (poly p = poly (([−a, 1] %^ (order a p)) *** q)) &*
        ~([−a, 1] divides q)
apply (*unfold divides-def*)
apply (*drule order2* [**where** *a = a*])
apply (*simp add*: *divides-def del*: *pexp-Suc pmult-Cons*, *safe*)
apply (*rule-tac x = q* **in** *exI*, *safe*)
apply (*drule-tac x = qa* **in** *spec*)
apply (*auto simp add*: *poly-mult fun-eq poly-exp mult-ac simp del*: *pmult-Cons*)
done

Important composition properties of orders.

lemma *order-mult*: *poly* (*p* *** *q*) ≠ *poly* []
==> *order a* (*p* *** *q*) = *order a p* + *order* (*a*::'*a*::{*idom-char-0*}) *q*
apply (*cut-tac a* = *a* **and** *p* = *p* *** *q* **and** *n* = *order a p* + *order a q* **in** *order*)
apply (*auto simp add*: *poly-entire simp del*: *pmult-Cons*)
apply (*drule-tac a* = *a* **in** *order2*)+
apply *safe*
apply (*simp add*: *divides-def fun-eq poly-exp-add poly-mult del*: *pmult-Cons*, *safe*)
apply (*rule-tac x* = *qa* *** *qaa* **in** *exI*)
apply (*simp add*: *poly-mult mult-ac del*: *pmult-Cons*)
apply (*drule-tac a* = *a* **in** *order-decomp*)+
apply *safe*
apply (*subgoal-tac* [−*a*,*1*] *divides* (*qa* *** *qaa*))
apply (*simp add*: *poly-primes del*: *pmult-Cons*)
apply (*auto simp add*: *divides-def simp del*: *pmult-Cons*)
apply (*rule-tac x* = *qb* **in** *exI*)
apply (*subgoal-tac poly* ([−*a*, *1*] %^ (*order a p*) *** (*qa* *** *qaa*)) = *poly* ([−*a*, *1*] %^ (*order a p*) *** ([−*a*, *1*] *** *qb*)))
apply (*drule poly-mult-left-cancel* [*THEN iffD1*], *force*)
apply (*subgoal-tac poly* ([−*a*, *1*] %^ (*order a q*) *** ([−*a*, *1*] %^ (*order a p*) *** (*qa* *** *qaa*))) = *poly* ([−*a*, *1*] %^ (*order a q*) *** ([−*a*, *1*] %^ (*order a p*) *** ([−*a*, *1*] *** *qb*))))
apply (*drule poly-mult-left-cancel* [*THEN iffD1*], *force*)
apply (*simp add*: *fun-eq poly-exp-add poly-mult mult-ac del*: *pmult-Cons*)
done

lemma (**in** *idom-char-0*) *order-mult*:
assumes *pq0*: *poly* (*p* *** *q*) ≠ *poly* []
shows *order a* (*p* *** *q*) = *order a p* + *order a q*
proof−
let *?order* = *order*
let *?divides* = *op divides*
let *?poly* = *poly*
from *pq0*
show *?thesis*
apply (*cut-tac a* = *a* **and** *p* = *pmult p q* **and** *n* = *?order a p* + *?order a q* **in** *order*)
apply (*auto simp add*: *poly-entire simp del*: *pmult-Cons*)
apply (*drule-tac a* = *a* **in** *order2*)+
apply *safe*
apply (*simp add*: *divides-def fun-eq poly-exp-add poly-mult del*: *pmult-Cons*, *safe*)
apply (*rule-tac x* = *pmult qa qaa* **in** *exI*)
apply (*simp add*: *poly-mult mult-ac del*: *pmult-Cons*)
apply (*drule-tac a* = *a* **in** *order-decomp*)+
apply *safe*
apply (*subgoal-tac ?divides* [*uminus a*,*one*] (*pmult qa qaa*))
apply (*simp add*: *poly-primes del*: *pmult-Cons*)
apply (*auto simp add*: *divides-def simp del*: *pmult-Cons*)
apply (*rule-tac x* = *qb* **in** *exI*)

apply (*subgoal-tac ?poly* (*pmult* (*pexp* [*uminus a*, *one*] (*?order a p*)) (*pmult qa qaa*)) = *?poly* (*pmult* (*pexp* [*uminus a*, *one*] (*?order a p*)) (*pmult* [*uminus a*, *one*] *qb*)))
apply (*drule poly-mult-left-cancel* [*THEN iffD1*], *force*)
apply (*subgoal-tac ?poly* (*pmult* (*pexp* [*uminus a*, *one*] (*order a q*)) (*pmult* (*pexp* [*uminus a*, *one*] (*order a p*)) (*pmult qa qaa*))) = *?poly* (*pmult* (*pexp* [*uminus a*, *one*] (*order a q*)) (*pmult* (*pexp* [*uminus a*, *one*] (*order a p*)) (*pmult* [*uminus a*, *one*] *qb*))))
apply (*drule poly-mult-left-cancel* [*THEN iffD1*], *force*)
apply (*simp add*: *fun-eq poly-exp-add poly-mult mult-ac del*: *pmult-Cons*)
done
qed

lemma (**in** *idom-char-0*) *order-root2*: *poly p* ≠ *poly* [] ==> (*poly p a* = *0*) = (*order a p* ≠ *0*)
by (*rule order-root* [*THEN ssubst*], *auto*)

lemma (**in** *semiring-1*) *pmult-one*[*simp*]: [*1*] *** *p* = *p* **by** *auto*

lemma (**in** *semiring-0*) *poly-Nil-zero*: *poly* [] = *poly* [*0*]
by (*simp add*: *fun-eq*)

lemma (**in** *idom-char-0*) *rsquarefree-decomp*:
  [| *rsquarefree p*; *poly p a* = *0* |]
  ==> ∃ *q*. (*poly p* = *poly* ([−*a*, *1*] *** *q*)) & *poly q a* ≠ *0*
apply (*simp add*: *rsquarefree-def*, *safe*)
apply (*frule-tac a* = *a* **in** *order-decomp*)
apply (*drule-tac x* = *a* **in** *spec*)
apply (*drule-tac a* = *a* **in** *order-root2* [*symmetric*])
apply (*auto simp del*: *pmult-Cons*)
apply (*rule-tac x* = *q* **in** *exI*, *safe*)
apply (*simp add*: *poly-mult fun-eq*)
apply (*drule-tac p1* = *q* **in** *poly-linear-divides* [*THEN iffD1*])
apply (*simp add*: *divides-def del*: *pmult-Cons*, *safe*)
apply (*drule-tac x* = [] **in** *spec*)
apply (*auto simp add*: *fun-eq*)
done

Normalization of a polynomial.

lemma (**in** *semiring-0*) *poly-normalize*[*simp*]: *poly* (*pnormalize p*) = *poly p*
apply (*induct p*)
apply (*auto simp add*: *fun-eq*)
done

The degree of a polynomial.

lemma (**in** *semiring-0*) *lemma-degree-zero*:
  list-all (%*c*. *c* = *0*) *p* ⟷ *pnormalize p* = []
by (*induct p*, *auto*)

lemma (**in** *idom-char-0*) *degree-zero*:
assumes *pN*: *poly p* = *poly* [] **shows***degree p* = *0*
proof−
let *?pn* = *pnormalize*
from *pN*
show *?thesis*
apply (*simp add*: *degree-def*)
apply (*case-tac ?pn p* = [])
apply (*auto simp add*: *poly-zero lemma-degree-zero*)
done
qed

lemma (**in** *semiring-0*) *pnormalize-sing*: (*pnormalize* [*x*] = [*x*]) ⟷ *x* ≠ *0*
by *simp*
lemma (**in** *semiring-0*) *pnormalize-pair*: *y* ≠ *0* ⟷ (*pnormalize* [*x*, *y*] = [*x*, *y*])
by *simp*
lemma (**in** *semiring-0*) *pnormal-cons*: *pnormal p* ⟹ *pnormal* (*c*#*p*)
unfolding *pnormal-def* **by** *simp*
lemma (**in** *semiring-0*) *pnormal-tail*: *p*≠[] ⟹ *pnormal* (*c*#*p*) ⟹ *pnormal p*
unfolding *pnormal-def* **by**(*auto split*: *split-if-asm*)

lemma (**in** *semiring-0*) *pnormal-last-nonzero*: *pnormal p* ==> *last p* ≠ *0*
by(*induct p*) (*simp-all add*: *pnormal-def split*: *split-if-asm*)

lemma (**in** *semiring-0*) *pnormal-length*: *pnormal p* ⟹ *0* < *length p*
unfolding *pnormal-def length-greater-0-conv* **by** *blast*

lemma (**in** *semiring-0*) *pnormal-last-length*: ⟦*0* < *length p* ; *last p* ≠ *0*⟧ ⟹ *pnormal p*
by (*induct p*) (*auto simp*: *pnormal-def split*: *split-if-asm*)

lemma (**in** *semiring-0*) *pnormal-id*: *pnormal p* ⟷ (*0* < *length p* ∧ *last p* ≠ *0*)
using *pnormal-last-length pnormal-length pnormal-last-nonzero* **by** *blast*

lemma (**in** *idom-char-0*) *poly-Cons-eq*: *poly* (*c*#*cs*) = *poly* (*d*#*ds*) ⟷ *c*=*d* ∧ *poly cs* = *poly ds* (**is** *?lhs* ⟷ *?rhs*)
proof
assume *eq*: *?lhs*
hence ⋀*x*. *poly* ((*c*#*cs*) +++ −− (*d*#*ds*)) *x* = *0*
by (*simp only*: *poly-minus poly-add algebra-simps*) *simp*
hence *poly* ((*c*#*cs*) +++ −− (*d*#*ds*)) = *poly* [] **by**(*simp add*:*expand-fun-eq*)
hence *c* = *d* ∧ *list-all* (λ*x*. *x*=*0*) ((*cs* +++ −− *ds*))
unfolding *poly-zero* **by** (*simp add*: *poly-minus-def algebra-simps*)
hence *c* = *d* ∧ (∀ *x*. *poly* (*cs* +++ −− *ds*) *x* = *0*)
unfolding *poly-zero*[*symmetric*] **by** *simp*
thus *?rhs* **by** (*simp add*: *poly-minus poly-add algebra-simps expand-fun-eq*)
next

assume *?rhs* **then show** *?lhs* **by**(*simp add*:*expand-fun-eq*)
qed

lemma (**in** *idom-char-0*) *pnormalize-unique*: *poly p* = *poly q* ⟹ *pnormalize p* = *pnormalize q*
proof(*induct q arbitrary*: *p*)
case *Nil* **thus** *?case* **by** (*simp only*: *poly-zero lemma-degree-zero*) *simp*
next
case (*Cons c cs p*)
thus *?case*
proof(*induct p*)
case *Nil*
hence *poly* [] = *poly* (*c*#*cs*) **by** *blast*
then have *poly* (*c*#*cs*) = *poly* [] **by** *simp*
thus *?case* **by** (*simp only*: *poly-zero lemma-degree-zero*) *simp*
next
case (*Cons d ds*)
hence *eq*: *poly* (*d* # *ds*) = *poly* (*c* # *cs*) **by** *blast*
hence *eq'*: ⋀*x*. *poly* (*d* # *ds*) *x* = *poly* (*c* # *cs*) *x* **by** *simp*
hence *poly* (*d* # *ds*) *0* = *poly* (*c* # *cs*) *0* **by** *blast*
hence *dc*: *d* = *c* **by** *auto*
with *eq* **have** *poly ds* = *poly cs*
unfolding *poly-Cons-eq* **by** *simp*
with *Cons.prems* **have** *pnormalize ds* = *pnormalize cs* **by** *blast*
with *dc* **show** *?case* **by** *simp*
qed
qed

lemma (**in** *idom-char-0*) *degree-unique*: **assumes** *pq*: *poly p* = *poly q*
shows *degree p* = *degree q*
using *pnormalize-unique*[*OF pq*] **unfolding** *degree-def* **by** *simp*

lemma (**in** *semiring-0*) *pnormalize-length*: *length* (*pnormalize p*) ≤ *length p* **by** (*induct p*, *auto*)

lemma (**in** *semiring-0*) *last-linear-mul-lemma*:
last ((*a* %* *p*) +++ (*x*#(*b* %* *p*))) = (*if p*=[] *then x else b*∗*last p*)

apply (*induct p arbitrary*: *a x b*, *auto*)
apply (*subgoal-tac padd* (*cmult aa p*) (*times b a* # *cmult b p*) ≠ [], *simp*)
apply (*induct-tac p*, *auto*)
done

lemma (**in** *semiring-1*) *last-linear-mul*: **assumes** *p*:*p*≠[] **shows** *last* ([*a*,*1*] ∗∗∗ *p*) = *last p*
proof−
from *p* **obtain** *c cs* **where** *cs*: *p* = *c*#*cs* **by** (*cases p*, *auto*)
from *cs* **have** *eq*:[*a*,*1*] ∗∗∗ *p* = (*a* %* (*c*#*cs*)) +++ (*0*#(*1* %* (*c*#*cs*)))
by (*simp add*: *poly-cmult-distr*)

show *?thesis* **using** *cs*
unfolding *eq last-linear-mul-lemma* **by** *simp*
qed

lemma (**in** *semiring-0*) *pnormalize-eq*: *last p* ≠ *0* ⟹ *pnormalize p* = *p*
by (*induct p*) (*auto split*: *split-if-asm*)

lemma (**in** *semiring-0*) *last-pnormalize*: *pnormalize p* ≠ [] ⟹ *last* (*pnormalize p*) ≠ *0*
by (*induct p*, *auto*)

lemma (**in** *semiring-0*) *pnormal-degree*: *last p* ≠ *0* ⟹ *degree p* = *length p* − *1*
using *pnormalize-eq*[*of p*] **unfolding** *degree-def* **by** *simp*

lemma (**in** *semiring-0*) *poly-Nil-ext*: *poly* [] = (λ*x*. *0*) **by** (*rule ext*) *simp*

lemma (**in** *idom-char-0*) *linear-mul-degree*: **assumes** *p*: *poly p* ≠ *poly* []
shows *degree* ([*a*,*1*] **** p*) = *degree p* + *1*
proof−
from *p* **have** *pnz*: *pnormalize p* ≠ []
unfolding *poly-zero lemma-degree-zero* **.**

from *last-linear-mul*[*OF pnz*, *of a*] *last-pnormalize*[*OF pnz*]
have *l0*: *last* ([*a*, *1*] **** pnormalize p*) ≠ *0* **by** *simp*
from *last-pnormalize*[*OF pnz*] *last-linear-mul*[*OF pnz*, *of a*]
pnormal-degree[*OF l0*] *pnormal-degree*[*OF last-pnormalize*[*OF pnz*]] *pnz*

have *th*: *degree* ([*a*,*1*] **** pnormalize p*) = *degree* (*pnormalize p*) + *1*
by (*auto simp add*: *poly-length-mult*)

have *eqs*: *poly* ([*a*,*1*] **** pnormalize p*) = *poly* ([*a*,*1*] **** p*)
by (*rule ext*) (*simp add*: *poly-mult poly-add poly-cmult*)
from *degree-unique*[*OF eqs*] *th*
show *?thesis* **by** (*simp add*: *degree-unique*[*OF poly-normalize*])
qed

lemma (**in** *idom-char-0*) *linear-pow-mul-degree*:
degree([*a*,*1*] *%^n *** p*) = (*if poly p* = *poly* [] *then 0 else degree p* + *n*)
proof(*induct n arbitrary*: *a p*)
case (*0 a p*)
{assume *p*: *poly p* = *poly* []
hence *?case* **using** *degree-unique*[*OF p*] **by** (*simp add*: *degree-def*)**}**
moreover
{assume *p*: *poly p* ≠ *poly* [] **hence** *?case* **by** (*auto simp add*: *poly-Nil-ext*) **}**
ultimately show *?case* **by** *blast*
next
case (*Suc n a p*)
have *eq*: *poly* ([*a*,*1*] *%^*(*Suc n*) **** p*) = *poly* ([*a*,*1*] *%^ n **** ([*a*,*1*] **** p*))

```
   apply (rule ext, simp add: poly-mult poly-add poly-cmult)
   by (simp add: mult-ac add-ac right-distrib)
  note deq = degree-unique[OF eq]
  {assume p: poly p = poly []
   with eq have eq': poly ([a,1] % ^(Suc n) *** p) = poly []
    by - (rule ext,simp add: poly-mult poly-cmult poly-add)
   from degree-unique[OF eq'] p have ?case by (simp add: degree-def)}
  moreover
  {assume p: poly p ≠ poly []
   from p have ap: poly ([a,1] *** p) ≠ poly []
    using poly-mult-not-eq-poly-Nil unfolding poly-entire by auto
   have eq: poly ([a,1] % ^(Suc n) *** p) = poly ([a,1]% ^n *** ([a,1] *** p))
    by (rule ext, simp add: poly-mult poly-add poly-exp poly-cmult algebra-simps)
   from ap have ap': (poly ([a,1] *** p) = poly []) = False by blast
   have th0: degree ([a,1]% ^n *** ([a,1] *** p)) = degree ([a,1] *** p) + n
    apply (simp only: Suc.hyps[of a pmult [a,one] p] ap')
    by simp

   from degree-unique[OF eq] ap p th0 linear-mul-degree[OF p, of a]
   have ?case by (auto simp del: poly.simps)}
  ultimately show ?case by blast
qed

lemma (in idom-char-0) order-degree:
  assumes p0: poly p ≠ poly []
  shows order a p ≤ degree p
proof−
  from order2[OF p0, unfolded divides-def]
  obtain q where q: poly p = poly ([− a, 1]% ^ (order a p) *** q) by blast
  {assume poly q = poly []
   with q p0 have False by (simp add: poly-mult poly-entire)}
  with degree-unique[OF q, unfolded linear-pow-mul-degree]
  show ?thesis by auto
qed
```

Tidier versions of finiteness of roots.

```
lemma (in idom-char-0) poly-roots-finite-set: poly p ≠ poly [] ==> finite {x. poly
p x = 0}
unfolding poly-roots-finite .
```

bound for polynomial.

```
lemma poly-mono: abs(x) ≤ k ==> abs(poly p (x::'a::{linordered-idom})) ≤ poly
(map abs p) k
apply (induct p, auto)
apply (rule-tac y = abs a + abs (x * poly p x) in order-trans)
apply (rule abs-triangle-ineq)
apply (auto intro!: mult-mono simp add: abs-mult)
done
```

lemma (**in** *semiring-0*) *poly-Sing*: *poly* [*c*] *x* = *c* **by** *simp*

end

72 While-Combinator: A general "while" combinator

theory *While-Combinator*
imports *Main*
begin

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

function (*tailrec*) *while* :: ('*a* ⇒ *bool*) ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a*
where
while-unfold[*simp del*]: *while b c s* = (*if b s then while b c* (*c s*) *else s*)
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 anti-symmetry simproc turns the subset relationship back into equality.

```
theorem P (lfp (λN::int set. {0} ∪ {(n + 2) mod 6 | n. n ∈ N})) =
  P {0, 4, 2}
proof −
  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

73 Order-Relation: Orders as Relations

theory *Order-Relation*
imports *Main*
begin

73.1 Orders on a set

definition *preorder-on A r* ≡ *refl-on A r* ∧ *trans r*

definition *partial-order-on A r* ≡ *preorder-on A r* ∧ *antisym r*

definition *linear-order-on A r* ≡ *partial-order-on A r* ∧ *total-on A r*

definition *strict-linear-order-on A r* ≡ *trans r* ∧ *irrefl r* ∧ *total-on A r*

definition *well-order-on A r* ≡ *linear-order-on A r* ∧ *wf*(*r* − *Id*)

lemmas *order-on-defs* =
preorder-on-def partial-order-on-def linear-order-on-def
strict-linear-order-on-def well-order-on-def

lemma *preorder-on-empty*[*simp*]: *preorder-on* {} {}
by(*simp add*:*preorder-on-def trans-def*)

lemma *partial-order-on-empty*[*simp*]: *partial-order-on* {} {}
by(*simp add*:*partial-order-on-def*)

lemma *lnear-order-on-empty*[*simp*]: *linear-order-on* {} {}
by(*simp add*:*linear-order-on-def*)

lemma *well-order-on-empty*[*simp*]: *well-order-on* {} {}
by(*simp add*:*well-order-on-def*)

lemma *preorder-on-converse*[*simp*]: *preorder-on A* (*r*^−*1*) = *preorder-on A r*
by (*simp add*:*preorder-on-def*)

lemma *partial-order-on-converse*[*simp*]:
partial-order-on A (*r*^−1) = *partial-order-on A r*
by (*simp add*: *partial-order-on-def*)

lemma *linear-order-on-converse*[*simp*]:
linear-order-on A (*r*^−1) = *linear-order-on A r*
by (*simp add*: *linear-order-on-def*)

lemma *strict-linear-order-on-diff-Id*:
linear-order-on A r ⟹ *strict-linear-order-on A* (*r*−*Id*)
by(*simp add*: *order-on-defs trans-diff-Id*)

73.2 Orders on the field

abbreviation *Refl r* ≡ *refl-on* (*Field r*) *r*

abbreviation *Preorder r* ≡ *preorder-on* (*Field r*) *r*

abbreviation *Partial-order r* ≡ *partial-order-on* (*Field r*) *r*

abbreviation *Total r* ≡ *total-on* (*Field r*) *r*

abbreviation *Linear-order r* ≡ *linear-order-on* (*Field r*) *r*

abbreviation *Well-order r* ≡ *well-order-on* (*Field r*) *r*

lemma *subset-Image-Image-iff*:
⟦ *Preorder r*; *A* ⊆ *Field r*; *B* ⊆ *Field r*⟧ ⟹
r `` *A* ⊆ *r* `` *B* ⟷ (∀ *a*∈*A*.∃ *b*∈*B*. (*b*,*a*):*r*)
apply(*auto simp add*: *subset-eq preorder-on-def refl-on-def Image-def*)
apply *metis*
by(*metis trans-def*)

lemma *subset-Image1-Image1-iff*:
⟦ *Preorder r*; *a* : *Field r*; *b* : *Field r*⟧ ⟹ *r* `` {*a*} ⊆ *r* `` {*b*} ⟷ (*b*,*a*):*r*
by(*simp add*:*subset-Image-Image-iff*)

lemma *Refl-antisym-eq-Image1-Image1-iff*:
⟦*Refl r*; *antisym r*; *a*:*Field r*; *b*:*Field r*⟧ ⟹ *r* `` {*a*} = *r* `` {*b*} ⟷ *a*=*b*
by(*simp add*: *expand-set-eq antisym-def refl-on-def*) *metis*

lemma *Partial-order-eq-Image1-Image1-iff*:
⟦*Partial-order r*; *a*:*Field r*; *b*:*Field r*⟧ ⟹ *r* `` {*a*} = *r* `` {*b*} ⟷ *a*=*b*
by(*auto simp*:*order-on-defs Refl-antisym-eq-Image1-Image1-iff*)

73.3 Orders on a type

abbreviation *strict-linear-order* ≡ *strict-linear-order-on UNIV*

abbreviation *linear-order* ≡ *linear-order-on UNIV*

abbreviation *well-order r* ≡ *well-order-on UNIV*

end

74 Zorn: Zorn's Lemma

theory *Zorn*
imports *Order-Relation Main*
begin

definition *chain-subset* :: *'a set set* ⇒ *bool* (*chain*$_{\subseteq}$) **where**
chain$_{\subseteq}$ *C* ≡ ∀ *A*∈*C*.∀ *B*∈*C*. *A* ⊆ *B* ∨ *B* ⊆ *A*

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

definition
chain :: *'a set set* => *'a set set set* **where**
chain S = {*F*. *F* ⊆ *S* & *chain*$_{\subseteq}$ *F*}

definition
super :: [*'a set set*,*'a set set*] => *'a set set set* **where**
super S c = {*d*. *d* ∈ *chain S* & *c* ⊂ *d*}

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

definition
succ :: [*'a set set*,*'a set set*] => *'a set set* **where**
succ S c =
(*if c* ∉ *chain S* | *c* ∈ *maxchain S*
then c else SOME c'. *c'* ∈ *super S c*)

inductive-set
TFin :: *'a set set* => *'a set set set*
for *S* :: *'a set set*
where
succI: *x* ∈ *TFin S* ==> *succ S x* ∈ *TFin S*
| *Pow-UnionI*: *Y* ∈ *Pow*(*TFin S*) ==> *Union*(*Y*) ∈ *TFin S*

74.1 Mathematical Preamble

lemma *Union-lemma0*:
(∀ *x* ∈ *C*. *x* ⊆ *A* | *B* ⊆ *x*) ==> *Union*(*C*) ⊆ *A* | *B* ⊆ *Union*(*C*)
by *blast*

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

lemma *Abrial-axiom1*: $x \subseteq succ\ S\ x$
apply (*auto simp add*: *succ-def super-def maxchain-def*)
apply (*rule contrapos-np*, *assumption*)
apply (*rule-tac Q*=$\lambda S.\ xa \in S$ **in** *someI2*, *blast*+)
done

lemmas *TFin-UnionI* = *TFin.Pow-UnionI* [*OF PowI*]

lemma *TFin-induct*:
assumes *H*: $n \in TFin\ S$
and *I*: $!!x.\ x \in TFin\ S \Longrightarrow P\ x \Longrightarrow P\ (succ\ S\ x)$
$!!Y.\ Y \subseteq TFin\ S \Longrightarrow Ball\ Y\ P \Longrightarrow P(Union\ Y)$
shows *P n* **using** *H*
apply (*induct rule*: *TFin.induct* [**where** *P*=*P*])
apply (*blast intro*: *I*)+
done

lemma *succ-trans*: $x \subseteq y \Longrightarrow x \subseteq succ\ S\ y$
apply (*erule subset-trans*)
apply (*rule Abrial-axiom1*)
done

Lemma 1 of section 3.1

lemma *TFin-linear-lemma1*:
$[|\ n \in TFin\ S;\ m \in TFin\ S;$
$\forall x \in TFin\ S.\ x \subseteq m \longrightarrow x = m \mid succ\ S\ x \subseteq m$
$|] \Longrightarrow n \subseteq m \mid succ\ S\ m \subseteq n$
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*:
$m \in TFin\ S \Longrightarrow \forall n \in TFin\ S.\ n \subseteq m \longrightarrow n=m \mid succ\ S\ n \subseteq m$
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*:
[| $n \subseteq m$; $m \in TFin\ S$; $n \in TFin\ S$ |] ==> $n{=}m \mid succ\ S\ n \subseteq m$
by (*rule TFin-linear-lemma2* [*rule-format*])

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

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

74.2 Hausdorff's Theorem: Every Set Contains a Maximal Chain.

NB: We assume the partial ordering is $\subseteq$, the subset relation!

lemma *empty-set-mem-chain*: $(\{\} :: 'a\ set\ set) \in chain\ S$
by (*unfold chain-def chain-subset-def*) *auto*

lemma *super-subset-chain*: *super S c* ⊆ *chain S*
by (*unfold super-def*) *blast*

lemma *maxchain-subset-chain*: *maxchain S* ⊆ *chain S*
by (*unfold maxchain-def*) *blast*

lemma *mem-super-Ex*: *c* ∈ *chain S* − *maxchain S* ==> *EX d*. *d* ∈ *super S c*
by (*unfold super-def maxchain-def*) *auto*

lemma *select-super*:
c ∈ *chain S* − *maxchain S* ==> (ϵ *c'*. *c'*: *super S c*): *super S c*
apply (*erule mem-super-Ex* [*THEN exE*])
apply (*rule someI2* [**where** *Q*=%*X*. *X* : *super S c*], *auto*)
done

lemma *select-not-equals*:
c ∈ *chain S* − *maxchain S* ==> (ϵ *c'*. *c'*: *super S c*) ≠ *c*
apply (*rule notI*)
apply (*drule select-super*)
apply (*simp add*: *super-def less-le*)
done

lemma *succI3*: *c* ∈ *chain S* − *maxchain S* ==> *succ S c* = (ϵ *c'*. *c'*: *super S c*)
by (*unfold succ-def*) (*blast intro*!: *if-not-P*)

lemma *succ-not-equals*: *c* ∈ *chain S* − *maxchain S* ==> *succ S c* ≠ *c*
apply (*frule succI3*)
apply (*simp* (*no-asm-simp*))
apply (*rule select-not-equals*, *assumption*)
done

lemma *TFin-chain-lemma4*: *c* ∈ *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 chain-subset-def*)
apply (*rule CollectI*, *safe*)
apply (*drule bspec*, *assumption*)
apply (*rule-tac* [*2*] *m1* = *Xa* **and** *n1* = *X* **in** *TFin-subset-linear* [*THEN disjE*], *best*+)
done

theorem *Hausdorff*: ∃ *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

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

lemma *chain-extend*:
[| $c \in$ *chain* S; $z \in S$; $\forall x \in c.\ x \subseteq (z::'a\ set)$ |] ==> $\{z\}$ *Un* $c \in$ *chain* S
by (*unfold chain-def chain-subset-def*) *blast*

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

lemma *chain-ball-Union-upper*: $c \in$ *chain* S ==> $\forall x \in c.\ x \subseteq$ *Union*(c)
by *auto*

lemma *maxchain-Zorn*:
[| $c \in$ *maxchain* S; $u \in S$; *Union*(c) $\subseteq u$ |] ==> *Union*(c) = u
apply (*rule ccontr*)
apply (*simp add*: *maxchain-def*)
apply (*erule conjE*)
apply (*subgoal-tac* ($\{u\}$ *Un* c) $\in$ *super* S c)
apply *simp*
apply (*unfold super-def less-le*)
apply (*blast intro*: *chain-extend dest*: *chain-Union-upper*)
done

theorem *Zorn-Lemma*:
$\forall c \in$ *chain* S. *Union*(c): S ==> $\exists y \in S.\ \forall z \in S.\ y \subseteq z$ --> $y = z$
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

74.4 Alternative version of Zorn's Lemma

lemma *Zorn-Lemma2*:
$\forall c \in$ *chain* S. $\exists y \in S.\ \forall x \in c.\ x \subseteq y$
==> $\exists y \in S.\ \forall x \in S.\ (y::'a\ set) \subseteq x$ --> $y = x$
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 less-le*)
apply (*blast elim*!: *equalityCE*)
done

Various other lemmas

lemma *chainD*: [| $c \in$ *chain* S; $x \in c$; $y \in c$ |] ==> $x \subseteq y \mid y \subseteq x$
by (*unfold chain-def chain-subset-def*) *blast*

lemma *chainD2*: !!(c :: *'a set set*). $c \in$ *chain* S ==> $c \subseteq S$
by (*unfold chain-def*) *blast*

definition *Chain* :: (*'a***'a*)*set* ⇒ *'a set set* **where**
Chain r ≡ {*A*. ∀ *a*∈*A*.∀ *b*∈*A*. (*a*,*b*) : *r* ∨ (*b*,*a*) ∈ *r*}

lemma *mono-Chain*: *r* ⊆ *s* ⟹ *Chain r* ⊆ *Chain s*
unfolding *Chain-def* **by** *blast*

Zorn's lemma for partial orders:

lemma *Zorns-po-lemma*:
assumes *po*: *Partial-order r* **and** *u*: ∀ *C*∈*Chain r*. ∃ *u*∈*Field r*. ∀ *a*∈*C*. (*a*,*u*):*r*
shows ∃ *m*∈*Field r*. ∀ *a*∈*Field r*. (*m*,*a*):*r* ⟶ *a*=*m*
proof−
have *Preorder r* **using** *po* **by**(*simp add*:*partial-order-on-def*)
— Mirror r in the set of subsets below (wrt r) elements of A
let *?B* = %*x*. *r*^−1 `` {*x*} **let** *?S* = *?B* ` *Field r*
have ∀ *C* ∈ *chain ?S*. *EX U*:*?S*. *ALL A*:*C*. *A*⊆*U*
proof (*auto simp*:*chain-def chain-subset-def*)
fix *C* **assume** *1*: *C* ⊆ *?S* **and** *2*: ∀ *A*∈*C*.∀ *B*∈*C*. *A*⊆*B* | *B*⊆*A*
let *?A* = {*x*∈*Field r*. ∃ *M*∈*C*. *M* = *?B x*}
have *C* = *?B* ` *?A* **using** *1* **by**(*auto simp*: *image-def*)
have *?A*∈*Chain r*
proof (*simp add*:*Chain-def*, *intro allI impI*, *elim conjE*)
fix *a b*
assume *a* ∈ *Field r ?B a* ∈ *C b* ∈ *Field r ?B b* ∈ *C*
hence *?B a* ⊆ *?B b* ∨ *?B b* ⊆ *?B a* **using** *2* **by** *auto*
thus (*a*, *b*) ∈ *r* ∨ (*b*, *a*) ∈ *r* **using** ‹*Preorder r*› ‹*a*:*Field r*› ‹*b*:*Field r*›
by (*simp add*:*subset-Image1-Image1-iff*)
qed
then obtain *u* **where** *uA*: *u*:*Field r* ∀ *a*∈*?A*. (*a*,*u*) : *r* **using** *u* **by** *auto*
have ∀ *A*∈*C*. *A* ⊆ *r*^−1 `` {*u*} (**is** *?P u*)
proof *auto*
fix *a B* **assume** *aB*: *B*:*C a*:*B*
with *1* **obtain** *x* **where** *x*:*Field r B* = *r*^−1 `` {*x*} **by** *auto*

```
    thus (a,u) : r using uA aB ‹Preorder r›
      by (auto simp add: preorder-on-def refl-on-def) (metis transD)
   qed
   thus EX u:Field r. ?P u using ‹u:Field r› by blast
 qed
 from Zorn-Lemma2[OF this]
 obtain m B where m:Field r B = r^-1 `` {m}
   ∀x∈Field r. B ⊆ r^-1 `` {x} ⟶ B = r^-1 `` {x}
   by auto
 hence ∀a∈Field r. (m, a) ∈ r ⟶ a = m using po ‹Preorder r› ‹m:Field r›
   by(auto simp:subset-Image1-Image1-iff Partial-order-eq-Image1-Image1-iff)
 thus ?thesis using ‹m:Field r› by blast
qed


definition init-seg-of :: (('a*'a)set * ('a*'a)set)set where
init-seg-of == {(r,s). r ⊆ s ∧ (∀ a b c. (a,b):s ∧ (b,c):r ⟶ (a,b):r)}

abbreviation initialSegmentOf :: ('a*'a)set ⇒ ('a*'a)set ⇒ bool
          (infix initial'-segment'-of 55) where
r initial-segment-of s == (r,s):init-seg-of

lemma refl-on-init-seg-of[simp]: r initial-segment-of r
by(simp add:init-seg-of-def)

lemma trans-init-seg-of:
  r initial-segment-of s ⟹ s initial-segment-of t ⟹ r initial-segment-of t
by(simp (no-asm-use) add: init-seg-of-def)
  (metis Domain-iff UnCI Un-absorb2 subset-trans)

lemma antisym-init-seg-of:
  r initial-segment-of s ⟹ s initial-segment-of r ⟹ r=s
unfolding init-seg-of-def by safe

lemma Chain-init-seg-of-Union:
  R ∈ Chain init-seg-of ⟹ r∈R ⟹ r initial-segment-of ⋃R
by(auto simp add:init-seg-of-def Chain-def Ball-def) blast

lemma chain-subset-trans-Union:
  chain⊆ R ⟹ ∀r∈R. trans r ⟹ trans(⋃R)
apply(auto simp add:chain-subset-def)
apply(simp (no-asm-use) add:trans-def)
apply (metis subsetD)
done

lemma chain-subset-antisym-Union:
  chain⊆ R ⟹ ∀r∈R. antisym r ⟹ antisym(⋃R)
apply(auto simp add:chain-subset-def antisym-def)
apply (metis subsetD)
```

done

lemma *chain-subset-Total-Union*:
assumes $chain_{\subseteq}\ R\ \forall r{\in}R.\ Total\ r$
shows $Total\ (\bigcup R)$
proof (*simp add*: *total-on-def Ball-def*, *auto del*:*disjCI*)
  fix *r s a b* **assume** *A*: *r*:*R s*:*R a*:*Field r b*:*Field s* $a{\neq}b$
  from ‹$chain_{\subseteq}\ R$› ‹*r*:*R*› ‹*s*:*R*› **have** $r{\subseteq}s \vee s{\subseteq}r$
    by(*simp add*:*chain-subset-def*)
  thus $(\exists r{\in}R.\ (a,b) \in r) \vee (\exists r{\in}R.\ (b,a) \in r)$
  proof
    assume $r{\subseteq}s$ **hence** $(a,b){:}s \vee (b,a){:}s$ **using** *assms*(*2*) *A*
      by(*simp add*:*total-on-def*)(*metis mono-Field subsetD*)
    thus *?thesis* **using** ‹*s*:*R*› **by** *blast*
  next
    assume $s{\subseteq}r$ **hence** $(a,b){:}r \vee (b,a){:}r$ **using** *assms*(*2*) *A*
      by(*simp add*:*total-on-def*)(*metis mono-Field subsetD*)
    thus *?thesis* **using** ‹*r*:*R*› **by** *blast*
  qed
qed

lemma *wf-Union-wf-init-segs*:
assumes $R \in Chain\ init\text{-}seg\text{-}of$ **and** $\forall r{\in}R.\ wf\ r$ **shows** $wf(\bigcup R)$
proof(*simp add*:*wf-iff-no-infinite-down-chain*, *rule ccontr*, *auto*)
  fix *f* **assume** *1*: $\forall i.\ \exists r{\in}R.\ (f(Suc\ i), f\ i) \in r$
  then obtain *r* **where** *r*:*R* **and** $(f(Suc\ 0), f\ 0) : r$ **by** *auto*
  { fix *i* **have** $(f(Suc\ i), f\ i) \in r$
    proof(*induct i*)
      case *0* **show** *?case* **by** *fact*
    next
      case (*Suc i*)
      moreover obtain *s* **where** $s{\in}R$ **and** $(f(Suc(Suc\ i)), f(Suc\ i)) \in s$
        using *1* **by** *auto*
      moreover hence *s initial-segment-of r* $\vee$ *r initial-segment-of s*
        using *assms*(*1*) ‹*r*:*R*› **by**(*simp add*: *Chain-def*)
      ultimately show *?case* **by**(*simp add*:*init-seg-of-def*) *blast*
    qed
  }
  thus *False* **using** *assms*(*2*) ‹*r*:*R*›
    by(*simp add*:*wf-iff-no-infinite-down-chain*) *blast*
qed

lemma *initial-segment-of-Diff*:
  p initial-segment-of q $\Longrightarrow$ *p* $-$ *s initial-segment-of q* $-$ *s*
unfolding *init-seg-of-def* **by** *blast*

lemma *Chain-inits-DiffI*:
  $R \in Chain\ init\text{-}seg\text{-}of \Longrightarrow \{r - s\ |r.\ r \in R\} \in Chain\ init\text{-}seg\text{-}of$
unfolding *Chain-def* **by** (*blast intro*: *initial-segment-of-Diff*)

theorem *well-ordering*: $\exists\, r{::}('a*'a)set.\ Well\text{-}order\ r \wedge Field\ r = UNIV$
proof−
— The initial segment relation on well-orders:
let *?WO* = $\{r{::}('a*'a)set.\ Well\text{-}order\ r\}$
def $I \equiv init\text{-}seg\text{-}of \cap\ ?WO \times\ ?WO$
have *I-init*: $I \subseteq init\text{-}seg\text{-}of$ **by**(*auto simp*:*I-def*)
hence *subch*: $!!R.\ R : Chain\ I \Longrightarrow chain_{\subseteq}\ R$
by(*auto simp*:*init-seg-of-def chain-subset-def Chain-def*)
have *Chain-wo*: $!!R\ r.\ R \in Chain\ I \Longrightarrow r \in R \Longrightarrow Well\text{-}order\ r$
by(*simp add*:*Chain-def I-def*) *blast*
have *FI*: *Field I* = *?WO* **by**(*auto simp add*:*I-def init-seg-of-def Field-def*)
hence *0*: *Partial-order I*
by(*auto simp*: *partial-order-on-def preorder-on-def antisym-def antisym-init-seg-of refl-on-def trans-def I-def elim*!: *trans-init-seg-of*)
— I-chains have upper bounds in ?WO wrt I: their Union
{ fix *R* **assume** $R \in Chain\ I$
hence *Ris*: $R \in Chain\ init\text{-}seg\text{-}of$ **using** *mono-Chain*[*OF I-init*] **by** *blast*
have *subch*: $chain_{\subseteq}\ R$ **using** ‹$R : Chain\ I$› *I-init*
by(*auto simp*:*init-seg-of-def chain-subset-def Chain-def*)
have $\forall\, r{\in}R.\ Refl\ r\ \forall\, r{\in}R.\ trans\ r\ \forall\, r{\in}R.\ antisym\ r\ \forall\, r{\in}R.\ Total\ r$
$\forall\, r{\in}R.\ wf(r{-}Id)$
using *Chain-wo*[*OF* ‹$R \in Chain\ I$›] **by**(*simp-all add*:*order-on-defs*)
have *Refl* $(\bigcup R)$ **using** ‹$\forall\, r{\in}R.\ Refl\ r$› **by**(*auto simp*:*refl-on-def*)
moreover have *trans* $(\bigcup R)$
by(*rule chain-subset-trans-Union*[*OF subch* ‹$\forall\, r{\in}R.\ trans\ r$›])
moreover have *antisym*$(\bigcup R)$
by(*rule chain-subset-antisym-Union*[*OF subch* ‹$\forall\, r{\in}R.\ antisym\ r$›])
moreover have *Total* $(\bigcup R)$
by(*rule chain-subset-Total-Union*[*OF subch* ‹$\forall\, r{\in}R.\ Total\ r$›])
moreover have *wf*$((\bigcup R){-}Id)$
proof−
have $(\bigcup R){-}Id = \bigcup\{r{-}Id|r.\ r \in R\}$ **by** *blast*
with ‹$\forall\, r{\in}R.\ wf(r{-}Id)$› *wf-Union-wf-init-segs*[*OF Chain-inits-DiffI*[*OF Ris*]]
show *?thesis* **by** (*simp* (*no-asm-simp*)) *blast*
qed
ultimately have *Well-order* $(\bigcup R)$ **by**(*simp add*:*order-on-defs*)
moreover have $\forall\, r \in R.\ r\ initial\text{-}segment\text{-}of\ \bigcup R$ **using** *Ris*
by(*simp add*: *Chain-init-seg-of-Union*)
ultimately have $\bigcup R : ?WO \wedge (\forall\, r{\in}R.\ (r{,}\bigcup R) : I)$
using *mono-Chain*[*OF I-init*] ‹$R \in Chain\ I$›
by(*simp* (*no-asm*) *add*:*I-def del*:*Field-Union*)(*metis Chain-wo subsetD*)
}
hence *1*: $\forall\, R \in Chain\ I.\ \exists\, u{\in}Field\ I.\ \forall\, r{\in}R.\ (r{,}u) : I$ **by** (*subst FI*) *blast*
— Zorn's Lemma yields a maximal well-order m:
then obtain $m{::}('a*'a)set$ **where** *Well-order m* **and**
max: $\forall\, r.\ Well\text{-}order\ r \wedge (m{,}r){:}I \longrightarrow r{=}m$
using *Zorns-po-lemma*[*OF 0 1*] **by** (*auto simp*:*FI*)
— Now show by contradiction that m covers the whole type:

{ fix $x::'a$ **assume** $x \notin$ *Field m*
— We assume that x is not covered and extend m at the top with x
have $m \neq \{\}$
proof
assume $m=\{\}$
moreover have *Well-order* $\{(x,x)\}$
by(*simp add*:*order-on-defs refl-on-def trans-def antisym-def total-on-def Field-def Domain-def Range-def*)
ultimately show *False* **using** *max*
by (*auto simp*:*I-def init-seg-of-def simp del*:*Field-insert*)
qed
hence *Field m* $\neq \{\}$ **by**(*auto simp*:*Field-def*)
moreover have *wf*(*m*−*Id*) **using** ‹*Well-order m*›
by(*simp add*:*well-order-on-def*)
— The extension of m by x:
let *?s* = $\{(a,x)|a.\ a$: *Field m*$\}$ **let** *?m* = *insert* (x,x) *m Un ?s*
have *Fm*: *Field ?m* = *insert x* (*Field m*)
apply(*simp add*:*Field-insert Field-Un*)
unfolding *Field-def* **by** *auto*
have *Refl m trans m antisym m Total m wf*(*m*−*Id*)
using ‹*Well-order m*› **by**(*simp-all add*:*order-on-defs*)
— We show that the extension is a well-order
have *Refl ?m* **using** ‹*Refl m*› *Fm* **by**(*auto simp*:*refl-on-def*)
moreover have *trans ?m* **using** ‹*trans m*› ‹*x* $\notin$ *Field m*›
unfolding *trans-def Field-def Domain-def Range-def* **by** *blast*
moreover have *antisym ?m* **using** ‹*antisym m*› ‹*x* $\notin$ *Field m*›
unfolding *antisym-def Field-def Domain-def Range-def* **by** *blast*
moreover have *Total ?m* **using** ‹*Total m*› *Fm* **by**(*auto simp*: *total-on-def*)
moreover have *wf*(*?m*−*Id*)
proof−
have *wf ?s* **using** ‹*x* $\notin$ *Field m*›
by(*auto simp add*:*wf-eq-minimal Field-def Domain-def Range-def*) *metis*
thus *?thesis* **using** ‹*wf*(*m*−*Id*)› ‹*x* $\notin$ *Field m*›
wf-subset[*OF* ‹*wf ?s*› *Diff-subset*]
by (*fastsimp intro*!: *wf-Un simp add*: *Un-Diff Field-def*)
qed
ultimately have *Well-order ?m* **by**(*simp add*:*order-on-defs*)
— We show that the extension is above m
moreover hence $(m,?m) : I$ **using** ‹*Well-order m*› ‹*x* $\notin$ *Field m*›
by(*fastsimp simp*:*I-def init-seg-of-def Field-def Domain-def Range-def*)
ultimately
— This contradicts maximality of m:
have *False* **using** *max* ‹*x* $\notin$ *Field m*› **unfolding** *Field-def* **by** *blast*
}
hence *Field m* = *UNIV* **by** *auto*
moreover with ‹*Well-order m*› **have** *Well-order m* **by** *simp*
ultimately show *?thesis* **by** *blast*
qed

corollary *well-order-on*: $\exists\, r{::}('a*'a)set.$ *well-order-on A r*
proof −
obtain $r{::}('a*'a)set$ **where** *wo*: *Well-order r* **and** *univ*: *Field r = UNIV*
using *well-ordering*[**where** $'a = 'a$] **by** *blast*
let *?r* = $\{(x,y).\ x{:}A\ \&\ y{:}A\ \&\ (x,y){:}r\}$
have *1*: *Field ?r = A* **using** *wo univ*
by(*fastsimp simp*: *Field-def Domain-def Range-def order-on-defs refl-on-def*)
have *Refl r trans r antisym r Total r wf*$(r-Id)$
using ‹*Well-order r*› **by**(*simp-all add*:*order-on-defs*)
have *Refl ?r* **using** ‹*Refl r*› **by**(*auto simp*:*refl-on-def 1 univ*)
moreover have *trans ?r* **using** ‹*trans r*›
unfolding *trans-def* **by** *blast*
moreover have *antisym ?r* **using** ‹*antisym r*›
unfolding *antisym-def* **by** *blast*
moreover have *Total ?r* **using** ‹*Total r*› **by**(*simp add*:*total-on-def 1 univ*)
moreover have *wf*$(?r - Id)$ **by**(*rule wf-subset*[*OF* ‹*wf*$(r-Id)$›]) *blast*
ultimately have *Well-order ?r* **by**(*simp add*:*order-on-defs*)
with *1* **show** *?thesis* **by** *metis*
qed

end

75 List-Prefix: List prefixes and postfixes

theory *List-Prefix*
imports *List Main*
begin

75.1 Prefix order on lists

instantiation *list* :: (*type*) *order*
begin

definition
prefix-def [*code del*]: $xs \le ys = (\exists\, zs.\ ys = xs\ @\ zs)$

definition
strict-prefix-def [*code del*]: $xs < ys = (xs \le ys \wedge xs \neq (ys{::}'a\ list))$

instance
by *intro-classes* (*auto simp add*: *prefix-def strict-prefix-def*)

end

lemma *prefixI* [*intro?*]: $ys = xs\ @\ zs \Longrightarrow xs \le ys$
unfolding *prefix-def* **by** *blast*

lemma *prefixE* [*elim?*]:

assumes $xs \le ys$
obtains zs **where** $ys = xs \mathbin{@} zs$
using *assms* **unfolding** *prefix-def* **by** *blast*

lemma *strict-prefixI'* [*intro?*]: $ys = xs \mathbin{@} z \# zs \Longrightarrow xs < ys$
unfolding *strict-prefix-def prefix-def* **by** *blast*

lemma *strict-prefixE'* [*elim?*]:
assumes $xs < ys$
obtains $z\ zs$ **where** $ys = xs \mathbin{@} z \# zs$
proof −
from ‹$xs < ys$› **obtain** us **where** $ys = xs \mathbin{@} us$ **and** $xs \neq ys$
unfolding *strict-prefix-def prefix-def* **by** *blast*
with *that* **show** *?thesis* **by** (*auto simp add*: *neq-Nil-conv*)
qed

lemma *strict-prefixI* [*intro?*]: $xs \le ys \Longrightarrow xs \neq ys \Longrightarrow xs < (ys::'a\ list)$
unfolding *strict-prefix-def* **by** *blast*

lemma *strict-prefixE* [*elim?*]:
fixes $xs\ ys :: 'a\ list$
assumes $xs < ys$
obtains $xs \le ys$ **and** $xs \neq ys$
using *assms* **unfolding** *strict-prefix-def* **by** *blast*

75.2 Basic properties of prefixes

theorem *Nil-prefix* [*iff*]: $[] \le xs$
by (*simp add*: *prefix-def*)

theorem *prefix-Nil* [*simp*]: $(xs \le []) = (xs = [])$
by (*induct xs*) (*simp-all add*: *prefix-def*)

lemma *prefix-snoc* [*simp*]: $(xs \le ys \mathbin{@} [y]) = (xs = ys \mathbin{@} [y] \vee xs \le ys)$
proof
assume $xs \le ys \mathbin{@} [y]$
then obtain zs **where** *zs*: $ys \mathbin{@} [y] = xs \mathbin{@} zs$ **..**
show $xs = ys \mathbin{@} [y] \vee xs \le ys$
by (*metis append-Nil2 butlast-append butlast-snoc prefixI zs*)
next
assume $xs = ys \mathbin{@} [y] \vee xs \le ys$
then show $xs \le ys \mathbin{@} [y]$
by (*metis order-eq-iff strict-prefixE strict-prefixI' xt1*(7))
qed

lemma *Cons-prefix-Cons* [*simp*]: $(x \# xs \le y \# ys) = (x = y \wedge xs \le ys)$
by (*auto simp add*: *prefix-def*)

lemma *same-prefix-prefix* [*simp*]: $(xs \mathbin{@} ys \le xs \mathbin{@} zs) = (ys \le zs)$

by (*induct xs*) *simp-all*

lemma *same-prefix-nil* [*iff*]: (*xs* @ *ys* ≤ *xs*) = (*ys* = []) **by** (*metis append-Nil2 append-self-conv order-eq-iff prefixI*)

lemma *prefix-prefix* [*simp*]: *xs* ≤ *ys* ==> *xs* ≤ *ys* @ *zs* **by** (*metis order-le-less-trans prefixI strict-prefixE strict-prefixI*)

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 (*metis append-eq-appendI*)
done

lemma *append-one-prefix*:
xs ≤ *ys* ==> *length xs* < *length ys* ==> *xs* @ [*ys* ! *length xs*] ≤ *ys*
unfolding *prefix-def*
by (*metis Cons-eq-appendI append-eq-appendI append-eq-conv-conj eq-Nil-appendI nth-drop'*)

theorem *prefix-length-le*: *xs* ≤ *ys* ==> *length xs* ≤ *length ys* **by** (*auto simp add*: *prefix-def*)

lemma *prefix-same-cases*:
(*xs*$_1$::'*a list*) ≤ *ys* ⟹ *xs*$_2$ ≤ *ys* ⟹ *xs*$_1$ ≤ *xs*$_2$ ∨ *xs*$_2$ ≤ *xs*$_1$
unfolding *prefix-def* **by** (*metis append-eq-append-conv2*)

lemma *set-mono-prefix*: *xs* ≤ *ys* ⟹ *set xs* ⊆ *set ys* **by** (*auto simp add*: *prefix-def*)

lemma *take-is-prefix*: *take n xs* ≤ *xs*
unfolding *prefix-def* **by** (*metis append-take-drop-id*)

lemma *map-prefixI*: *xs* ≤ *ys* ⟹ *map f xs* ≤ *map f ys* **by** (*auto simp*: *prefix-def*)

lemma *prefix-length-less*: *xs* < *ys* ⟹ *length xs* < *length ys* **by** (*auto simp*: *strict-prefix-def prefix-def*)

lemma *strict-prefix-simps* [*simp*]:
xs < [] = *False*

```
    [] < (x # xs) = True
    (x # xs) < (y # ys) = (x = y ∧ xs < ys)
  by (simp-all add: strict-prefix-def cong: conj-cong)

lemma take-strict-prefix: xs < ys ⟹ take n xs < ys
  apply (induct n arbitrary: xs ys)
   apply (case-tac ys, simp-all)[1]
  apply (metis order-less-trans strict-prefixI take-is-prefix)
  done

lemma not-prefix-cases:
  assumes pfx: ¬ ps ≤ ls
  obtains
    (c1) ps ≠ [] and ls = []
  | (c2) a as x xs where ps = a#as and ls = x#xs and x = a and ¬ as ≤ xs
  | (c3) a as x xs where ps = a#as and ls = x#xs and x ≠ a
proof (cases ps)
  case Nil then show ?thesis using pfx by simp
next
  case (Cons a as)
  note c = ‹ps = a#as›
  show ?thesis
  proof (cases ls)
    case Nil then show ?thesis by (metis append-Nil2 pfx c1 same-prefix-nil)
  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*: $\neg\ ps \leq (y \# ys)$ **by** *simp*
then obtain *x xs* **where** *pv*: $ps = x \# xs$
by (*rule not-prefix-cases*) *auto*
show *?case* **by** (*metis Cons.hyps Cons-prefix-Cons npfx pv r1 r2*)
qed

75.3 Parallel lists

definition
$(xs \parallel ys) = (\neg\ xs \leq ys \wedge \neg\ ys \leq xs)$

lemma *parallelI* [*intro*]: $\neg\ xs \leq ys \Longrightarrow \neg\ ys \leq xs \Longrightarrow xs \parallel ys$
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*:
$xs \parallel ys \Longrightarrow \exists\, as\ b\ bs\ c\ cs.\ b \neq c \wedge xs = as\ @\ b \# bs \wedge ys = as\ @\ c \# cs$
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 $ys' = []$
then show *?thesis* **by** (*metis append-Nil2 parallelE prefixI snoc.prems ys*)
next
fix *c cs* **assume** *ys'*: $ys' = c \# cs$
then show *?thesis*
by (*metis Cons-eq-appendI eq-Nil-appendI parallelE prefixI*
same-prefix-prefix snoc.prems ys)
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 ∥ ys
    with snoc obtain as b bs c cs where neq: (b::'a) ≠ 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 ∥ b ⟹ a @ c ∥ b @ d
  apply (rule parallelI)
    apply (erule parallelE, erule conjE,
      induct rule: not-prefix-induct, simp+)+
  done

lemma parallel-appendI: xs ∥ ys ⟹ x = xs @ xs' ⟹ y = ys @ ys' ⟹ x ∥ y
  by (simp add: parallel-append)

lemma parallel-commute: a ∥ b ⟷ b ∥ a
  unfolding parallel-def by auto
```

75.4 Postfix order on lists

```
definition
  postfix :: 'a list => 'a list => bool  ((-/ >>= -) [51, 50] 50) where
  (xs >>= ys) = (∃zs. xs = zs @ ys)

lemma postfixI [intro?]: xs = zs @ ys ==> xs >>= ys
  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⟧ ⟹ xs >>= zs
  by (auto simp add: postfix-def)
lemma postfix-antisym: ⟦xs >>= ys; ys >>= xs⟧ ⟹ 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 ⟹ x#xs >>= ys*
by (*auto simp add*: *postfix-def*)
lemma *postfix-ConsD*: *xs >>= y#ys ⟹ xs >>= ys*
by (*auto simp add*: *postfix-def*)

lemma *postfix-appendI*: *xs >>= ys ⟹ 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*: *distinct xs ⟹ xs >>= ys ⟹ distinct ys*
by (*clarsimp elim*!: *postfixE*)

lemma *postfix-map*: *xs >>= ys ⟹ map f xs >>= map f ys*
by (*auto elim*!: *postfixE intro*: *postfixI*)

lemma *postfix-drop*: *as >>= drop n as*
unfolding *postfix-def*
apply (*rule exI* [**where** *x = take n as*])
apply *simp*

done

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*
by (*metis Cons-prefix-Cons parallelE parallelI*)

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 *Nil*
then show *?case* **by** *simp*
next
case (*Cons 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** *Cons* **by** *simp*
then show *?thesis* **by** (*rule Cons-parallelI2* [*OF True ih*])
next
case *False*
then show *?thesis* **by** (*rule Cons-parallelI1*)
qed
qed

75.5 Executable code

lemma *less-eq-code* [*code*]:
([]::*'a*::{*eq*, *ord*} *list*) ≤ *xs ⟷ True*

$(x::'a::\{eq,\ ord\})\ \#\ xs \leq [] \longleftrightarrow False$
$(x::'a::\{eq,\ ord\})\ \#\ xs \leq y\ \#\ ys \longleftrightarrow x = y \wedge xs \leq ys$
by *simp-all*

lemma *less-code* [*code*]:
$xs < ([]::'a::\{eq,\ ord\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{eq,\ ord\})\#\ xs \longleftrightarrow True$
$(x::'a::\{eq,\ ord\})\ \#\ xs < y\ \#\ ys \longleftrightarrow x = y \wedge xs < ys$
unfolding *strict-prefix-def* **by** *auto*

lemmas [*code*] = *postfix-to-prefix*

end

76 List-lexord: Lexicographic order on lists

theory *List-lexord*
imports *List Main*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def [*code del*]: $(xs::('a::ord)\ list) < ys \longleftrightarrow (xs,\ ys) \in lexord\ \{(u,v).\ u < v\}$

definition
list-le-def [*code del*]: $(xs::('a::ord)\ list) \leq ys \longleftrightarrow (xs < ys \vee xs = ys)$

instance ..

end

instance *list* :: (*order*) *order*
proof
fix *xs* :: *'a list*
show *xs* $\leq$ *xs* **by** (*simp add*: *list-le-def*)
next
fix *xs ys zs* :: *'a list*
assume *xs* $\leq$ *ys* **and** *ys* $\leq$ *zs*
then show *xs* $\leq$ *zs* **by** (*auto simp add*: *list-le-def list-less-def*)
(*rule lexord-trans*, *auto intro*: *transI*)
next
fix *xs ys* :: *'a list*
assume *xs* $\leq$ *ys* **and** *ys* $\leq$ *xs*
then show *xs* = *ys* **apply** (*auto simp add*: *list-le-def list-less-def*)
apply (*rule lexord-irreflexive* [*THEN notE*])

defer
apply (*rule lexord-trans*) **apply** (*auto intro*: *transI*) **done**
next
fix *xs ys* :: *'a list*
show *xs* < *ys* ⟷ *xs* ≤ *ys* ∧ ¬ *ys* ≤ *xs*
apply (*auto simp add*: *list-less-def list-le-def*)
defer
apply (*rule lexord-irreflexive* [*THEN notE*])
apply *auto*
apply (*rule lexord-irreflexive* [*THEN notE*])
defer
apply (*rule lexord-trans*) **apply** (*auto intro*: *transI*) **done**
qed

instance *list* :: (*linorder*) *linorder*
proof
fix *xs ys* :: *'a list*
have (*xs*, *ys*) ∈ *lexord* {(*u*, *v*). *u* < *v*} ∨ *xs* = *ys* ∨ (*ys*, *xs*) ∈ *lexord* {(*u*, *v*). *u* < *v*}
by (*rule lexord-linear*) *auto*
then show *xs* ≤ *ys* ∨ *ys* ≤ *xs*
by (*auto simp add*: *list-le-def list-less-def*)
qed

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition
[*code del*]: (*inf* :: *'a list* ⇒ -) = *min*

definition
[*code del*]: (*sup* :: *'a list* ⇒ -) = *max*

instance
by *intro-classes*
(*auto simp add*: *inf-list-def sup-list-def min-max.sup-inf-distrib1*)

end

lemma *not-less-Nil* [*simp*]: ¬ (*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* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* < *y*
by (*unfold list-less-def*) *simp*

lemma *le-Nil* [*simp*]: *x* ≤ [] ⟷ *x* = []

by (*unfold list-le-def*, *cases x*) *auto*

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
by (*unfold list-le-def*, *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-le-def*) *auto*

lemma *less-code* [*code*]:
$xs < ([]::'a::\{eq, order\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{eq, order\}) \# xs \longleftrightarrow True$
$(x::'a::\{eq, order\}) \# xs < y \# ys \longleftrightarrow x < y \vee x = y \wedge xs < ys$
by *simp-all*

lemma *less-eq-code* [*code*]:
$x \# xs \leq ([]::'a::\{eq, order\}\ list) \longleftrightarrow False$
$[] \leq (xs::'a::\{eq, order\}\ list) \longleftrightarrow True$
$(x::'a::\{eq, order\}) \# xs \leq y \# ys \longleftrightarrow x < y \vee x = y \wedge xs \leq ys$
by *simp-all*

end

77 Sublist-Order: Sublist Ordering

theory *Sublist-Order*
imports *Main*
begin

This theory defines sublist ordering on lists. A list *ys* is a sublist of a list *xs*, iff one obtains *ys* by erasing some elements from *xs*.

77.1 Definitions and basic lemmas

instantiation *list* :: (*type*) *ord*
begin

inductive *less-eq-list* **where**
empty [*simp*, *intro*!]: $[] \leq xs$
| *drop*: $ys \leq xs \Longrightarrow ys \leq x \# xs$
| *take*: $ys \leq xs \Longrightarrow x \# ys \leq x \# xs$

definition
[*code del*]: $(xs :: 'a\ list) < ys \longleftrightarrow xs \leq ys \wedge \neg\ ys \leq xs$

instance proof qed

end

lemma *le-list-length*: $xs \leq ys \Longrightarrow length\ xs \leq length\ ys$
by (*induct rule*: *less-eq-list.induct*) *auto*

lemma *le-list-same-length*: $xs \leq ys \Longrightarrow length\ xs = length\ ys \Longrightarrow xs = ys$
by (*induct rule*: *less-eq-list.induct*) (*auto dest*: *le-list-length*)

lemma *not-le-list-length*[*simp*]: $length\ ys < length\ xs \Longrightarrow \sim xs <= ys$
by (*metis le-list-length linorder-not-less*)

lemma *le-list-below-empty* [*simp*]: $xs \leq [] \longleftrightarrow xs = []$
by (*auto dest*: *le-list-length*)

lemma *le-list-drop-many*: $xs \leq ys \Longrightarrow xs \leq zs\ @\ ys$
by (*induct zs*) (*auto intro*: *drop*)

lemma [*code*]: $[] <= xs \longleftrightarrow True$
by(*metis less-eq-list.empty*)

lemma [*code*]: $(x\#xs) <= [] \longleftrightarrow False$
by *simp*

lemma *le-list-drop-Cons*: **assumes** $x\#xs <= ys$ **shows** $xs <= ys$
proof−
 { fix xs' ys'
  assume $xs' <= ys$
  hence *ALL x xs*. $xs' = x\#xs \longrightarrow xs <= ys$
  proof *induct*
   case *empty* **thus** *?case* **by** *simp*
  next
   case *drop* **thus** *?case* **by** (*metis less-eq-list.drop*)
  next
   case *take* **thus** *?case* **by** (*simp add*: *drop*)
  qed }
 from *this*[*OF assms*] **show** *?thesis* **by** *simp*
qed

lemma *le-list-drop-Cons2*:
assumes $x\#xs <= x\#ys$ **shows** $xs <= ys$
using *assms*
proof *cases*
 case *drop* **thus** *?thesis* **by** (*metis le-list-drop-Cons list.inject*)
qed *simp-all*

lemma *le-list-drop-Cons-neq*: **assumes** $x \# xs <= y \# ys$
shows $x \sim= y \Longrightarrow x \# xs <= ys$
using *assms* **proof** *cases* **qed** *auto*

lemma *le-list-Cons2-iff*[*simp*,*code*]: $(x\#xs) <= (y\#ys) \longleftrightarrow$
 (*if* $x=y$ *then* $xs <= ys$ *else* $(x\#xs) <= ys$)

by (*metis drop take le-list-drop-Cons2 le-list-drop-Cons-neq*)

lemma *le-list-take-many-iff*: *zs* @ *xs* ≤ *zs* @ *ys* ⟷ *xs* ≤ *ys*
by (*induct zs*) (*auto intro*: *take*)

lemma *le-list-Cons-EX*:
assumes *x # ys <= zs* **shows** *EX us vs. zs = us @ x # vs & ys <= vs*
proof−
{ fix *xys zs* :: *'a list* **assume** *xys <= zs*
hence *ALL x ys. xys = x#ys* ⟶ (*EX us vs. zs = us @ x # vs & ys <= vs*)
proof *induct*
case *empty* **show** *?case* **by** *simp*
next
case *take* **thus** *?case* **by** (*metis list.inject self-append-conv2*)
next
case *drop* **thus** *?case* **by** (*metis append-eq-Cons-conv*)
qed
} with *assms* **show** *?thesis* **by** *blast*
qed

instantiation *list* :: (*type*) *order*
begin

instance proof
fix *xs ys* :: *'a list*
show *xs < ys* ⟷ *xs* ≤ *ys* ∧ ¬ *ys* ≤ *xs* **unfolding** *less-list-def* **..**
next
fix *xs* :: *'a list*
show *xs* ≤ *xs* **by** (*induct xs*) (*auto intro*!: *less-eq-list.drop*)
next
fix *xs ys* :: *'a list*
assume *xs <= ys*
hence *ys <= xs* ⟶ *xs = ys*
proof *induct*
case *empty* **show** *?case* **by** *simp*
next
case *take* **thus** *?case* **by** *simp*
next
case *drop* **thus** *?case*
by(*metis le-list-drop-Cons le-list-length Suc-length-conv Suc-n-not-le-n*)
qed
moreover assume *ys <= xs*
ultimately show *xs = ys* **by** *blast*
next
fix *xs ys zs* :: *'a list*
assume *xs <= ys*
hence *ys <= zs* ⟶ *xs <= zs*
proof (*induct arbitrary*:*zs*)
case *empty* **show** *?case* **by** *simp*

```
  next
    case (take xs ys x) show ?case
    proof
      assume x # ys <= zs
      with take show x # xs <= zs
        by(metis le-list-Cons-EX le-list-drop-many less-eq-list.take local.take(2))
    qed
  next
    case drop thus ?case by (metis le-list-drop-Cons)
  qed
  moreover assume ys <= zs
  ultimately show xs <= zs by blast
qed

end

lemma le-list-append-le-same-iff: xs @ ys <= ys ⟷ xs=[]
by (auto dest: le-list-length)

lemma le-list-append-mono: ⟦ xs <= xs'; ys <= ys' ⟧ ⟹ xs@ys <= xs'@ys'
apply (induct rule:less-eq-list.induct)
  apply (metis eq-Nil-appendI le-list-drop-many)
 apply (metis Cons-eq-append-conv le-list-drop-Cons order-eq-refl order-trans)
apply simp
done

lemma less-list-length: xs < ys ⟹ length xs < length ys
by (metis le-list-length le-list-same-length le-neq-implies-less less-list-def)

lemma less-list-empty [simp]: [] < xs ⟷ xs ≠ []
by (metis empty order-less-le)

lemma less-list-below-empty[simp]: xs < [] ⟷ False
by (metis empty less-list-def)

lemma less-list-drop: xs < ys ⟹ xs < x # ys
by (unfold less-le) (auto intro: less-eq-list.drop)

lemma less-list-take-iff: x # xs < x # ys ⟷ xs < ys
by (metis le-list-Cons2-iff less-list-def)

lemma less-list-drop-many: xs < ys ⟹ xs < zs @ ys
by(metis le-list-append-le-same-iff le-list-drop-many order-less-le self-append-conv2)

lemma less-list-take-many-iff: zs @ xs < zs @ ys ⟷ xs < ys
by (metis le-list-take-many-iff less-list-def)
```

77.2 Appending elements

lemma *le-list-rev-take-iff*[*simp*]: *xs* @ *zs* ≤ *ys* @ *zs* ⟷ *xs* ≤ *ys* (**is** *?L* = *?R*)
proof
{ fix *xs' ys' xs ys zs* :: *'a list* **assume** *xs'* <= *ys'*
hence *xs'* = *xs* @ *zs* & *ys'* = *ys* @ *zs* ⟶ *xs* <= *ys*
proof (*induct arbitrary*: *xs ys zs*)
case *empty* **show** *?case* **by** *simp*
next
case (*drop xs' ys' x*)
{ assume *ys*=[] **hence** *?case* **using** *drop*(*1*) **by** *auto* **}**
moreover
{ fix *us* **assume** *ys* = *x*#*us*
hence *?case* **using** *drop*(*2*) **by**(*simp add*: *less-eq-list.drop*) **}**
ultimately show *?case* **by** (*auto simp*:*Cons-eq-append-conv*)
next
case (*take xs' ys' x*)
{ assume *xs*=[] **hence** *?case* **using** *take*(*1*) **by** *auto* **}**
moreover
{ fix *us vs* **assume** *xs*=*x*#*us ys*=*x*#*vs* **hence** *?case* **using** *take*(*2*) **by** *auto***}**
moreover
{ fix *us* **assume** *xs*=*x*#*us ys*=[] **hence** *?case* **using** *take*(*2*) **by** *bestsimp* **}**
ultimately show *?case* **by** (*auto simp*:*Cons-eq-append-conv*)
qed }
moreover assume *?L*
ultimately show *?R* **by** *blast*
next
assume *?R* **thus** *?L* **by**(*metis le-list-append-mono order-refl*)
qed

lemma *less-list-rev-take*: *xs* @ *zs* < *ys* @ *zs* ⟷ *xs* < *ys*
by (*unfold less-le*) *auto*

lemma *le-list-rev-drop-many*: *xs* ≤ *ys* ⟹ *xs* ≤ *ys* @ *zs*
by (*metis append-Nil2 empty le-list-append-mono*)

77.3 Relation to standard list operations

lemma *le-list-map*: *xs* ≤ *ys* ⟹ *map f xs* ≤ *map f ys*
by (*induct rule*: *less-eq-list.induct*) (*auto intro*: *less-eq-list.drop*)

lemma *le-list-filter-left*[*simp*]: *filter f xs* ≤ *xs*
by (*induct xs*) (*auto intro*: *less-eq-list.drop*)

lemma *le-list-filter*: *xs* ≤ *ys* ⟹ *filter f xs* ≤ *filter f ys*
by (*induct rule*: *less-eq-list.induct*) (*auto intro*: *less-eq-list.drop*)

lemma *xs* ≤ *ys* ⟷ (*EX N*. *xs* = *sublist ys N*) (**is** *?L* = *?R*)
proof
assume *?L*

```
  thus ?R
  proof induct
    case empty show ?case by (metis sublist-empty)
  next
    case (drop xs ys x)
    then obtain N where xs = sublist ys N by blast
    hence xs = sublist (x#ys) (Suc ' N)
      by (clarsimp simp add:sublist-Cons inj-image-mem-iff)
    thus ?case by blast
  next
    case (take xs ys x)
    then obtain N where xs = sublist ys N by blast
    hence x#xs = sublist (x#ys) (insert 0 (Suc ' N))
      by (clarsimp simp add:sublist-Cons inj-image-mem-iff)
    thus ?case by blast
  qed
next
  assume ?R
  then obtain N where xs = sublist ys N ..
  moreover have sublist ys N <= ys
  proof (induct ys arbitrary:N)
    case Nil show ?case by simp
  next
    case Cons thus ?case by (auto simp add:sublist-Cons drop)
  qed
  ultimately show ?L by simp
qed

end
```

References

[1] Abrial and Laffitte. Towards the mechanization of the proofs of some classical theorems of set theory. Unpublished.

[2] J. Avigad and K. Donnelly. Formalizing O notation in Isabelle/HOL. In D. Basin and M. Rusiowitch, editors, *Automated Reasoning: second international conference, IJCAR 2004*, pages 357–371. Springer, 2004.

[3] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.